MISSION-PHASING TECHNIQUES FOR CONSTRAINED AGENTS IN STOCHASTIC ENVIRONMENTS

by

Jianhui Wu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

To my parents and wife.

ACKNOWLEDGEMENTS

I would like to first give one million thanks to my advisor, Edmund Durfee. After working with him for six years, I still find it difficult to believe that he is so knowledgeable in his field. I was so fortunate that I could acquire instruction from him during my graduate studies. It was truly a once-in-a-lifetime experience to receive his support and encouragement.

I wish to express my sincerest thanks to the members of my committee, Satinder Singh, Kang Shin, and Demos Teneketsis, for their insightful comments and suggestions on my work.

Some techniques (heuristic search) presented in this work were built upon codes and concepts developed by a large research team centered at Honeywell labs. I appreciate the work of Dave Musliner, Robert Goldman, John Phelps, and Mark Boddy, among many others.

I would also like to say thanks to all the people in the AI lab. You are so nice and I have felt so happy with you during the past six years. Especially, thanks to Dmitri Dolgov, Jeffrey Cox, Stefan Witwicki, and Thom Bartold, you have offered me so much help.

Finally, I am forever indebted to my parents and my wife for their understanding, endless patience, and encouragement when it was most required.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

Introduction

An omnipresent issue in realistic application domains is that the performance of autonomous agents is subject to system constraints. Resource constraints restrict the set of actions that an agent can take, which means that the agent might not be able to accomplish all its goals under its current resource configuration. Computational time limitations restrict the number of states that an agent can model and reason over, which means that the agent might not be able to formulate a policy that can respond to all possible eventualities in the pre-execution planning stage.

Let us consider the following example application. A Mars rover needs to reach particular locations to carry out its scientific tasks. The environment is stochastic and dangerous. To move safely, the rover should carry particular instruments (such as a rubberized anti-skip grip), each of which can help the rover better handle an exogenous event during its travel. The rover should also carry some other particular instruments (such as a panoramic camera) required to perform its scientific tasks. If the capacity of the rover is restricted, a problem arises: the rover might be unable to carry all its desired instruments (e.g., because of the limited weight it can carry). Furthermore, even without agent capacity limitations, a similar resource-constrained issue may occur in an environment where a group of rovers share a set of instruments.

When the shared instruments are scarce and when the rovers are distributed in the environment, a rover might only be able to perform a subset of its desired tasks because the scarcity of the shared resources could mean that a rover can have only a subset of its desired instruments, and the distributed nature of the problem world could mean that an instrument assigned to one rover cannot be utilized by others.

Another possible type of constraints in the rover world is that a rover may only have limited time to "think" about how to carry out its tasks. For example, if a rover has multiple complex scientific tasks, each of which requires the rover to reason over a large state space, the rover may fall into trouble when it has to start to operate in the world at a particular time. The limitation of computation time restricts the number of states that the rover can search, which implies that the rover might be unable to formulate a complete and optimal policy that can respond, in a timely manner, to all possible eventualities that might occur when performing the scientific tasks.

As other examples, a real-time autonomous driving agent might be unable to schedule all of its desired actions (watching for pedestrians, checking surrounding traffic, reading gauges, etc.) frequently enough because it cannot redirect its limited perceptual resources fast enough in all relevant directions. A branch of a delivery company might be unable to deliver all its packages in time because it should share scarce delivery vehicles with other branches of the company. An autonomous aircraft flying a prolonged mission might not have time to prepare a plan over the entire mission before it starts to execute the plan.

These (and similar) problems motivate the study of mission-phasing techniques. We argue that one effective way of improving performance of constrained agents is to adopt a phasing strategy. In resource-constrained environments, constrained agents can improve their performance by exploiting resource reconfiguration/reallocation opportunities, at which points a large problem (mission) can be decomposed into a collection of sub-problems (phases).[1] That is to say, a capacity-limited agent may enhance its performance by reconfiguring how it uses its capacity, re-customizing its action set, and switching to a new policy that can handle future events better as it moves from phase to phase; a group of agents sharing scarce resources may reconfigure the distribution of resources and adopt a new joint policy to use the limited resources more effectively as time passes.

Analogously, in time-critical environments, an agent can adopt a phasing strategy to improve its performance too. By decomposing a large problem into phases, an agent could choose to focus computation only on near-term *high-value* phases. Then, while executing the plans for earlier phases, the agent could use available computation time during execution to reconsider aspects of the problem and improve its solutions for the current and future phases.

The objective of this dissertation is the development of two classes of mission-phasing techniques, including the resource-driven mission-phasing approach and the computation-driven mission-phasing approach, corresponding to the aforementioned resource constraints and computational time limitations respectively. It should be emphasized that this mission-phasing study is not only to optimize the use of the predefined/existing resource-reconfiguration or problem-reconsideration opportunities available in the midst of execution, but to automate the process of determining where to establish such opportunities, according for the cost of creating them, in complex stochastic environments.

The structural frameworks of the resource-driven mission-phasing (RMP) and

[1]A phase, by definition, means a stage in a process of change or development.

computation-driven mission-phasing (CMP) techniques have a lot in common. They are both based upon problem decomposition, they both need to allocate "things" (i.e., resources or time), and they both should formulate policies in each phase. Nevertheless, because of fundamental differences between the constraints caused by the limitations of *non-consumable* execution resources and the constraints caused by the limitations of *consumable* computation time[2], the implementations of these techniques (problem decomposition, resource/time allocation, and policy formulation) are considerably different. Briefly put, RMP deals with the constraints that restrict the set of actions to include in the policy while CMP deals with the constraints that affect the policy formulation procedure. Section 1.1 and Section 1.2 separately discuss these two classes of the mission-phasing problems.

1.1 Phasing for Resource Constraints

As was briefly stated above, a capacity-limited agent may improve its performance by adopting a phasing strategy — when the agent reaches a particular state, it can choose to reconfigure its resources and adopt a new policy to use its limited capacity more effectively given the particular trajectory the world has taken. That is, unlike an unconstrained agent that can execute a policy that is optimal for all possible eventualities, a capacity-limited agent can benefit from judiciously breaking its overall mission into phases, where as it moves from phase to phase it can reconfigure its capacity usage and adopt a different, more effective policy for its current phase.

For the Mars rover example, the performance of a capacity-limited rover (that can carry either travel instruments or scientific instruments but not both) will be improved if a resource-reconfiguration point (e.g., a toolbox or a supply station)

[2]Computation time, which refers to the time for using computational processors (e.g., CPUs), is consumable, but processors themselves are non-consumable. If we deal with limitations on allocating processors that enable computation, we should use RMP (instead of CMP) techniques to schedule the allocation of the processors.

can be set up near the location where scientific tasks are performed. This resource-reconfiguration point decomposes the rover's mission into two phases. In the phase starting at the rover's initial location and ending at the resource reconfiguration point, the rover can choose to carry only the travel instruments to reduce its risk during travel. When it reaches the resource reconfiguration point, the rover can switch to the scientific instruments to get ready for its scientific tasks in the subsequent phase.

In a similar fashion to single-agent systems, reconfiguring resources during execution can also have an advantageous effect in environments where multiple agents share scarce resources. In such environments, an individual agent is often unable to execute some of its possible actions because resources required by those actions are currently held by other agents. How the resources are allocated among the agents will dictate the actions each agent will be capable of performing, and thus how the agents will act to accomplish their goals in the environment. One way to alleviate such resource scarcity is to redistribute resources among the agents over time. As an example, the rovers' need for their previously assigned instruments may diminish as time passes, since the relevant tasks may have been accomplished (or have expired), which suggests that redistributing the instruments in the group of rovers at some proper time points in the midst of execution may be an effective way to improve the total expected utility of the rovers, even if redistribution can incur a cost.

The idea of reconfiguring resources to improve agent performance is fairly straightforward, but it is a challenging problem to reconfigure resources in the optimal way. Briefly put, it requires solving the problem of optimally creating and exploiting mission phases, accounting for the cost of creating them, along with the problem of making the optimal resource configurations at the entries of each phase, as well as the problem of formulating optimal executable policies for each phase. The primary goal of our resource-driven mission-phasing study in this dissertation is to design computationally efficient algorithms to exactly solve this class of challenging problems. Toward this end, this work develops a suite of algorithms that can formulate complex resource-driven mission-phasing problems into compact mathematical formulations. Thereafter, by simultaneously solving problem decomposition, resource (re)configuration, and policy formulation problems, these algorithms can effectively and fruitfully exploit problem structure, which often results in a significant reduction in computational cost.

1.2 Phasing for Computational Time Limitations

The second primary objective of this dissertation is the design of computation-driven mission-phasing techniques to handle the challenges raised by computational time limitations.

Let us revisit the example problem where a rover implementing a scientific mission needs to plan and perform a sequence of independent tasks (each of which can be thought of an independent phase decision procedure). The rover has to start to execute its mission at a particular time point, which could mean that the rover does not have sufficient time to reason over all of its future tasks prior to beginning execution. As mentioned previously, the rover may do better by focusing its computational effort only on near-term high-value tasks in its pre-execution planning stage, and then taking advantage of additional computation time (or even paying some costs to acquire more computation time) during execution to reconsider and improve its solutions for future tasks. This assertion is not surprising. The challenges, though, are in automating the process of allocating computation time to appropriate phases given the uncertainty and complexity of the problem domain, and in designing computationally efficient algorithms to solve the coupled problems of deciding both when to deliberate given its cost, and which decision procedures to execute during deliberation intervals.

Furthermore, besides needing to make intelligent decisions about time allocation for multiple phases, the agent should also be able to utilize the allocated computation time within each phase effectively. One potential way is to adopt a heuristic search method as the inner-phase policy-formulation solver. Unlike a classical dynamic programming algorithm (e.g., value iteration) that evaluates the full state space and finds an optimal policy for every state, the heuristic search typically focuses on the states that are likely to be reached when following an optimal (or high-quality) policy from the initial state, while ignoring other states, to yield a good solution within limited time.

In a similar structure to the resource-driven mission-phasing approach, the computation-driven mission-phasing approach also consists of three components: problem decomposition, time allocation, and policy formulation. The problem decomposition component defines boundary states so that the phases' problems can be solved (approximately) independently. The time allocation component manages the distribution of computation time (that may be available before or during execution) among phase decision procedures to let the agent bias its computation on more valuable or more important phases. The policy formulation component uses the heuristic search to selectively expand and explore states in order to build a high-quality partial policy within the allotted computation time. These techniques work together to help an autonomous agent improve its performance in time-critical environments.

1.3 Main Contributions

The primary goal of this dissertation is the development of mission-phasing techniques to improve the performance of constrained agents by establishing and exploiting resource reconfiguration or problem reconsideration opportunities in the midst of mission execution. Toward this end, a suite of computationally efficient algorithms are designed, analyzed and empirically evaluated in this work.

The major contributions of the work presented in this dissertation are outlined below.

Resource-Driven Mission-Phasing

- **Effectively Exploiting Resource Reconfiguration Opportunities.**

  This work explicitly takes into account potential resource reconfiguration opportunities in the midst of execution, and extends prior one-shot resource-allocation-and-policy-formulation techniques to also solve the problem of how to optimally reconfigure resources during execution. By considering and exploiting such opportunities, autonomous agents may well accomplish their goals even when they are subject to capacity/resource constraints. This represents an effective and inexpensive strategy to improve agent performance in resource-constrained environments.

- **Automatically Determining Phase-Switching Points.**

  This work designs automated resource-driven mission-phasing techniques, which can automatically determine the optimal phase-switching points (where the constrained agents reconfigure resources and switch policies), accounting for the cost of creating them, in stochastic and constrained environments. It eliminates the need for having phases predefined in the description of a mission, and, in turn, resolves potential sub-optimality due to improper phase definitions by users, which improves the applicability of the resource-driven mission-phasing techniques.

- **Exploiting Problem Structure for Finding Exact Solutions Efficiently.**

  In comparison with the straightforward MDP-based approach that explicitly models resources in the state representation and treats resource reconfiguration activities as explicit actions, the resource-driven mission-phasing approach presented in this dissertation takes an alternative way to characterize resource reconfigurations and phase transitions, which thus avoids the exponential increase in the size of the state/action space. Moreover, through simultaneously solving three intertwined problems, including problem decomposition, resource configuration, and policy formulation, the approach presented in this dissertation can effectively exploit problem structure to reduce computational cost.

Computation-Driven Mission-Phasing

- **Judiciously Allocating Time in Complex Environments.**

  The computation-driven mission-phasing approach decomposes a large problem into multiple phases for utilizing limited computation time better. To meet the demand for new algorithms that can intelligently and quickly manage the distribution of limited time among multiple phases in complex stochastic environments, this work develops a novel deliberation scheduling approach based upon mathematical programming. This deliberation scheduling approach can be applied to a considerably wider variety of problem domains to find optimal or near-optimal deliberation schedules, compared to prior computationally-tractable deliberation scheduling approaches.

- **Effectively Using Limited Computation Time by Heuristic Search.**

  To cope with the challenging problem where an autonomous agent may have only a finite amount of "think time" to build and solve a large Markov decision process for its planning problem within a phase, we design a heuristic search method in this work, which biases state space expansion towards states that are believed to lie along trajectories of high-quality policies, while ignoring other states, to yield a better policy within time limits. Empirical results highlight the ability of this algorithm to cope with limited computation time, and thus it represents a promising new strategy for anytime policy formulation.

1.4 Overview

This dissertation is organized as follows:

Chapter II starts with a relatively simple single-agent resource-driven mission-phasing problem where phase-switching states are known *a priori*. Exploiting such fixed phase-switching states, we can work out a particular, efficient algorithm. Of course, not all applications have phase-switching states predefined in their problem descriptions. This chapter then describes solution algorithms for solving general resource-driven mission-phasing problems, in which an agent needs to determine for itself where to reconfigure resources, how to reconfigure resources, and what are optimal executable policies subject to the (re)configured resources.

Chapter III extends the resource-driven mission-phasing techniques presented in Chapter II to a class of multi-agent systems for sequentially allocating resources among a group of cooperative agents. This chapter follows a similar progression as in Chapter II, in terms of giving the agents increasing latitude in determining when to reallocate resources.

Chapter IV focuses on the deliberation scheduling component of the computation-driven mission-phasing approach. This chapter begins by describing a fundamental mathematical-programming-based approach for scheduling phase decision procedures. Then this formulation is extended to solve more challenging problems where phase transitions can be non-deterministic and problems where phase transitions can be affected by an agent itself. Several other extensions are also discussed in this chapter, including how to linearize nonlinear objective functions.

Chapter V presents an anytime policy formulation algorithm that prioritizes the queue of states waiting to be expanded based on an estimate of the likelihood that the state would be encountered when following a high-quality policy from the initial state. Together with the deliberation scheduling techniques presented in Chapter IV, this algorithm enables an autonomous agent to concentrate its limited computation time on high-value portions of the problem state space. To evaluate the overall computation-driven mission-phasing approach, this chapter also describes and illustrates a heuristic decomposition algorithm, which can effectively and efficiently decompose a class of time-critical problems (represented in TÆMS models) into multiple sub-problems.

Chapter VI concludes this dissertation with a summary of contributions of this work, and a discussion of questions that remain open together with possible future research directions.

CHAPTER II

Resource Reconfiguration in Single-Agent Systems

As was previously stated in Section 1.1, one potential way to enhance the performance of a capacity-limited agent is to break its overall mission into phases. In such a way, the constrained agent could handle each smaller and simpler phase better under its capacity limits, and utilize its restricted capacity more effectively by reconfiguring its resources (which affects its capacity use) when moving from phase to phase. Obviously, if an agent could reconfigure its resources and its capacity use in every state, then it could obtain the same reward as in the unconstrained case (assuming that there is no action whose total capacity costs all by itself exceed the agent's capacity limits). In practice, though, constrained systems often have restrictions on the states in which an agent can reconfigure resources and switch policies. For instance, a rover might require being close to a supply station for changing its instruments, but the number of supply stations that can be built in the rover environment may be limited (e.g., due to the limited amount of supplies that can be sent to Mars), which means only a subset of states can have accessible supply stations.

In this chapter, a state that allows an agent to reconfigure its resources and switch its policy is referred as a *phase-switching* state. A set of phase-switching states decompose the overall state space of a problem into multiple (not necessarily non-overlapped) phases, where each phase is associated with its own way of resource configuration and executable policy. The objectives of this chapter are to systematically investigate the effects of the resource-driven mission-phasing strategy, and to develop solution algorithms that can automate the process of creating and using optimal phase-switching states even when the system is complex and stochastic.

The rest of this chapter is organized as follows: it begins by giving a formal definition of the single-agent resource-driven mission-phasing problem in Section 2.1, and then recaps Markov decision processes (MDPs) and related policy formulation techniques in Section 2.2 because most work in this dissertation is based upon MDP models. In Section 2.3, the computational complexity of the problem is theoretically analyzed and discussed, illustrating why standard approaches are computationally intractable for the problem. Section 2.4 and Section 2.5 look into the single-agent resource-driven mission-phasing problem and some of its variations, and for each, present, analyze, and illustrate a solution algorithm. Experimental results are shown in Section 2.6 where the effectiveness and efficiency of our automated mission-phasing techniques are empirically evaluated. Then, this work is contrasted with prior work in Section 2.7. Finally, Section 2.8 concludes this chapter with a summary of contributions of the work presented in this chapter.[1]

2.1 Problem Definition

In numerous application domains, planning processes are complicated by uncertainties in the environments. The Markov decision process provides a formal framework for stochastic planning. However, the optimal policy derived by modeling and solving a classical unconstrained MDP might not be executable by a capacity-limited agent because the capacity limitation may restrict the set of actions that can

[1]This chapter is largely based on work that was originally reported in (Wu and Durfee, 2005).

be scheduled in the policy. To cope with this issue, this work introduces a problem model that extends the classical MDP model to also take into account constraints in the agent capacity as well as constraints in reconfiguring the usage of the agent capacity.

In a formal definition, a *single-agent resource-driven mission-phasing* (S-RMP) optimization problem is a constrained optimization problem with the inputs of Markov decision process $\mathcal{M}$, initial probability distribution α, agent capacity constraint $\mathcal{C}$, and resource reconfiguration constraint[2] $\mathcal{R}$, where:

- □ $\mathcal{M}$ is a classical MDP, which can be represented as a tuple $\langle S, A, P, R \rangle$, where S is a finite state space, A is a finite action space, P is the state transition probability function, and R is the reward function.[3] A detailed description of MDPs will be given in Section 2.2.

- □ $\alpha = \{\alpha_i\}$ specifies the initial probability distribution over states, where α_i is the probability that the agent starts at state i.

- □ Agent capacity constraint $\mathcal{C}$ can be represented as $\langle O, C, U, \Gamma, \hat{\Gamma} \rangle$, in a similar manner to that used by Dolgov and Durfee (2006):

  - ◇ $O = \{o\}$ is a finite set of indivisible non-consumable execution resources, e.g., $O = \{$*camera, spectrometer, etc.*$\}$.

  - ◇ $C = \{c\}$ is a finite set of capacities of the agent, e.g., $C = \{$*weight, space, etc.*$\}$.

  - ◇ $U = \{u_{o,a,i}\}$ represents resource requirements for executing actions, where $u_{o,a,i} \in \{0, 1\}$ indicates whether the agent requires resource o to execute

[2]Because resource reconfiguration comes along with phase switching, in the following discussion, resource reconfiguration constraints are sometimes called phase-switching constraints to improve readability.

[3]In this work, we do not separately model temporal constraints or ordering constraints between tasks. If present, these would be captured implicitly in the MDP model.

action a in state i.[4] For example, $u_{o=camera,\, a=take\_picture,\, i=any\_state} = 1$ says that the prerequisite of taking a picture is having a camera.

- ◇ $\Gamma = \{\tau_{o,c}\}$ defines capacity costs of the resources, where $\tau_{o,c}$ defines the amount of agent capacity c required to hold one unit of resource o. For example, $\tau_{o=camera,\, c=weight} = 2$ and $\tau_{o=camera,\, c=space} = 1$ says that carrying a camera will consume two units of the carrying weight and one unit of the carrying space of the agent.
- ◇ $\hat{\Gamma} = \{\hat{\tau}_c\}$ specifies the limits of the agent capacities, e.g., $\hat{\tau}_{c=weight} = 4$ denotes the maximum weight of four units that an agent can carry.

☐ Resource reconfiguration constraint $\mathcal{R}$ (sometimes also called phase-switching constraint) specifies restrictions on creating phase-switching states at which the constrained agent can reconfigure its resources and adjust its use of its limited capacities. A typical resource reconfiguration constraint $\mathcal{R}$ can be formulated as $\langle \lambda, \hat{\lambda} \rangle$ (and one of its generalizations will be discussed in Section 2.5.3), where:

- ◇ $\lambda = \{\lambda_i\}$ indicates resource reconfiguration costs, where λ_i denotes the cost for making state i into one that is conducive for the constrained agent to reconfigure resources and switch policies. For the rover example, $\lambda_{i=any\_state} = 10$ defines the cost of setting up each additional supply station (equivalently, the cost of creating each additional phase-switching state) in the world where the rover operates.
- ◇ $\hat{\lambda}$ specifies the cost limit for creating phase-switching states. For example, $\hat{\lambda} = 40$, given the above constraint $\lambda_{i=any\_state} = 10$, indicates that at most four phase-switching states can be created in the rover world.

[4]To simplify the presentation, it is assumed that the resource requirement is binary, which implies that an agent will not be interested in more than one unit of a particular resource, but all results presented in this and the next chapter can be generalized to non-binary resource requirement cases without much difficulty.

Given the inputs $\mathcal{M}$, α, $\mathcal{C}$, and $\mathcal{R}$, the objective of the S-RMP optimization problem is to maximize the total expected utility of the capacity-restricted agent by identifying a set of phase-switching states $S' = \{s^k\}$, which decompose the overall problem into a collection of phases, and, for each phase k, determining a resource configuration Δ^k and an executable policy π^k that would be adopted by the agent at the entry to that phase.

Specifically, from a constrained optimization perspective, the S-RMP optimization problem could be formulated as follows:

Objective:

$$\text{maximize the utility of the overall policy } \pi$$

subject to the following constraints:

i) The set of phase-switching states $S' = \{s^k\}$ should satisfy the phase-switching constraint $\mathcal{R}$.

ii) Within each phase k, resource configuration Δ^k should satisfy the agent capacity constraint $\mathcal{C}$.

iii) Within each phase k, policy π^k should be executable with respect to the resource configuration Δ^k.

iv) The overall policy π is composed of phase policies π^k, i.e., phase policy π^k is adopted by the agent when it encounters a phase-switching state $s^k \in S'$ in the midst of its execution.

Clearly, the S-RMP optimization problem involves three intertwined components: i) problem decomposition, ii) resource configuration, and, iii) policy formulation, whose relationships are illustrated in Figure 2.1. Problem decomposition (which

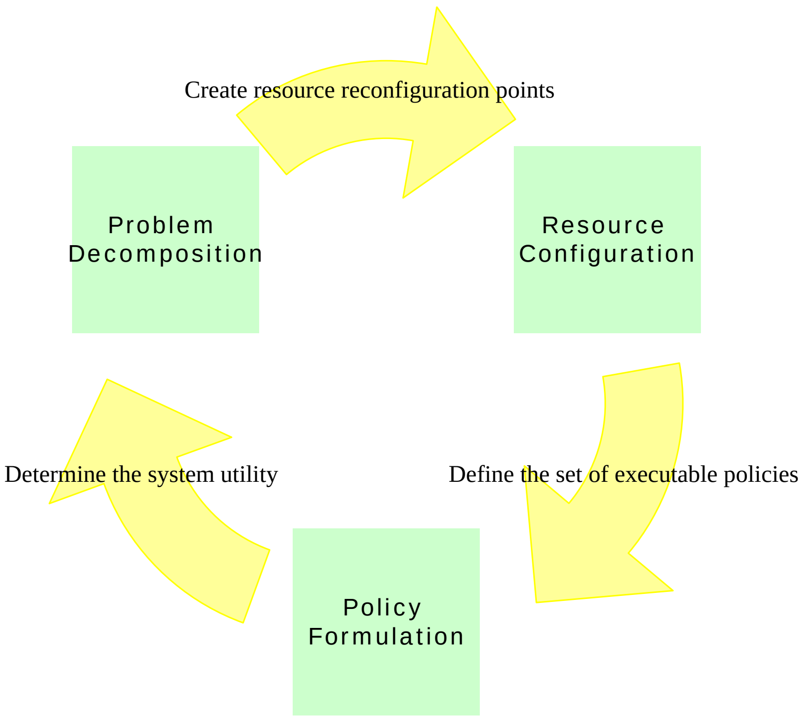

Create resource reconfiguration points
Problem Decomposition
Resource Configuration
Determine the system utility
Define the set of executable policies
Policy Formulation

Figure 2.1: The structural framework of the S-RMP optimization problem, involving problem decomposition, resource configuration, and policy formulation.

creates phase-switching states) paves the foundation for resource configuration and reconfiguration; resource configuration dictates what policies are executable in each phase; policy formulation determines transitions within and among phases as well as what goals can be achieved by the agent, which in turn determines the utility of problem decomposition and resource (re)configuration.

Each of these three component problems and some combinations of them have been investigated in a number of research fields (but none of the prior approaches is computationally tractable to the S-RMP optimization problem that tightly couples problem decomposition, resource configuration, and policy formulation). A comprehensive discussion that contrasts this work with prior work is postponed to Section 2.7 after our computationally efficient solution approach to the S-RMP optimization problems is presented.

2.2 Background: MDPs and Constrained MDPs

The main aim of this section is to introduce the classical Markov decision process (MDP) and its extended model — the constrained MDP, because the majority of the work throughout this dissertation is based on these foundations.

2.2.1 MDPs

In general, a classical discrete-time, fully-observable Markov decision process with a finite state space and a finite action space can be defined as a four-tuple $\langle S, A, P, R \rangle$ (Puterman, 1994), where:

- S is a finite state space.
- A is a finite action space. For a state $i \in S$, $A_i \subseteq A$ represents the set of actions that can be executed at the state i.
- $P = \{p_{i,a,j}\}$ represents state transition probability where $p_{i,a,j}$ is the probability that the agent reaches state j if it executes action a in state i.

  For any state i and action a, $\sum_j p_{i,a,j}$ must be no greater than one. $\sum_j p_{i,a,j} = 1$ means that the agent will always stay in the system when executing action a in state i, while $\sum_j p_{i,a,j} < 1$ means that there is some probability of the agent being out of the system (which can be equivalently interpreted as that the agent enters a *sink* state where the agent would stay there forever) when executing action a in state i.
- $R = \{r_{i,a}\}$ is the (bounded) reward function where $r_{i,a}$ is the reward that the agent will receive if it executes action a in state i.

The Markov decision process is an extension of the well-known Markov chain. The main property of a MDP is that it possesses the Markov property (Bellman, 1957):

if the current state of a MDP at time t is known, transitions to a new state at time $t+1$ only depend on the current state (and, of course, the action(s) chosen at it), but are independent of the previous history of states.

In a MDP, the decision-making agent chooses its actions based upon its observation of the current state of the world, with the motivation of maximizing its aggregate reward. A (stationary) policy for a MDP is defined as a mapping from states to actions: $\pi : i \rightarrow a$ where $i \in S$ and $a \in A_i$. The objective of the decision-making agent is to find an optimal policy that maximizes some predefined cumulative function of rewards. Let $\{i_0, i_1, ..., i_t, ...\}$ and $\{a_0, a_1, ..., a_t, ...\}$ represent a particular state and action sequence by following the policy π, and let $E[\,]$ denote the expectation function, then a typical cumulative reward function of a non-discounted MDP can be defined as:

$$U(\pi) = E[\sum_{t=0}^{\infty} r_{i_t,a_t}]$$

Similarly, the cumulative reward function of a discounted MDP with the discount factor γ can be defined as:[5]

$$U(\pi) = E[\sum_{t=0}^{\infty} (\gamma)^t \times r_{i_t,a_t}]$$

The mission-phasing techniques in this dissertation are illustrated using transient, non-discounted MDPs, although in general these techniques will also apply to discounted MDPs and other contracting MDPs. In a transient MDP (in which $\sum_j p_{i,a,j} < 1$ at some states), an agent will eventually leave the corresponding Markov chain, after running a policy for a finite number of steps (Kallenberg, 1983). In other words, given a finite state space, it is assumed that the agent visits any state only a finite number of times for any policy, which in turn means that the total expected reward function $U(\pi) = E[\sum_{t=0}^{\infty} r_{i_t,a_t}]$ is bounded.

[5]In this dissertation, $(a)^b$ represents an exponent, while a^b represents a superscript.

There have been a number of computationally-efficient polynomial-time algorithms for deriving an optimal policy to a given MDP. The following introduces some of these techniques.

Value Iteration

The value iteration algorithm starts with arbitrary initial state values $V^0(i)$ for every state $i \in S$, and then repeats the *Bellman backup* iteration process defined below (assuming no discounts).

$$V^{k+1}(i) \leftarrow \max_{a \in A_i}[\, r_{i,a} + \sum_{j \in S} p_{i,a,j} \times V^k(j)\,]$$

It has been shown that the sequence of state values V^k will eventually converge to optimal state values V^* in any contracting MDP (Kallenberg, 1983; Puterman, 1994; Sutton and Barto, 1998).[6]

At that convergence point,

$$V^*(i) = \max_{a \in A_i}[\, r_{i,a} + \sum_{j \in S} p_{i,a,j} \times V^*(j)\,]$$

and the choices of maximizing action for each state form an optimal policy $\pi^* = \{\pi(i)\}$, where

$$\pi(i) \leftarrow \operatorname*{argmax}_{a \in A_i}[\, r_{i,a} + \sum_{j \in S} p_{i,a,j} \times V^*(j)\,]$$

Policy Iteration

The policy iteration algorithm is another common policy formulation algorithm. It alternates the following two steps, beginning with a randomly generated initial policy π^0:

▷ *Policy Evaluation*: for the current policy π^k, calculate state values V^k. Since the policy is fixed and known, the Markov decision process is reduced to a Markov

[6]The transient MDP of interest in this work is a subclass of contracting MDPs.

chain. A Markov chain can be solved in $O(|S|^3)$, where $|S|$ represents the size of the MDP state space, by formulating it into linear equations and adopting standard linear algebra methods.

▷ *Policy Improvement*: calculate an updated policy π^{k+1} using the following one-step look-ahead:

$$\pi^{k+1}(i) \leftarrow \operatorname*{argmax}_{a \in A_i} [\, r_{i,a} + \sum_{j \in S} p_{i,a,j} \times V^k(j) \,]$$

Because each policy is guaranteed to be a strict improvement over the previous one (unless the policy has already been optimal) and a finite MDP (with a finite state and action space) has only a finite number of different policies, the policy iteration procedure will eventually converge to an optimal policy and optimal state values after a finite number of iterations (Puterman, 1994).

Linear Programming

The value iteration and policy iteration algorithms are widely used in solving classical unconstrained MDPs. However, it is surprisingly hard to extend these algorithms to also work on a constrained problem without considerably increasing the size of the state space and/or the action space of the unconstrained MDP. For that reason, a number of researchers have proposed and utilized an alternative solution approach, which is based upon mathematical programming (Altman, 1998; Feinberg, 2000; Dolgov and Durfee, 2003). The procedure of formulating an unconstrained MDP into a linear program (whose solution yields an optimal policy maximizing the total expected reward) is described below, because our work extends this approach.

Let $x_{i,a}$, which is often called *occupation measure* or *visitation frequency* (Dolgov and Durfee, 2006), denote the expected number of times action a is executed in state i, then the function $\sum_i \sum_a x_{i,a} \times r_{i,a}$ can be used to represent the total expected reward, and the problem of finding an optimal policy to the MDP is equivalent to solving the following linear program:

$$\max \sum_i \sum_a x_{i,a} \times r_{i,a} \tag{2.1}$$

subject to:

$$\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a} \qquad : \forall j$$

$$x_{i,a} \geq 0 \qquad : \forall i, \forall a$$

where α_j is the probability that the agent is initially in state j, and the constraint (named the *probability conservation* constraint) $\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a}$ guarantees that the expected number of times state j is visited must equal the initial probability distribution at state j plus the expected number of times state j is entered via all possible transitions.

When the linear program Eq. 2.1 is solved, it is trivial to derive an optimal policy that specifies the action(s) to take in a given state. Specifically, assigning a probability of executing action a in state i as $\pi_{i,a} = \frac{x_{i,a}}{\sum_a x_{i,a}}$ will maximize the total expected reward. If any probability $\pi_{i,a}$ has a value other than zero or one, the optimal policy is randomized; otherwise it is deterministic.

2.2.2 Constrained MDPs

Formulating unconstrained MDPs as linear programs makes it easier to take into account additional constraints, including the agent capacity constraints and resource constraints. Several of such constrained optimization problems have been investigated by Dolgov and Durfee (2003, 2006). In order to familiarize readers with some background knowledge on constrained Markov decision processes and the techniques used to solve them, a brief introduction to their work is given below.

Using the same notations as in the S-RMP optimization problem defined in Section 2.1, a constrained MDP that models agent capacity limitations can be represented as $\langle \mathcal{M}, \alpha, \mathcal{C} \rangle$, where:

- ☐ The classical MDP $\mathcal{M}$ is represented as $\langle S, A, P, R \rangle$.
- ☐ $\alpha = \{\alpha_i\}$ indicates the initial probability distribution.
- ☐ The agent capacity constraint $\mathcal{C}$ is represented as $\langle O, C, U, \Gamma, \hat{\Gamma} \rangle$. As a reminder, $U = \{u_{o,a,i}\}$ indicates whether the agent needs resource o to execute action a in state i, $\Gamma = \{\tau_{o,c}\}$ defines how much capacity c would be consumed to hold one unit of resource o, and $\hat{\Gamma} = \{\hat{\tau}_c\}$ specifies the limit of the agent capacity c.

The linear programming formulation (Eq. 2.1) paves the way for incorporating agent capacity constraints. Namely, the capacity limitations can be modeled by imposing the following mathematical constraints (shown in Eq. 2.2) on occupation measure $x_{i,a}$ defined and used in the linear program Eq. 2.1.

$$\sum_{o} \tau_{o,c} \times \Theta\Big(\sum_{i} \sum_{a} u_{o,a,i} \times x_{i,a} \Big) \leq \hat{\tau}_c \qquad : \forall c \qquad (2.2)$$

where $\Theta(z)$ is a step function, defined as

$$\Theta(z) = \begin{cases} 1 & z > 0 \\ 0 & otherwise \end{cases}$$

The constraint indicates that, given the resource requirement parameter $u_{o,a,i} = 1$, the agent will have to employ $\tau_{o,c}$ amount of its capacity c to hold resource o if it decides to schedule action a in state i in its policy.

Note that the $\Theta(z)$ function is a nonlinear function. In general, directly solving nonlinear constrained optimization problems is difficult. Fortunately, there is a simple way that can transform the nonlinear constraint Eq. 2.2 into linear constraints through introducing some integer variables (Dolgov and Durfee, 2003). Thereafter, state-of-art integer linear programming techniques (which are typically more efficient than nonlinear programming techniques) can be adopted to solve constrained MDPs.

The reformulated constraints are depicted below.

$$\begin{aligned}
&\frac{\sum_i \sum_a u_{o,a,i} \times x_{i,a}}{X} \leq \Delta_o && : \forall o \\
&\sum_o \tau_{o,c} \times \Delta_o \leq \hat{\tau}_c && : \forall c \\
&\Delta_o \in \{0,1\} && : \forall o
\end{aligned}$$

where Δ_o, a binary integer in the set $\{0,1\}$, is introduced to indicate whether the agent uses its limited capacity to hold resource o. X is a constant that is no less than $\sup \sum_i \sum_a x_{i,a}$, which is applied to guarantee that $\frac{\sum_i \sum_a u_{o,a,i} \times x_{i,a}}{X}$ never exceeds one (because $\sum_i \sum_a u_{o,a,i} \times x_{i,a} \leq \sum_i \sum_a x_{i,a} \leq \sup \sum_i \sum_a x_{i,a} \leq X$). One way to compute X is to solve an unconstrained MDP:

$$X = \max \sum_i \sum_a x_{i,a} \tag{2.3}$$

subject to:

$$\begin{aligned}
&\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a} && : \forall j \\
&x_{i,a} \geq 0 && : \forall i, \forall a
\end{aligned}$$

To summarize, the constrained MDP that models the agent's capacity limitations can be formulated into a mathematical program Eq. 2.4 (i.e., by putting Eq. 2.1 and the above integer linear constraints together), whose solution will yield an optimal capacity usage configuration and an optimal executable policy.

$$\max \sum_i \sum_a x_{i,a} \times r_{i,a} \tag{2.4}$$

subject to:

$$\begin{aligned}
&\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a} && : \forall j \\
&\frac{\sum_i \sum_a u_{o,a,i} \times x_{i,a}}{X} \leq \Delta_o && : \forall o \\
&\sum_o \tau_{o,c} \times \Delta_o \leq \hat{\tau}_c && : \forall c \\
&x_{i,a} \geq 0 && : \forall i, \forall a \\
&\Delta_o \in \{0,1\} && : \forall o
\end{aligned}$$

In Eq. 2.4, $p_{i,a,j}$, $r_{i,a}$, α_j, $u_{o,a,i}$, $\tau_{o,c}$, $\hat{\tau}_c$, and X are constants, while $x_{i,a}$ are continuous variables and Δ_o are binary integer variables, which indicates that Eq. 2.4 is a mixed integer linear program (MILP).

Mixed integer linear programming is the discrete version of linear programming with an additional requirement that partial variables must be integers. MILPs can be solved by a variety of highly optimized algorithms and tools (Cook et al., 1998; Wolsey, 1998). Recently, there has been substantial progress on using MILPs in automated planning (Earl and D'Andrea, 2005; Kautz and Walser, 2000; van Beek and Chen, 1999; Vossen et al., 1999), and the automated resource-driven mission-phasing techniques presented in this dissertation are based upon the MILP as well.

2.2.3 An Example

To help readers better understand the constrained MDP approach, this subsection illustrates it through an example problem. Our automated resource-driven mission-phasing approach, which extends the constrained MDP approach, will be illustrated using the same example problem as well.

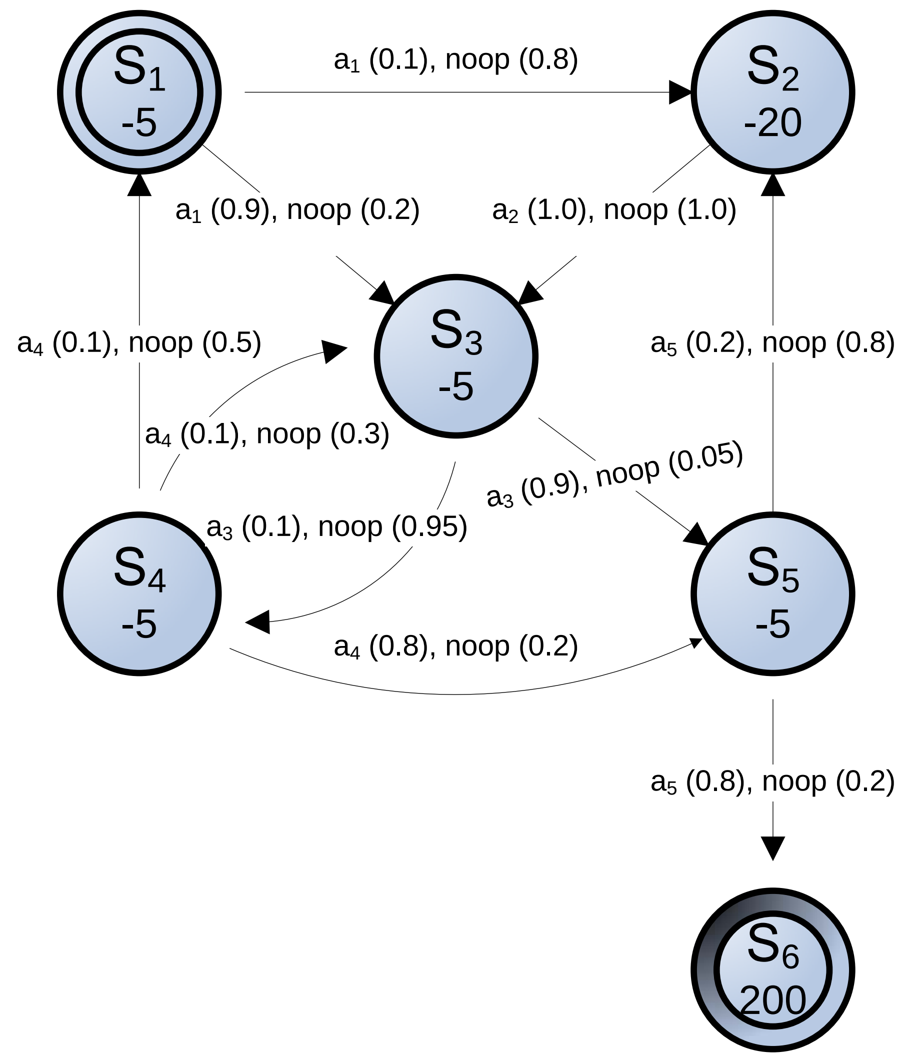


Figure 2.2: A simple single-agent example.

The example problem is shown in Figure 2.2. The problem has six states $\{S_1, S_2, S_3, S_4, S_5, S_6\}$ and six possible actions $\{a_0{=}noop, a_1, a_2, a_3, a_4, a_5\}$, where the agent starts at S_1, and a_0 is a *noop* that represents the fact that the agent has the freedom of not executing any action. There are five types of resources $\{o_1, o_2, o_3, o_4, o_5\}$ in the problem. Executing action $a_i \in \{a_1, a_2, a_3, a_4, a_5\}$ requires the agent having resource o_i, while the action *noop* does not require any resource by definition. That is, $u_{1,1,1} = 1$, $u_{2,2,2} = 1$, $u_{3,3,3} = 1$, $u_{4,4,4} = 1$, $u_{5,5,5} = 1$, and $u_{o,a,i} = 0$ in all other cases.

If the number of resources the agent can carry is unlimited, this example problem is, in fact, a classical unconstrained MDP. Using a policy formulation algorithm (e.g., value iteration or policy iteration), we could easily compute the optimal policy, which is $[S_1 \rightarrow a_1, S_2 \rightarrow noop/a_2, S_3 \rightarrow a_3, S_4 \rightarrow a_4, S_5 \rightarrow a_5, S_6 \rightarrow noop]$, and the total expected reward is 174.65.

Suppose instead that the capacity of the agent is highly restricted, such that now the agent can carry only one resource (but it can choose which one to carry), which means it can only execute a policy with at most one action that is not a *noop*. We can solve this problem by parameterizing Eq. 2.4 with the resource requirement parameters described above, the MDP-related parameters depicted in Figure 2.2, and the constant $X = 70.24$ computed by Eq. 2.3. The resulting MILP is shown below:

$$\begin{aligned}\max\ & -5 \times x_{1,0} - 5 \times x_{1,1} - 20 \times x_{2,0} - 20 \times x_{2,2} - 5 \times x_{3,0} - 5 \times x_{3,3} \\ & -5 \times x_{4,0} - 5 \times x_{4,4} - 5 \times x_{5,0} - 5 \times x_{5,5} + 200 \times x_{6,0}\end{aligned}$$

subject to:

$$\begin{aligned}
x_{1,0} + x_{1,1} &= 0.5 \times x_{4,0} + 0.1 \times x_{4,4} + 1.0 \\
x_{2,0} + x_{2,2} &= 0.8 \times x_{1,0} + 0.1 \times x_{1,1} + 0.8 \times x_{5,0} + 0.2 \times x_{5,5} \\
x_{3,0} + x_{3,3} &= 0.2 \times x_{1,0} + 0.9 \times x_{1,1} + x_{2,0} + x_{2,2} + 0.3 \times x_{4,0} + 0.1 \times x_{4,4} \\
x_{4,0} + x_{4,4} &= 0.95 \times x_{3,0} + 0.1 \times x_{3,3} \\
x_{5,0} + x_{5,5} &= 0.05 \times x_{3,0} + 0.9 \times x_{3,3} + 0.2 \times x_{4,0} + 0.8 \times x_{4,4} \\
x_{6,0} &= 0.2 \times x_{5,0} + 0.8 \times x_{5,5}
\end{aligned}$$

(additional constraints on the next page)

$$x_{1,1} \le 70.24 \times \Delta_1$$

$$x_{2,2} \le 70.24 \times \Delta_2$$

$$x_{3,3} \le 70.24 \times \Delta_3$$

$$x_{4,4} \le 70.24 \times \Delta_4$$

$$x_{5,5} \le 70.24 \times \Delta_5$$

$$\sum_{o=1}^{5} \Delta_o \le 1$$

Using a MILP solver, such as *cplex* (www.ilog.com), we can easily derive an optimal solution to the above MILP:

$$\begin{aligned} &[(x_{10}, x_{11}), (x_{20}, x_{22}), (x_{30}, x_{33}), (x_{40}, x_{44}), (x_{50}, x_{55}), x_{60}] \\ =&[(3.47, 0), (3.03, 0), (5.21, 0), (4.95, 0), (0, 1.25), 1] \\ &[\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5] = [0, 0, 0, 0, 1] \end{aligned}$$

That is, the optimal policy is $[S_1 \rightarrow noop, S_2 \rightarrow noop, S_3 \rightarrow noop, S_4 \rightarrow noop, S_5 \rightarrow a_5, S_6 \rightarrow noop]$, and the corresponding total expected reward is reduced to 65.02 (from 174.65 in the unconstrained case) due to the limitation on the agent capacity. This is the optimal policy for the constrained agent that uses a single policy throughout its entire mission. We will use this example as we go along to illustrate the degree to which our automated mission-phasing techniques can improve that expected reward.

2.3 Computational Complexity Analysis

Typically, finding an exact solution to a S-RMP optimization problem is computationally challenging because not only should it decide on an optimal way to decompose the problem into phases, but it should determine how to reconfigure resources (and capacity usage) at the entry to each phase and determine what the optimal executable policy is within each phase. These three component problems — problem decomposition, resource configuration, and policy formulation — are strongly intertwined: the optimality of the solution to one component problem is with respect to the solutions to the other two component problems.

The purposes of this section are to theoretically analyze computational complexity of the S-RMP optimization problem and to illustrate why standard approaches are not computationally tractable in solving it. The improvement of the techniques presented in this chapter over those approaches will be empirically illustrated and discussed in Section 2.6.

Let us start by proving the following theorem:

Theorem II.1. *S-RMP optimization is NP-complete.*

Proof: The proof of S-RMP optimization being NP-hard is trivial, because one of its special cases, which includes only one phase (i.e., the agent can only configure its resources at the beginning of mission execution), has been proven to be NP-hard through a reduction from the well-known KNAPSACK problem (Dolgov and Durfee, 2003; Dolgov, 2006).

The presence in NP can be proven in the following way. For a MDP with n states, it is clear that there can be at most n phases (i.e., n phases in the extreme situation where every state is a phase-switching state). By featuring phase id (assuming each phase has a unique id) in the state representation, a generalized MDP with at most n^2 states can be constructed in polynomial time and the phase policies can be combined into an overall policy to this generalized MDP in polynomial time too. Given the generalized MDP and its policy, the problem is reduced to solving a Markov chain. Since a Markov chain can be verified in polynomial time, S-RMP optimization is in

NP.

Given its presence in both NP and NP-hard, S-RMP optimization is proven to be NP-complete. $\square$

To further illustrate computational complexity of the S-RMP optimization problem, the rest of this section discusses and explains why two standard approaches — the brute-force search algorithm and the MDP-based algorithm — are, in general, computationally challenging in solving the S-RMP optimization problem.

The brute-force search approach. The brute-force search algorithm enumerates all possible problem decomposition schemes, and, for each decomposition scheme, enumerates all possible ways to configure and reconfigure resources, and, finally, for each possible way of problem decomposition and resource (re)configuration, derives optimal phase policies that are executable with respect to the configured resources.[7]

Let us consider an example problem: there are o different resources, the agent can hold c out of o resources (given capacity cost $\tau_{o,c} = 1$ and capacity limit $\hat{\tau}_c = c$), the size of the MDP state space is s, and there can be up to k phases in the system (given that the cost for making one state i into a phase-switching state is $\lambda_i = 1$ and that the amount of cost that can be used to create phase-switching states is $\hat{\lambda} = k$).

In the example, the brute-force search approach would need to enumerate C_{k-1}^{s-1} different ways of problem decomposition, where C_y^x represents the total number of ways of taking y out of x things at a time, and C_{k-1}^{s-1} instead of C_k^s in the formulation is because it is assumed that the capacity-limited agent can always configure its resources in its initial state (i.e., the first phase always starts at

[7]Enumerating all possible policies is unnecessary, because there exist a number of efficient policy formulation algorithms.

the initial state). Given a decomposition, the number of possible ways to sequentially configure resources is a product of the numbers of possible ways of configuring resources in each phase. That is, for each problem decomposition scheme, there are $(C_c^o)^k$ possible ways of configuring and reconfiguring resources. As a result, the brute-force search approach needs to solve $C_{k-1}^{s-1} \times (C_c^o)^k$ MDPs, each of which is a non-trivial stochastic planning problem with $s \times k$ states.

Even for moderately complex problems, the brute-force search approach is computationally intractable. For example, if $s = 20$, $o = 9$, $k = 3$, and $c = 3$, the approach would need to enumerate and solve $101,352,384$ policy formulation problems in order to find an optimal solution to the S-RMP optimization problem. As another example, if $s = 40$, $o = 9$, $k = 6$, and $c = 3$, the approach would need to enumerate and solve 2.02×10^{17} policy formulation problems.

The MDP-based approach. Unlike the brute-force search approach that considers each S-RMP component problem in isolation, the MDP-based approach incorporates resources into the state representation, and models resource reconfiguration activities as explicit actions. Compared to the brute-force search approach, it avoids the examination of an exponential (exponential in the product of the number of resources and the number of phases) number of MDPs with a cost of an exponential (exponential only in the number of resources) increase in the size of the state and action space.

Intuitively, the MDP-based algorithm is faster than the brute-force search algorithm since the resource reconfiguration process is now embedded in the policy formulation process of the generalized MDP, and the policy formulation problem can be solved using efficient dynamic programming methods (instead of enumerating all possible combinations of resource configuration actions).

However, the MDP-based approach is still computationally challenging, not only because it now has an exponentially larger state space (i.e., the size of the state space increases from s to $s \times C_c^o$) but also because it still needs to cope with phase-switching constraints that restrict the states where resource-reconfiguration actions can be taken. There are typically two ways to take into account phase-switching constraints. We can either construct and solve a single constrained MDP that can be formulated into a MILP with $s-1$ binary variables (i.e., using the MILP formulation Eq. 2.4), or enumerate all C_{k-1}^{s-1} possible decomposition schemes, and, for each, solve an unconstrained MDP.

That is, for the previous example with $s = 40$, $o = 9$, $k = 6$, and $c = 3$, the decision-making agent can choose either to solve a single constrained MDP with $s \times C_c^o = 3,360$ states and s–$1 = 39$ binary variables, or to solve $C_{k-1}^{s-1} = 575,757$ unconstrained MDPs, each of which has the same size of the state space as the above constrained MDP.

As illustrated in the above example problem, both the brute-force search approach and the MDP-based approach are not computationally efficient for the S-RMP optimization problem. This is primarily because these two approaches do not take into account problem structure that can be exploited to speed up the process of finding an exact solution. The brute-force search approach deals with S-RMP component problems in isolation, while the MDP-based approach combines the resource-configuration and policy-formulation components in a naive way, which results in an exponentially larger policy formulation problem. Neither of these two approaches can effectively exploit interactions and influences among the S-RMP component problems.

Instead, as we will see, the solution algorithms designed in this work can take advantage of problem structure by formulating problem decomposition, resource configuration, and policy formulation problems into a compact mathematical program and solving these component problems simultaneously and effectively using a highly optimized tool. As will be shown in Section 2.6, the algorithms presented in this chapter can find exact solutions to the problems similar to the above examples within a reasonable time (i.e., a few seconds).

2.4 Exploiting Fixed Phase-Switching States

To this point, we have formally defined the S-RMP optimization problem, recapped its background, and theoretically analyzed its computational complexity. In this and the next sections, we will present and illustrate our computationally efficient automated mission-phasing algorithms for solving S-RMP optimization problems.

We begin our examination of automated mission-phasing techniques by first examining a simplified variation of the S-RMP optimization problem, in which phase-switching states are defined and known *a priori*. This problem is clearly a subclass of the standard S-RMP optimization problem with stricter restrictions: i) if state i is in a predefined phase-switching set, the phase-switching cost $\lambda_i = 0$ since that phase-switching state already exists, ii) $\lambda_i > 0$ otherwise, and iii) the cost limit $\hat{\lambda} = 0$.

This variation fits many problems where the opportunities to reconfigure resources and switch policies are dictated by the state of the world rather than being a choice of the agent. In the case of a Mars rover, the locations on Mars where it can change its instruments may be very limited, and well known to it. Exploiting the fact that phase-switching states are fixed, we can work out a particular, efficient algorithm (while a more general but maybe slower algorithm will be presented in Section 2.5).

Decomposition techniques for planning in stochastic domains are widely used for large environments with many states (a detailed discussion of problem decomposition techniques will be given in Section 2.7). In those approaches, states are partitioned into small regions, a policy is computed for each region, and then these local policies are pieced together to obtain an overall policy (Parr, 1998; Precup and Sutton, 1998; Lane and Kaelbling, 2001). Our automated mission-phasing techniques are analogous to those decomposition techniques — partitioning a mission into multiple phases leads to smaller state and action spaces in each phase — though our motivation for mission-phasing is the constraints on policies agents can execute rather than the reduction of computational cost during policy formulation. Nonetheless, we can exploit these ideas.

Our algorithm for solving S-RMP optimization problems with predefined phase-switching states is based upon *abstract MDPs*. An abstract MDP is composed of abstract states, each of which corresponds to a mission phase. The "action" for an abstract state is the policy used in its corresponding phase. It is here assumed that none of constraints is associated with more than one phase. The discussion of more general constraints is postponed to the next section.

Since it is assumed that agent constraints in one phase cannot be affected by policy choices in another phase, the abstract MDP is an unconstrained MDP (at the abstract level) even though internally each phase is still a constrained MDP. The algorithm thus uses a policy iteration approach at the abstract level together with an embedded MILP solver within phases. The embedded MILP solver finds possible executable policies and their expected rewards for each of the phases, while different policies may have different probabilities of reaching the various phase-switching states at the "edges" of the phase. The outer policy iteration algorithm at the abstract level iteratively searches for the combination of phase policies that maximizes the reward across the whole mission.

The detailed procedure of the abstract MDP solver is illustrated below:

1. ***Partitioning the mission into phases.***

   When phase-switching states are given, partitioning a mission into multiple phases is straightforward. Start from a phase-switching state, and then keep expanding through all connected transitions until encountering other phase-switching states. The resulting state space is the phase state space corresponding to that phase-switching state.

2. ***Policy iteration.***

   The following policy iteration algorithm is adopted after the mission is partitioned.

   (a) Solve the corresponding unconstrained MDP and compute state values $V(s)$ for each phase-switching state s. $V(s)$ are used as initial values of phase-switching states since they are likely to provide good estimates.

   (b) In the abstract MDP, each phase is treated as an abstract state and each policy for a phase is treated as an abstract action for that phase's abstract state. The policy iteration algorithm alternates between the following two steps:

      Policy improvement: Rather than enumerating all possible policies (abstract actions) for a phase (abstract state), the algorithm uses a constrained MDP solver (that was shown in Eq. 2.4) to calculate the optimal policy in the phase, given the current state values of the (outgoing) neighboring phase-switching states.

Policy evaluation: Given abstract actions, calculate $V(s)$ for each phase-switching state s. For small state spaces, standard linear algebra methods are often the best solutions for policy evaluation. For large state spaces, a simplified value iteration algorithm might be preferred (simplified because the policy in each phase is fixed).

Unlike much "best-response" hill-climbing work, the above abstract MDP has fixed state transition functions and fixed reward functions in both the abstract level and the phase level because the agent enters a phase always at the same phase-switching state, which guarantees the above policy iteration algorithm will return an optimal solution.

Theorem II.2. *The abstract MDP policy iteration procedure will converge to an optimal solution.*

Proof: In each iteration, the new abstract policy is a strict improvement over the previous one. Since the total expected reward of the abstract MDP is bounded (because the total expected reward of the corresponding unconstrained MDP is bounded), the iteration procedure will eventually converge.

At the convergence point, both the phase MDPs and the abstract MDP satisfy the Bellman optimality equation (because of the nature of the linear programming solver and the policy iteration algorithm), indicating that the derived policy is an optimal policy. □

Running Example

We now return to our running example introduced in Section 2.2.3 to illustrate how the total expected reward can be improved if the agent can reconfigure its resources at some states. Let us say that the agent knows it is able to reconfigure

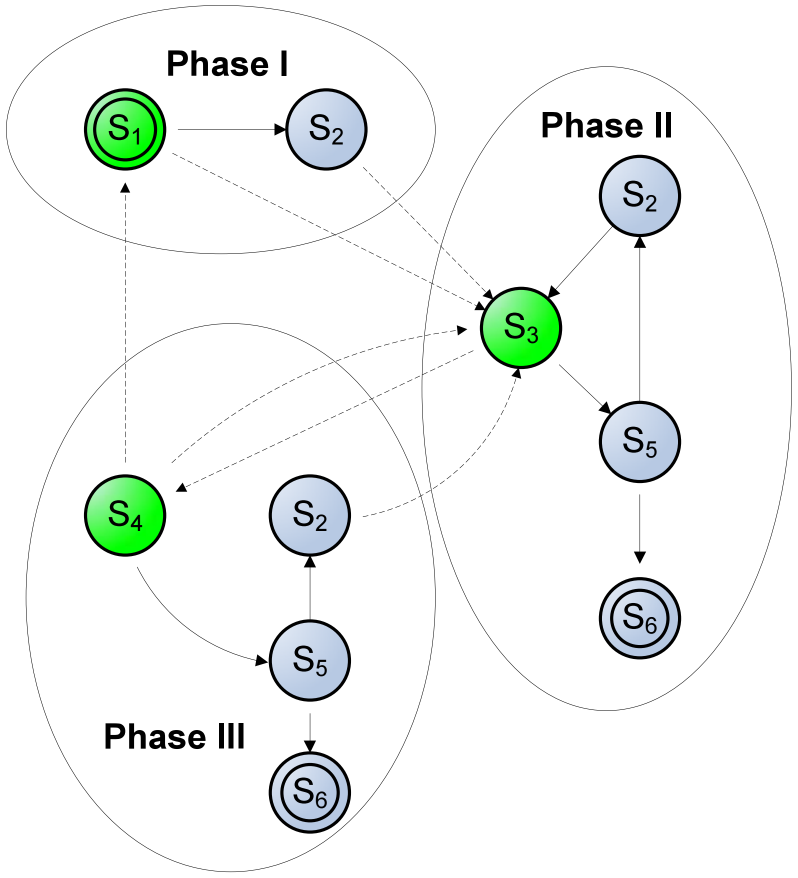


Figure 2.3: An abstract MDP with three phases.

resources and switch policies at states S_1, S_3 and S_4. These three phase-switching states decompose the example problem into three phases. The corresponding abstract MDP is constructed and shown in Figure 2.3, which is composed of three abstract states.

Using the above abstract MDP policy-iteration algorithm and assuming the same parameters (especially that an executable policy cannot have more than one action that is not a *noop*), we can see that the state values of the phase-switching states eventually converge to

$$V(S_1) = 113.65 \qquad V(S_3) = 120.65 \qquad V(S_4) = 123.05$$

The optimal policy in phase I is $[S_1 \rightarrow a_1, S_2 \rightarrow noop]$ (with resource o_1), the optimal policy in phase II is $[S_2 \rightarrow noop, S_3 \rightarrow noop, S_5 \rightarrow a_5, S_6 \rightarrow noop]$ (with resource o_5), and the optimal policy in phase III is $[S_2 \rightarrow noop, S_4 \rightarrow noop, S_5 \rightarrow a_5, S_6 \rightarrow noop]$ (with resource o_5 too). The total expected reward is now 113.65, which is 74.8% higher than not exploiting those phase-switching states.

Thanks to the policy iteration procedure, the abstract MDP solver generally converges quickly. However, it should be noted that two limitations are inherent in the abstract MDP solver. One of the limitations is that the abstract MDP solver requires that phase-switching states are known *a priori*, which restricts its applicability (although we can combine it with some phase-switching-state heuristic search techniques). The other limitation is due to the possible existence of constraints running across multiple phases. The abstract MDP is an unconstrained MDP and so the policy iteration algorithm is efficient and well suited. In other words, the abstract MDP solver cannot cope with constraints associated with multiple abstract states, such as restrictions on the expected number of visits to a particular state that belongs to multiple phases. In contrast, the general S-RMP solution algorithms that will be presented in the next section do not have such limitations.

2.5 Determining Optimal Phase-Switching States

2.5.1 Solution Algorithm

In a general S-RMP optimization problem, phase-switching states are not defined and known *a priori*. Instead, given phase-switching cost $\{\lambda_i\}$ (where λ_i denotes the cost for making state i into one that is conducive for an agent to reconfigure resources) and a cost limit $\hat{\lambda}$, the objective of the agent is to find an optimal phase-switching set $S' \subseteq S$ subject to $\sum_{i \in S'} \lambda_i \leq \hat{\lambda}$, along with optimal resource configurations and optimal executable policies within each phase, to maximize its expected cumulative reward.

As stated, the abstract MDP solver presented in Section 2.4 cannot be directly used for the general S-RMP optimization problem. In this section, we construct a mixed integer linear program, the solution to which yields the optimal set of phase-switching states maximizing the total expected reward, as well as optimal resource configurations and executable policies within each phase. We make a simplifying assumption that a START state (which has a positive initial probability distribution α_j) is always a phase-switching state. This assumption makes the presentation clearer and the representation more concise, as well as sidestepping the question of what the "default" agent policy might be (since that is what it would use if it could not configure resources in its START state).

Let $x^k_{i,a}$ be the expected number of times action a is executed in state i within phase k. Clearly, if state i is not reachable in phase k, then $x^k_{i,a} = 0$. Let $\alpha^k_j = \sum_a x^k_{j,a} - \sum_i \sum_a p_{i,a,j} \times x^k_{i,a}$ where $p_{i,a,j}$ is the state transition probability, then α^k_j provides a way to characterize transitions among phases. If state j is not a phase-switching state, then $\alpha^k_j = 0$ for any k, since within any phase the expected number of times of visiting state j ($\sum_a x^k_{j,a}$) must equal the expected number of times of entering state j through all possible transitions ($\sum_i \sum_a p_{i,a,j} \times x^k_{i,a}$). If state j is a phase-switching state, $\sum_k \alpha^k_j = \alpha_j$. Recall that α_j is the initial probability distribution for state j. $\sum_k \alpha^k_j = \alpha_j$ guarantees that the total expected number of times of visiting state j must equal the initial probability distribution for state j plus the total expected number of times of entering state j through all possible transitions.

Now, we can formulate the S-RMP optimization problem into a mixed integer linear program,[8] which is shown in Eq. 2.5. The objective function $\sum_i \sum_a \sum_k x^k_{i,a} \times r_{i,a}$ in the MILP represents the total expected reward accumulated across all phases,

[8]Given that S-RMP optimization is NP-complete, MILP (also NP-complete) is a reasonable formulation and it allows us to exploit a variety of existing highly optimized MILP algorithms and tools.

where $r_{i,a}$ is the MDP reward function.

$$\max \sum_k \sum_i \sum_a x^k_{i,a} \times r_{i,a} \tag{2.5}$$

subject to:

probability conservation constraints:

$$\sum_a x^k_{j,a} = \alpha^k_j + \sum_i \sum_a p_{i,a,j} \times x^k_{i,a} \qquad : \forall k, \forall j$$

$$\sum_k \alpha^k_j = \alpha_j \qquad : \forall j$$

$$x^k_{i,a} \geq 0 \qquad : \forall k, \forall i, \forall a$$

capacity constraints:

$$\frac{\sum_i \sum_a u_{o,a,i} \times x^k_{i,a}}{X} \leq \Delta^k_o \qquad : \forall o, \forall k$$

$$\sum_o \tau_{o,c} \times \Delta^k_o \leq \hat{\tau}_c \qquad : \forall c, \forall k$$

$$\Delta^k_o \in \{0, 1\} \qquad : \forall o, \forall k$$

phase-switching constraints:

$$\frac{\alpha^k_j}{X} \leq \Lambda_j \qquad : \forall k, \forall j$$

$$\sum_j \lambda_j \times \Lambda_j \leq \hat{\lambda}$$

$$\Lambda_j \in \{0, 1\} \qquad : \forall j$$

- As stated above, the constraint $\sum_a x^k_{j,a} = \alpha^k_j + \sum_i \sum_a p_{i,a,j} \times x^k_{i,a}$ models the conservation of probability within each phase.

- The constraint $\sum_k \alpha^k_j = \alpha_j$ indicates the probability conservation constraint across phases, i.e., $\sum_k \alpha^k_j = \sum_k (\sum_a x^k_{j,a} - \sum_i \sum_a p_{i,a,j} \times x^k_{i,a}) = \sum_a x_{j,a} - \sum_i \sum_a p_{i,a,j} \times x_{i,a} = \alpha_j$, where $x_{i,a} = \sum_k x^k_{i,a}$ is the total expected number of times action a is executed in state i.

- The capacity constraints $\frac{\sum_i \sum_a u_{o,a,i} \times x_{i,a}^k}{X} \leq \Delta_o^k$ and $\sum_o \tau_{o,c} \times \Delta_o^k \leq \hat{\tau}_c$ are a multi-phase version of the capacity constraints discussed in Eq. 2.4, where $X = \max \sum_i \sum_a x_{i,a}$ can be computed by using Eq. 2.3.

- Λ_j in the constraint $\frac{\alpha_j^k}{X} \leq \Lambda_j$ is a binary variable, where $\Lambda_j = 1$ when state j is a phase-switching state, and $\Lambda_j = 0$ otherwise. We can prove $X \geq \sup \alpha_j^k$ as follows:

$$\begin{aligned}\sup \alpha_j^k &= \sup(\sum_a x_{ja}^k - \sum_i \sum_a p_{ij}^a x_{ia}^k)\\ &\leq \sup \sum_a x_{ja}^k\\ &\leq \sum_i \sum_a \sum_k x_{ia}^k\\ &= X\end{aligned}$$

  Therefore, this constraint and the constraint $\Lambda_j \in \{0, 1\}$ guarantee that $\exists k$: $\frac{\alpha_j^k}{X} > 0 \Rightarrow \Lambda_j = 1$, which implies that a state must be a phase-switching state if there is some "transition leakage" at that state in any phase.

- The constraint $\sum_j \lambda_j \times \Lambda_j \leq \hat{\lambda}$ says that the cost of creating phase-switching states must be no greater than the cost limit λ.

- Other constraints denote the ranges of variables. Note that there are no range restrictions for α_j^k.

By definition, Λ_j and Δ_o^k in the solution to Eq. 2.5 indicate phase-switching states and resource configuration (within each phase), respectively. We here show how to derive an optimal overall policy by examining $x_{i,a}^k$.

The detailed procedure of deriving the overall policy is described below.

1. ***Computing the optimal phase policy in each phase.***

   This is the same as before — at state i, action a is executed with probability $\pi^k_{i,a} = \frac{x^k_{i,a}}{\sum_a x^k_{i,a}}$. In the following discussion, we use $\pi^k = \{\pi^k_{i,a}\}$ to denote the phase policy in phase k.

2. ***Determining the phase policy to adopt at a phase-switching state.***

   This is also trivial. The agent should choose phase policy π^k with probability $\Pi^k_i = \frac{x^k_i}{\sum_k x^k_i}$ at phase-switching state i for maximizing its total expected reward, where $x^k_i = \sum_a x^k_{i,a}$.

Improving Computational Efficiency

The solution algorithm presented in Eq. 2.5 assumes that any state can be made into a phase-switching state (as long as its cost does not exceed the phase-switching cost limit), but it is very easy to make the algorithm also work in situations where some states are restricted from being phase-switching states. We just need to filter these states in the MILP formulation by, for each filtered state j, not modeling it with binary variable Λ_j and always setting its associated α^k_j to be zero.

In fact, the restriction on feasible phase-switching states often reduces computational cost of our MILP-based algorithm since it reduces the number of integer variables in the MILP formulation. This suggests that one potential way to improve computational efficiency of the algorithm is to examine the cost λ_i and discard some "bad" states with unreasonably high phase-switching costs before formulating the MILP. In such a way, only $|S| - |Bad|$ states need to be considered as candidate phase-switching states in the MILP formulation.

Another way, which can also often considerably reduce computational cost, is to trim down the set of k in the MILP formulation. That is to say, rather than ranging k from 1 to $|S|$ ($|S|$ is the size of the MDP state space), we can do a preliminary computation to compute the upper bound (denoted as K) of the number of phases among all possible ways of setting up phase-switching states, and then restrict $k \in \{1, .., K\}$ (instead of $k \in \{1, .., |S|\}$). The way of computing K is presented below.

$$K = \max \sum_i \Lambda_i \tag{2.6}$$

subject to:

$$\sum_j \lambda_j \times \Lambda_j \leq \hat{\lambda}$$

$$\Lambda_i \in \{0, 1\}$$

In tightly constrained S-RMP optimization problems, K is often much smaller than $|S|$, which greatly reduces the number of variables in the MILP formulation and thus can often considerably improve the efficiency.

Running Example

This section concludes by illustrating the solution algorithm on our running example illustrated in Figure 2.2. Recall that, as was shown in Section 2.4, when the agent is allowed to reconfigure its resources and switch its policy at S_1, S_3 and S_4, its total expected reward is 113.65 (higher than the reward 70.24 in the non-resource-reconfiguration case, but still much lower than the optimal reward 174.65 in the unconstrained case). Rather than predefining the phase-switching states, we now assume that $\lambda_1 = 0$, $\lambda_{i \in \{2,...,6\}} = 1$, and $\hat{\lambda} = 2$. That is to say, two additional phase-switching states besides the START state S_1 can be chosen by the agent from any states in the system.

We use the same transition probability $p_{i,a,j}$, reward $r_{i,a}$, initial probability distribution α_j, resource requirement cost $u_{o,a,i}$, capacity cost $\tau_{o,c}$, capacity limit $\hat{\tau}_c$, and the constant value X as in Section 2.2.3. The phase-switching costs λ_i and the cost limit $\hat{\lambda}$ are given above. Using Eq. 2.6, we can find $K = 3$, and the optimal integer solution to the mixed integer linear program Eq. 2.5 is:

$$[\Lambda_1, \Lambda_2, \Lambda_3, \Lambda_4, \Lambda_5, \Lambda_6] = [1, 0, 1, 0, 1, 0]$$

$$\begin{vmatrix} \Delta_1^1, \Delta_2^1, \Delta_3^1, \Delta_4^1, \Delta_5^1 \\ \Delta_1^2, \Delta_2^2, \Delta_3^2, \Delta_4^2, \Delta_5^2 \\ \Delta_1^3, \Delta_2^3, \Delta_3^3, \Delta_4^3, \Delta_5^3 \end{vmatrix} = \begin{vmatrix} 1, 0, 0, 0, 0 \\ 0, 0, 1, 0, 0 \\ 0, 0, 0, 0, 1 \end{vmatrix}$$

That is, the optimal set of phase-switching states is $S' = \{S_1, S_3, S_5\}$. By examining continuous variables $x_{i,a}^k$ (not shown here because there are too many of them), we can see that the total expected reward of the agent is 173.80 by choosing the resource o_1 and adopting the policy $[S_1 \rightarrow a_1, S_2 \rightarrow noop]$ at S_1, switching to the resource o_3 and the policy $[S_3 \rightarrow a_3, S_4 \rightarrow noop]$ when reaching S_3, and switching to the resource o_5 and the policy $[S_2 \rightarrow noop, S_5 \rightarrow a_5, S_6 \rightarrow noop]$ when reaching S_5.

2.5.2 Variation: Maximizing the Total Reward, Accounting for Cost

This subsection demonstrates the extensibility of our MILP-based algorithm by showing how easily it can be revised to work for another important variation of the S-RMP optimization problem where neither the phase-switching states are predefined (Section 2.4) nor the cost of creating phase-switching states is bounded (Section 2.5.1). We now assume that phase switching can occur at any state, and at as many states as desired, and that (similarly as in Section 2.5.1) there is a cost associated with letting a state be a phase-switching state. However, instead of being subject to some cost limits, these costs are now calibrated with the utility associated with executing policies. Now the optimization problem is to maximize the total expected reward, accounting for the costs of creating phase-switching states, without predetermining which are the phase-switching states or how many there will be. As shown below, designing an algorithm for such problems is trivial. It is just a simple mathematical reformulation of Eq. 2.5. The detail is presented in Eq. 2.7.

$$\max \sum_k \sum_i \sum_a x_{i,a}^k \times r_{i,a} - \sum_i \lambda_i \times \Lambda_i \tag{2.7}$$

subject to:

probability conservation constraints:

$$\sum_a x_{j,a}^k = \alpha_j^k + \sum_i \sum_a p_{i,a,j} \times x_{i,a}^k \qquad : \forall k, \forall j$$

$$\sum_k \alpha_j^k = \alpha_j \qquad : \forall j$$

$$x_{i,a}^k \geq 0 \qquad : \forall k, \forall i, \forall a$$

capacity constraints:

$$\frac{\sum_i \sum_a u_{o,a,i} \times x_{i,a}^k}{X} \leq \Delta_o^k \qquad : \forall o, \forall k$$

$$\sum_o \tau_{o,c} \times \Delta_o^k \leq \hat{\tau}_c \qquad : \forall c, \forall k$$

$$\Delta_o^k \in \{0,1\} \qquad : \forall o, \forall k$$

phase-switching constraints:

$$\frac{\alpha_j^k}{X} \leq \Lambda_j \qquad : \forall k, \forall j$$

$$\Lambda_j \in \{0,1\} \qquad : \forall j$$

where λ_i is the cost for creating phase-switching state i, and the objective function $\sum_k \sum_i \sum_a x_{i,a}^k \times r_{i,a} - \sum_i \lambda_i \times \Lambda_i$ represents the total expected reward of the policy minus the cost for creating phase-switching states.

Running Example

Let us revisit our running example to illustrate how the above algorithm can be used to solve the variation of the S-RMP optimization problem. Suppose that $\lambda_1 = 0$

Case	Phase-Switching States	Utility
unconstrained, Section 2.2.3	S_1, S_2, S_3, S_4, S_5, S_6	-75.35
non-phasing, Section 2.2.3	S_1	65.02
three fixed phases, Section 2.4	S_1, S_3, S_4	13.65
two additional phases, Section 2.5.1	S_1, S_3, S_5	73.80
unlimited phases, accounting for cost, Section 2.5.2	S_1, S_5	102.55

Figure 2.4: Comparison of the solutions to the example problem, given that the cost of creating each additional phase-switching state is 50.

(assuming the START state is already a phase-switching state) and $\lambda_i = c$ for any other state. Using the above MILP formulation (Eq. 2.7), we can find that when $0 < c \leq 0.85$ the optimal phase-switching states are $[S_1, S_3, S_4, S_5]$, when $0.85 < c \leq 21.25$, the optimal phase-switching states are $[S_1, S_3, S_5]$, when $21.25 < c \leq 87.53$, the optimal phase-switching states are $[S_1, S_5]$, and when $c > 87.53$ the optimal decision is not to create additional phase-switching states besides the START state S_1. As expected, the number of phase-switching states decreases as the cost of creating phase-switching states increases.

As a specific example, when $c = 50$, the optimal set of phase-switching states is $\{S_1, S_5\}$. The optimal resource configuration and executable policy in the phase initiated at S_1 are $\{o_3\}$ and $[S_1 \rightarrow noop, S_2 \rightarrow noop, S_3 \rightarrow a_3, S_4 \rightarrow noop]$, respectively; the optimal resource configuration and executable policy in the phase initiated at S_5 are $\{o_5\}$ and $[S_2 \rightarrow noop, S_3 \rightarrow noop, S_4 \rightarrow noop, S_5 \rightarrow a_5]$, respectively. The policy utility is 152.55 $(reward) - 50 \times 1$ $(cost) = 102.55$.

To help readers understand and compare this solution with the solutions derived in the previous sections, Figure 2.4 shows their solution utilities (where the utility is defined as the reward of the policy minus the cost of creating phase-switching states). Not surprisingly, the utility of the solution presented in this subsection is higher than the others since it is derived by the algorithm (Eq. 2.7) that can decide the optimal amount of cost for creating phase-switching states.

2.5.3 Variation: Cost Associated with Partial State Features

Note that, sometimes, making one state into a phase-switching state would provide "free" phase-switching feasibility for some other states, because whether a state supports phase-switching might be determined only by partial features in the state representation (and clearly it is possible that two distinct states have the same partial features). For example, in the rover domain, the state representation might include several other features (e.g., the direction that the rover faces) besides the feature "location", and so, if a supply station is built at a particular location for a particular state, then any state that has the same "location" feature as that particular state will allow for phase switching (without paying any additional cost) regardless of what its other features are.

Let us say that the MDP state space S consists of L disjoint subsets $\mathcal{S}_1$, $\mathcal{S}_2$, ..., $\mathcal{S}_l$, ..., $\mathcal{S}_L$ where if any state within $\mathcal{S}_l$ is a phase-switching state then all states in $\mathcal{S}_l$ are phase-switching states as well. Let λ_l denote the cost associated with $\mathcal{S}_l$, i.e., the cost of making any state in $\mathcal{S}_l$ into a phase-switching state, and let $\hat{\lambda}$ denote the cost limit for creating phase-switching states. Clearly, this is a generalization of the previous phase-switching constraint: when every $\mathcal{S}_l$ contains exactly one state, this representation is equivalent to the phase-switching constraint $\mathcal{R}$ previously presented in Section 2.1.

The new mixed integer linear program with the generalized phase-switching constraint is formulated in Eq. 2.8, which is very similar to Eq. 2.5, except for some minor revisions in the portion of phase-switching constraints.

$$\max \sum_k \sum_i \sum_a x_{i,a}^k \times r_{i,a} \tag{2.8}$$

subject to:

probability conservation constraints:

$$\sum_a x_{j,a}^k = \alpha_j^k + \sum_i \sum_a p_{i,a,j} \times x_{i,a}^k \qquad : \forall k, \forall j$$

$$\sum_k \alpha_j^k = \alpha_j \qquad : \forall j$$

$$x_{i,a}^k \geq 0 \qquad : \forall k, \forall i, \forall a$$

capacity constraints:

$$\frac{\sum_i \sum_a u_{o,a,i} \times x_{i,a}^k}{X} \leq \Delta_o^k \qquad : \forall o, \forall k$$

$$\sum_o \tau_{o,c} \times \Delta_o^k \leq \hat{\tau}_c \qquad : \forall c, \forall k$$

$$\Delta_o^k \in \{0, 1\} \qquad : \forall o, \forall k$$

phase-switching constraints:

$$\frac{\alpha_j^k}{X} \leq \Lambda_l \qquad : \forall k, \forall l, \forall j \in \mathcal{S}_l$$

$$\sum_l \lambda_l \times \Lambda_l \leq \hat{\lambda}$$

$$\Lambda_l \in \{0, 1\} \qquad : \forall l$$

where binary variable Λ_l denotes whether $\mathcal{S}_l$ is a phase-switching set.

Running Example

Let us go back to the running example. Suppose now that the state space is composed of $\mathcal{S}_{l=1} = \{S_1\}$, $\mathcal{S}_{l=2} = \{S_2, S_3\}$, $\mathcal{S}_{l=3} = \{S_4, S_5\}$, and $\mathcal{S}_{l=4} = \{S_6\}$, and that $\lambda_{l=0} = 0$, $\lambda_{l \neq 0} = 1$, and $\hat{\lambda} = 1$. The solution to Eq. 2.8 will yield a policy with a reward 165.68 using phase-switching states $\{S_1, S_4, S_5\}$, where the spending of one unit of cost creates both phase-switching state S_4 and phase-switching state S_5.

2.6 Experimental Evaluation

To this point, we have described a suite of single-agent resource-driven mission-phasing problems and techniques for solving them, using a simple example to illustrate these ideas. Ultimately, the significance of these techniques hinges on their computational efficiency in solving problems that are more difficult. In this section, we give an empirical evaluation of our techniques focusing on problems with a more complex state space and a larger resource set. Our experiments are implemented on a simplified Mars rover domain in which an autonomous rover operates in a stochastic environment. Following much of the literature on similar problems (Bererton et al., 2003; Dolgov and Durfee, 2006), the Mars rover domain is represented using a grid world.

2.6.1 Experimental Setup

In the grid world, there are some *wall* locations through which the rover cannot move. Each of other locations is associated with an execution resource, which, if held by the rover, can help the rover move safely in that location. Nonetheless, the agent can also move without holding any resource, but this will result in a high uncertainty in action outcomes and likely cause damage to the rover.

In addition, we say that there are multiple tasks randomly distributed in the grid world. When the rover reaches a location that possesses a task, if the rover currently carries the task-required execution resource, the rover can choose to perform a *do* action (that carries out the task) and receive a reward. Once any task is carried out, the mission is accomplished and the rover will leave the system.

The characteristics of results presented in this chapter are not sensitive to exact parameters used in our experiments, but for the sake of reproducibility, we describe the detailed parameters below. The procedure of building a random grid world is illustrated in Figure 2.5. When a $n \times n$ grid world is built, 40% of the locations are randomly chosen as wall locations, and 10% of the locations are randomly chosen as task locations. To avoid simple test problems, we only use grid worlds whose number of reachable locations (from the rover's starting location) is greater than half of the total number of locations (i.e., greater than $n^2/2$).

At each task location, there is a task that could be accomplished by the rover and generate a reward. To make the problem interesting and challenging, we distinguish tasks by setting different rewards for them. We sort tasks by their Manhattan distances to the starting location of the rover (the smallest distance first), and let the i^{th} task have a reward i. Therefore, it is not always true that the rover would desire and pursue high-reward tasks because low-reward tasks are closer to the rover and might be easier and safer to complete.

The rover always starts at the left bottom corner of a grid world, and its objective is to maximize its expected reward. At each time step, the rover chooses an action in its action set {*wait*, *up*, *left*, *down*, *right*, *safe-up*, *safe-left*, *safe-down*, *safe-right*, *do*}. Actions *wait*, *up*, *left*, *down*, and *right* can be executed without requiring the rover to carry any particular resource. In contrast, performing a safe-moving action *safe-up*, *safe-left*, *safe-down*, or *safe-right* in a non-wall location requires a particular resource (related to that location), which is randomly uniformly selected from resource set O when the problem is built. Analogously, performing action *do* at a task location requires a particular resource that is also randomly uniformly selected. It should be pointed out that performing an action needs at most one resource, but a resource

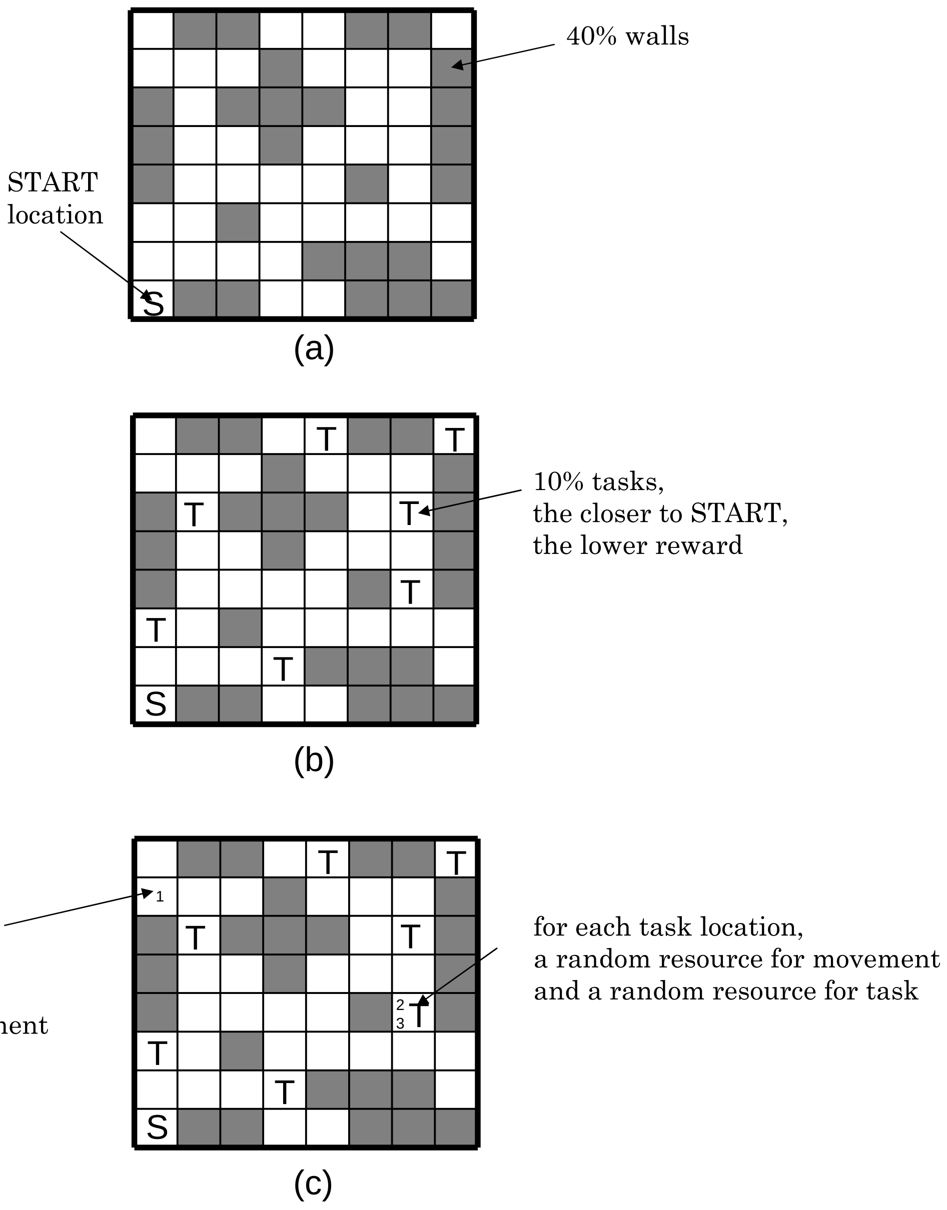

40% walls
START location
S
(a)
T
10% tasks, the closer to START, the lower reward
(b)
1
for each non-task location, a random resource for movement
2
3
for each task location, a random resource for movement and a random resource for task
(c)

Figure 2.5: The procedure of creating a random grid world. (a) 40% of the locations are randomly chosen as walls. (b) 10% of the locations are randomly chosen for tasks. (c) resource requirements are randomly set.

may enable the agent to safely move in multiple locations, and/or carry out multiple tasks. The resource requirement information is known to the rover *a priori.*

The following lists the detailed action parameters used in our experiments:

wait can be executed at any non-wall location without requiring any resource. After the execution of this action, the rover will stay at its current location with probability 0.95, and be out of the system with probability 0.05 (e.g., running out of battery).

up**,** ***down*****,** ***left*****,** ***right*** can be executed at any non-wall location without requiring any resource. Each of these actions achieves its intended effect with probability 0.4, moves the rover into each of the other three directions (except the intended direction) with probability 0.1, keeps the rover in the current location with probability 0.1, and causes damage to the rover (and then the rover is out of the system) with probability 0.2. Furthermore, if the rover bumps into a wall, it will stay at its current location.

safe-up**,** ***safe-down*****,** ***safe-left*****,** ***safe-right*** can be executed only in locations whose required resources are currently held by the rover. Compared to an unsafe-moving action, such a safe-moving action achieves the intended effect with a much higher probability 0.95, and falls into some failure situations with a lower probability 0.05. Similarly as before, when the rover bumps into a wall, it stays at its current location.

do can be executed only for tasks whose required resources are currently held by the rover. When action *do* is executed, the rover receives a reward, and leaves the system (since the mission is accomplished).

The capacity of the rover is restricted: the capacity limit is $\hat{\tau}$, and carrying each resource will incur one unit capacity cost. That is to say, the rover can carry no more than $\hat{\tau}$ resources.

We run experiments on a Core 2 Duo machine and use *cplex* 10.1 as our MILP solver. In our experiments, each average data point is computed from 20 randomly generated problems. We choose this number of random problems because, as shown in our experiments, 20 test problems (for each data point) are sufficient to illustrate the trend of our results while avoiding too long an experimental time to collect data (where the long time is because the prior standard algorithm, to which our approach is compared, finds optimal solutions slowly).

2.6.2 Optimality

We start the evaluation by showing the improved reward from using the phasing strategy over the approach that does not consider the possibility of switching resources in the midst of execution. Let us first consider the case where there are five supply stations distributed in the environment (the first station is always at the START state and the remaining four stations are randomly uniformly distributed in the grid world when the problem is generated). Other parameters are set as follows: $n = 8$, i.e., the size of the grid world is 8 by 8, and $|O| = 9$, i.e., there are nine different types of resources in the system. As shown in Figure 2.6 (i.e., the *5-fixed-phases* curve vs. the *non-phasing* curve), exploiting the resource reconfiguration opportunities (using the abstract MDP solver presented in Section 2.4) can considerably improve the performance of the rover, e.g., receiving a reward about 40% higher than the reward when not taking advantage of the supply stations, given that the rover can carry only three resources.

Figure 2.6 also compares the performance of the rover between the case where the locations of supply stations are randomly pre-selected and the case where the loca-

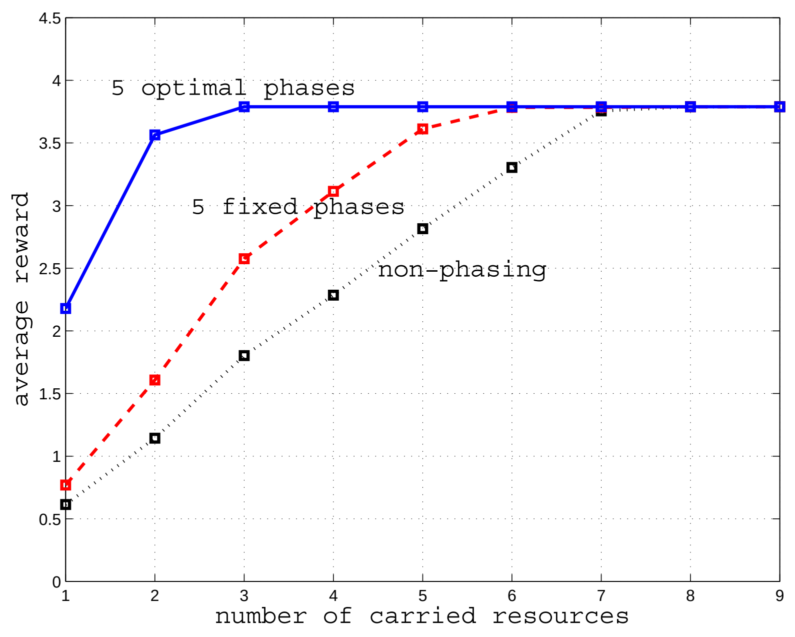


Figure 2.6: Exploiting fixed phase-switching states increases the agent's reward, and finding optimal phase-switching states further increases the reward.

tions of the same number of supply stations (i.e., five phases, given that $\lambda_{i=START} = 0$, $\lambda_{i \neq START} = 1$, and $\hat{\lambda} = 4$) can be determined by the rover itself. As expected, finding optimal phase-switching states (which can be done by using the MILP algorithm presented in Section 2.5.1) is of value in tightly constrained environments. For example, on average, it yields a reward about 46% higher than the approach that randomly selects phase-switching states when the number of carried resources is limited to $\hat{\tau} = 3$.

Figure 2.7 examines the effectiveness of the resource-driven mission-phasing approach from another perspective, showing the reward of the rover as a function of the number of phase-switching states that can be built in the environment (with other system parameters $n = 8$, $|O| = 9$, and $\hat{\tau} = 3$). We can see that (as expected) breaking the mission into multiple phases can significantly improve the total expected reward of the constrained rover. For example, setting up two additional

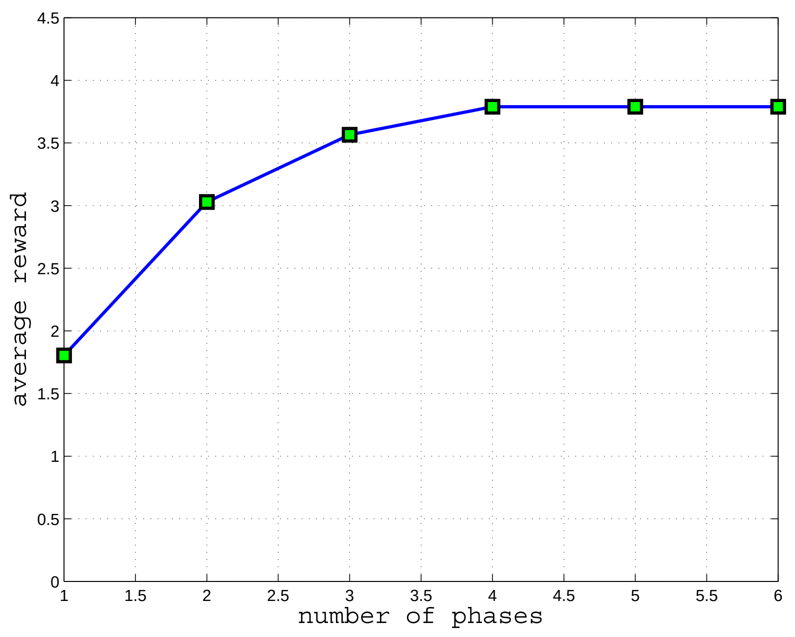
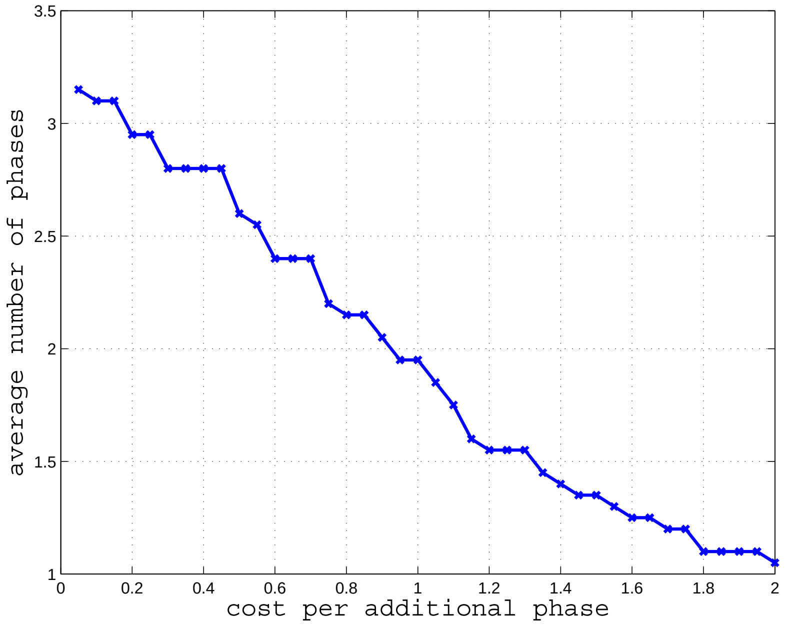


Figure 2.7: The reward increases as the number of phases increases.

supply stations in the 8×8 grid world environment (and so breaking it into three phases) can almost double the average reward that the rover can gain without using phasing.

We conclude the optimality evaluation by examining the MILP-based algorithm (presented in Section 2.5.2) in the case where supply stations can be built at any location, and at as many locations as desired, but creating each additional supply station (besides the existing one at the START location) will incur a cost c. Other parameters are the same as above. The problem objective is to maximize the net utility (the reward of the rover minus the cost of creating supply stations).

The empirical results in Figure 2.8 illustrate that our solution approach can wisely determine the number of phases that should be created, accounting for the cost of creating them. As shown in the top figure, when the cost is low, the approach makes a decision of creating 2.1 additional supply stations on average. In such cases, the rover can receive a reward close to the maximum reward (i.e., 3.77) that can be



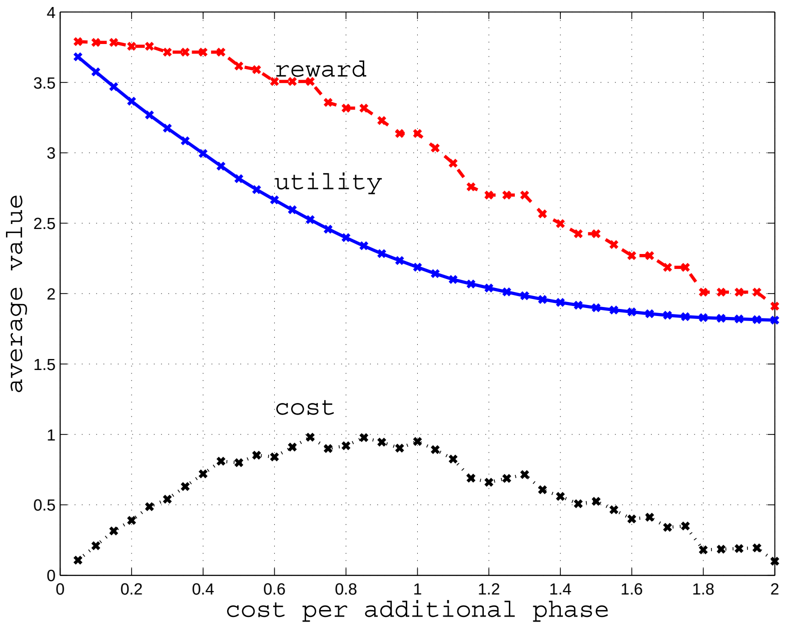


Figure 2.8: The impact of phase-switching cost on phases and utility. Top figure: the optimal number of phases decreases as the cost of creating each additional phase increases. Bottom figure: the expected utility of the system decreases as the cost of creating each additional phase increases.

gained in the unconstrained MDP case. On the other hand, when the cost is high (e.g., $c = 2$), the approach chooses not to create additional supply stations at all in most cases (and so the rover only configures resources at the START location). The resulting reward, cost, and utility (defined as reward minus cost) are shown as functions of the cost of creating each additional station in the bottom figure of Figure 2.8. Together with the results shown in Figure 2.6, we can clearly see that our approach (Eq. 2.7) yields a better solution than the approach of not using the phasing strategy that returns a constant utility 1.78 regardless of the cost parameter c. It is also better than the approach of always building a constant number (e.g., 2) of additional supply stations that can result in a negative utility when the cost parameter c is high.

2.6.3 Computational Efficiency

One of the major objectives of the work presented in this chapter is the design of a computationally-efficient solution approach for the S-RMP optimization problem. Section 2.3 has given a theoretical analysis on the computational complexity of the S-RMP optimization problem; this subsection is intended to empirically evaluate the efficiency of the solution approach presented in this chapter in solving complex S-RMP optimization problems. To make the presentation concise, only the runtime performance of the MILP-based algorithm described in Section 2.5.1 is shown, i.e., focusing on the standard S-RMP optimization problem defined in Section 2.1.[9]

Section 2.3 introduced two standard algorithms that may be used for the S-RMP optimization problem, including the brute-force search approach based upon enumeration and the MDP-based approach incorporating resource features in the MDP state representation. Enumerating all decompositions, and then, for each, enumerating all

[9]Our experiments also show that the trends of results for other variations of the S-RMP optimization problem are similar to those described in this subsection.

possible resource configurations and reconfigurations can be thought of as a (very slow) brute-force search algorithm for our formulated MILP. Therefore, we do not report its empirical results, since state-of-art MILP solvers (such as *cplex* which we use) usually follow more sophisticated branch-and-bound (B&B) strategies, and it is well established in the mathematical programming literature that the B&B approach is, in general, significantly better than the straightforward brute-force search (in both the runtime for finding an optimal solution and the anytime performance of finding a good solution). Based on an extensive search in the Artificial Intelligence and the Operations Research literatures, we have found that the MDP-based approach is the only existing approach (besides the brute-force search) that is directly applicable for the S-RMP optimization problem. We will thus focus on the comparison of our MILP-based algorithm and the MDP-based algorithm in the following discussion.

The MDP-based algorithm used in our experiment is a slightly revised version of that described in Section 2.3, where we reduce its exponential-size action space to a linear-size action space with the cost of making its state space reasonably bigger. Specifically, only a new "drop-all" action and $|O|$ new "pick-one" actions are added into the original action space (instead of adding $2^{|O|}$ "resource reconfiguration" actions). That is to say, rather than performing resource reconfiguration in one step, the agent now switches to a new bundle of resources by first implementing a "drop-all" action and then sequentially performing "pick-one" actions until it has all its desired resources. According to our experience, this revised algorithm is usually more computationally efficient than the version with the exponential-size action space.

Recall that the MDP resulting from incorporating resource features in the state representation is still a constrained MDP because phase-switching constraints place restrictions on which states resource-reconfiguration-related actions can be performed.

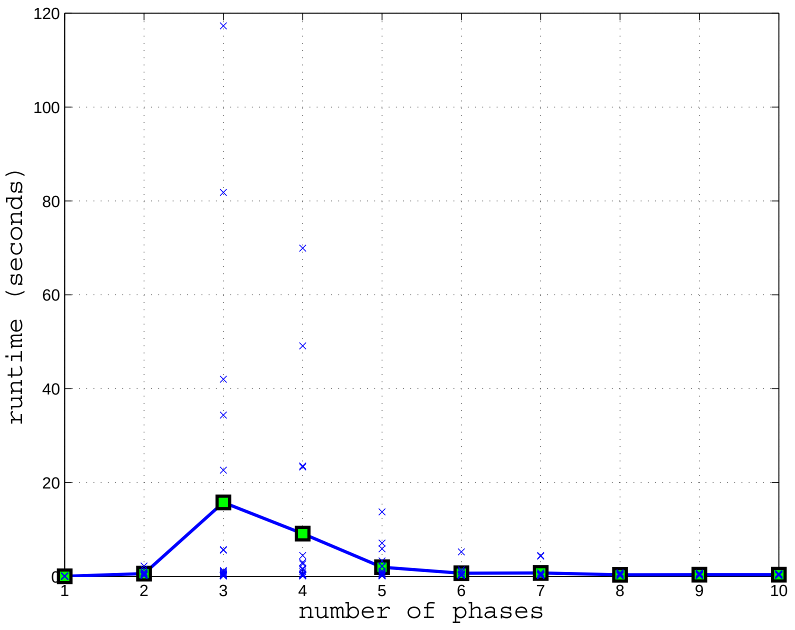


Figure 2.9: The runtime increases and then decreases as the number of phases increases.

The constrained MDP solver (Eq. 2.4) has been shown to be efficient in solving large-size constrained MDPs (Dolgov, 2006), and so this work uses it for solving such remodeled constrained MDPs.[10]

To provide a better idea about the computational complexity of our experiment domain and solution techniques, we begin by showing what a "hard" resource-driven mission-phasing problem is, particularly along the dimension of the number of phases that can be created. We use the same parameters as in Figure 2.7, but analyze runtime instead. The results are shown in Figure 2.9, which demonstrates how the running time for deriving an optimal S-RMP solution varies as the number of supply stations that can be created in the environment increases. In the figure, the solid line shows the average, and each data point, which is shown as "×", corresponds to a single run.

[10]In the MILP formulation for solving the remodeled constrained MDP, the number of binary variables equals the number of states specified in the S-RMP problem definition. That is, the runtime of the MDP-based algorithm is exponential to the input size but not doubly exponential.

As shown in the figure, the running time is low when the number of phases is small, and it gradually increases as the number of phases increases. This is not surprising, because the number of variables (both continuous variables and binary variables) in the MILP formulation is linear to the number of phases. However, the interesting discovery is that, after some point, the runtime starts to decrease although the size of the MILP still keeps increasing. We believe this is because, when the number of phases is large, there are a number of different ways to set up phase-switching states while achieving the same maximum reward. In other words, the S-RMP optimization problem with a large number of phase-switching states becomes under-constrained, and might have many different optimal solutions. The MILP-based algorithm presented in this work can effectively exploit this property, and reduce computational costs. Based upon this complexity profile, to highlight the ability of solving "hard" problem instances, the following experiments set the phase-switching cost limit $\hat{\lambda}$ to 2 (which means that there can be up to three phases in the system, assuming that creating each additional phase-switching state incurs one unit cost), unless we want to examine how the running time changes as the number of phases increases.

Figure 2.10 compares the average time for finding an optimal solution between our MILP-based algorithm and the standard MDP-based algorithm along the lines of the number of phases (top-left figure), the number of carried resources (top-right figure), the number of resource types (bottom-left figure), and the size of the grid world (bottom-right figure). We can see that our MILP-based algorithm is usually considerably faster than the MDP-based algorithm, particularly in complex problem instances.

In the top-left figure, the results of the MILP-based algorithm are the same as

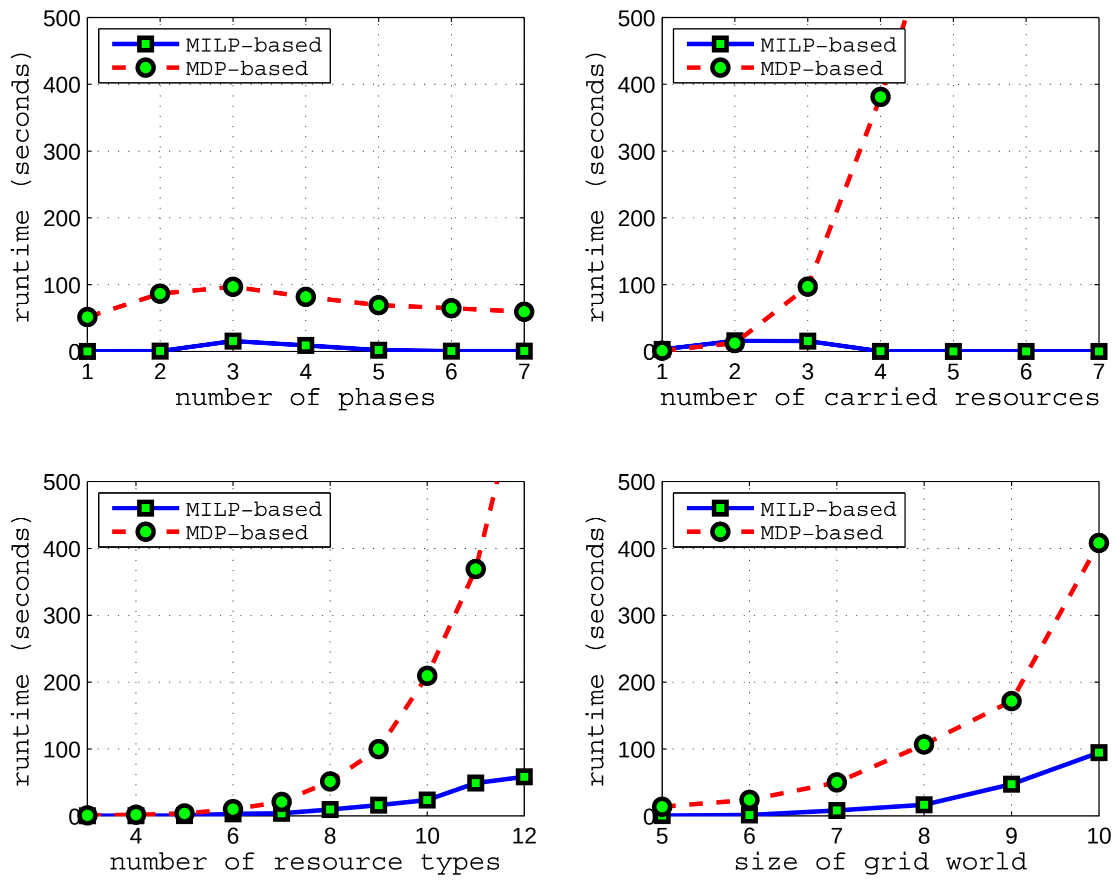


Figure 2.10: Runtime comparison between the MILP-based algorithm and the MDP-based algorithm. The MILP-based algorithm finds an exact solution to a S-RMP optimization problem faster than the standard MDP-based algorithm. Parameters are set as follows. Top-left figure: $n = 8$, $\hat{\tau} = 3$, $|O| = 9$, $\hat{\lambda} = \{0, 1.., 6\}$. Top-right figure: $n = 8$, $\hat{\tau} = \{1, ..., 7\}$, $|O| = 9$, $\hat{\lambda} = 2$. Bottom-left figure: $n = 8$, $\hat{\tau} = 3$, $|O| = \{3, 4, ..., 12\}$, $\hat{\lambda} = 2$. Bottom-right figure: $n = \{5, 6, ..., 10\}$, $\hat{\tau} = 3$, $|O| = 9$, $\hat{\lambda} = 2$.

those shown in Figure 2.9, which have already been discussed. Interestingly (but not surprisingly), unlike the other three figures, the curve of the MDP-based algorithm in this figure does not monotonically increase as the value of the input parameter increases. This is because the input parameter in this figure, the number of phases, does not affect the size of the state space of the expanded MDP. Furthermore, the constrained MDP method (Eq. 2.4) used to solve the expanded MDP can exploit problem structure when the problem becomes under-constrained. This explains why the running time decreases after some point (but the time is still much higher than that of the MILP-based approach).

The top-right figure also demonstrates a trend for the running time of the MILP-based algorithm decreasing after the value of the input parameter (i.e., the number of resources that can be carried by the rover) is above a particular threshold. The reason is similar to that used to explain Figure 2.9 — the MILP-based algorithm can effectively discover and exploit the fact that the problem becomes under-constrained. In contrast, the MDP-based algorithm incorporating resource features into the state representation leads to a MDP whose size grows very rapidly as the number of resources that can be carried increases, and thus results in a significant increase in the running time.

As illustrated in the bottom-left figure and the bottom-right figure, the runtime of the MILP-based algorithm also increases considerably slower than the MDP-based algorithm, although, unlike the top-left and top-right figures, the runtime monotonically increases as either the number of resource types or the size of the grid world increases (because in general the increases of these two parameters will not make the problem become under-constrained by themselves).

The reason for the significant reduction in computational cost is that our MILP-based approach can formulate the S-RMP optimization problem in a compact (as opposed to exponential) formulation, which paves the way for taking advantage of state-of-art MILP solvers to effectively solve the coupled problems of problem decomposition, resource configuration, and policy formulation. It is important to emphasize that the MILP-based approach uses no approximation techniques (and so it will find optimal solutions). The compactness of the formulation is because the MILP-based approach folds the process of solving a NP-complete S-RMP problem into the process of solving a NP-complete MILP (where the MILP can be solved efficiently by state-of-art solvers).

Specifically, the MDP-based approach models resources in the MDP representation regardless of valuations of subsets of the resources, and then it reasons over the generalized MDP to determine an optimal way of configuring and reconfiguring resources. In contrast, our MILP-based solver finds an exact S-RMP solution by taking advantage of the embedded branch-and-bound MILP method to discard subsets of fruitless candidate solutions (through upper and lower estimated bounds). Although the MILP-based approach and the MDP-based approach have similar worst-case runtime, i.e., requiring exponential time to enumerate all possible ways of sequentially configuring resources (which is reasonable because S-RMP is NP-complete), the average-case performance of the MILP-based approach is often much better than the MDP-based approach because of the effectiveness of the branch-and-bound algorithm for pruning suboptimal solutions. This is particularly helpful in cases where suboptimal decompositions can be detected easily and early because a large number of possible resource configurations and executable policies can then be discarded without much computational effort.

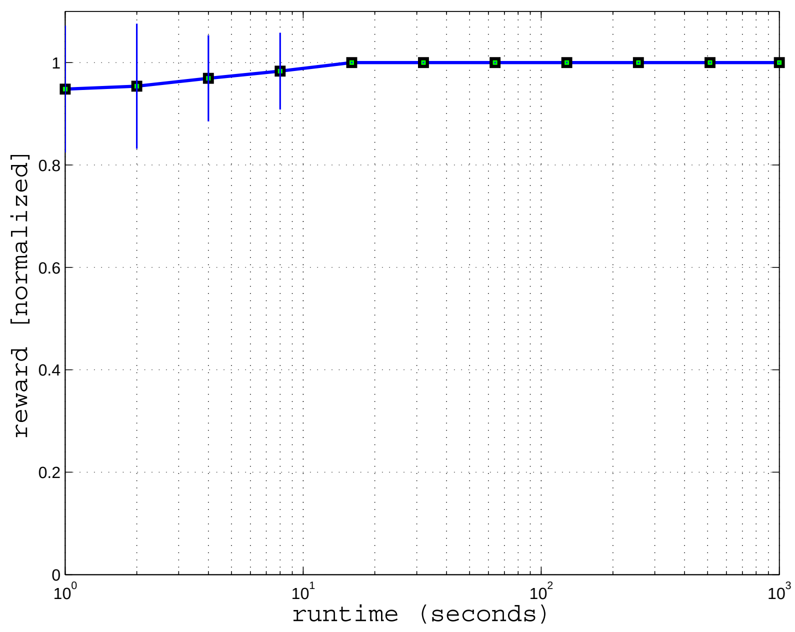
2.6.4 Anytime Performance

The primary goal (and contribution) of the work presented in this chapter is the development of a computationally efficient approach for finding an exact solution to a S-RMP optimization problem. The empirical results shown previously in this evaluation section have demonstrated and confirmed the effectiveness and efficacy of our approach in deriving an optimal solution. Nevertheless, it is worth pointing out that the MILP-based algorithm can also be used to find a high-quality solution within limited time through exploiting the anytime performance of state-of-art MILP solvers. We conclude this section with a preliminary examination of this ability.

Figure 2.11 presents empirical results under two different experimental settings, where the x-axis represents the runtime in a logarithmic scale, the y-axis represents the normalized reward,[11] and the error bars on the plot show the standard deviation. These results illustrate that the MILP-based algorithm performs reasonably well in finding a high-quality solution. For examples, in the problem with a 9×9 (12×12) grid world where finding an exact solution takes about 40 (570) seconds on average, our MILP-based algorithm can find a near-optimal solution with above 95% of the optimal solution quality within 4 (30) seconds on average.

Since the design of our S-RMP solver focuses on the capability of quickly finding an exact solution, we believe that some alternative solution approaches (such as heuristic search and factored MDPs) may have better anytime performance than the approach presented in this chapter. The significance of our MILP-based anytime S-RMP solution approach is that it does not rely on any particular domain-specific heuristic and knowledge, highlighting the ability to automate the process of creating and using mission phases in complex environments.

[11]The reward of the optimal solution is normalized to one for each test problem.



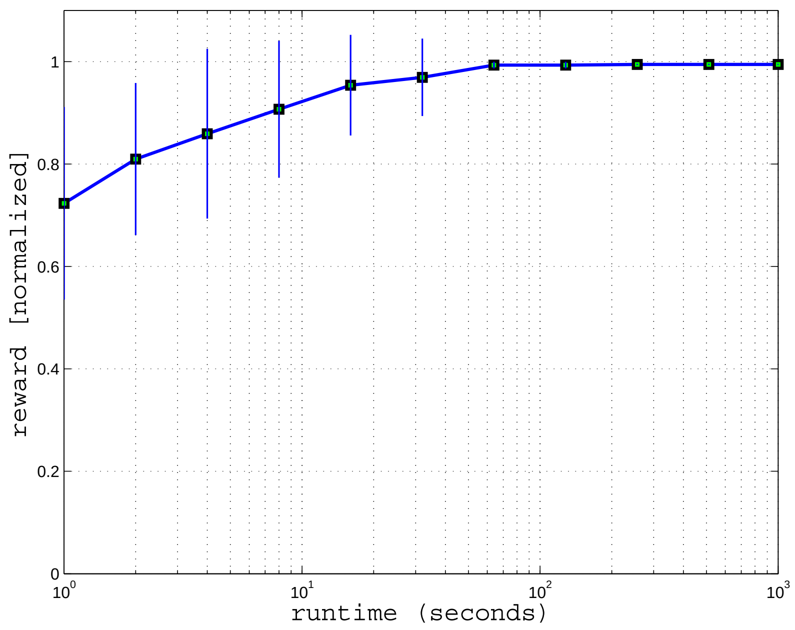


Figure 2.11: Anytime performance of the MILP-based algorithm. Parameters are set as follows: $\hat{\tau} = 3$, $|O| = 9$, $\hat{\lambda} = 2$, $n = 9$ (top figure), and $n = 12$ (bottom figure).

2.7 Related Work

The S-RMP optimization problem involves three intertwined component problems: mission (problem) decomposition, resource configuration, and policy formulation. Each of these S-RMP component problems has been studied in a wide variety of research fields. The combinations of any two of them have also gained much attention in recent years. This section gives an overview of related work, and discusses why those prior approaches are not directly applicable to the S-RMP optimization problem of interest in this chapter.

As was presented in Section 2.1, the S-RMP optimization problem is defined by extending an unconstrained MDP model to include agent capacity constraints and phase-switching constraints. The organization of this section follows the way of that definition. It begins with a discussion of policy formulation techniques, followed by a discussion of resource configuration techniques. It then reviews problem decomposition techniques and their combinations with policy formulation and/or resource configuration work. This section concludes with a discussion of the "mode-transition" research that is related to this work but does not fit clearly into the previous categories.

Policy Formulation. The well-known Markov decision process has been described in Section 2.2. By formulating a sequential decision-making problem into a MDP model, a number of efficient (polynomial-time) solvers, such as the value iteration and policy iteration algorithms, can be used to compute an optimal policy (Puterman, 1994).

However, directly applying these algorithms in resource-constrained systems, such as the resource-driven mission-phasing problem studied in this chapter, typically involves incorporating resource features in the MDP state representation (and so actions can be conditioned on resource availability), which will result in an exponential increase in the size of the state space (Meuleau et al., 1998), i.e., the well known "curse of dimensionality" challenge. It has been shown in the empirical results (Section 2.6.3) that the exponential-size state space can result in computational inefficiency.

Resource Configuration. Motivated by the fact that in a number of domains (such as the Mars rover domain) it is impossible (or too expensive) to resolve resource constraints by enhancing the agent's architecture, improving the performance of a constrained agent under its limited architecture has been an active subject in recent years, i.e., a class of "bounded optimality" study (Russel, 2002). The Cooperative Intelligent Real-Time Control Architecture (CIRCA) is one such research effort (Musliner et al., 1993, 1995). CIRCA uses a simple myopic approach to compute feasible policies. It starts with building an optimal unconstrained policy without worrying about its real-time requirements, and then myopically repairs the policy until executable on the real-time system.

Not surprisingly, the (fast) myopic approach adopted by CIRCA might result in suboptimal policies that cannot fully utilize the agent's capacity. Several other recent studies have proposed alternative algorithms for searching for a policy that is executable within the agent capacity constraints and that optimizes the expected (possibly discounted) reward accrued over the entire agent execution. For example, Altman (1998) adopted a Lagrangian and dual LP approach to solve constrained MDPs with total cost criteria. Feinberg (2000) analyzed the complexity of constrained discounted MDPs. Of particular relevance to the work in this dissertation is the study of strongly-coupled resource allocation and policy formulation problems by Dolgov and Durfee (2003, 2006). Their approach implements simultaneous combinatorial optimization and stochastic optimization via reduction to mixed integer linear programming, which has been recapped in Section 2.2.

However, these prior studies on constrained agents are based upon the assumption that the agent's limited capacity is configured by the resources it procures prior to execution but cannot be reconfigured during plan execution. In other words, they do not consider the possibilities of reconfiguring capacity usage in the midst of execution, and do not work on optimizing the use of such opportunities. A seemingly feasible solution to the S-RMP optimization problem is to enumerate all possible ways of decomposing a mission, then, for each decomposition, adopt existing (one-shot) constrained optimization techniques to derive an optimal executable policy in each phase independently, and finally combine these optimal phase solutions into an overall solution. However, this approach does not work in most cases. It not only suffers from a large number (often exponential in the number of MDP states) of possible ways of problem decomposition, but, more importantly, a policy in one phase can usually only be optimized with respect to the policies planned for subsequent phases that might be entered.

Problem Decomposition. In the literature of stochastic planning, a number of decomposition algorithms have been proposed to speed up the planning process. The discovery of "recurrent classes" of MDPs is one such decomposition strategy, which can discover an exact state space decomposition in an environment with uncertainties (Puterman, 1994; Boutilier et al., 1999). A recurrent class represents a special *absorbing* subset of the state space, which means that once an agent enters a recurrent class it remains there forever no manner what policy it adopts. Puterman (1994) has suggested a variation of the Fox-Landi algorithm (Fox and Landi, 1968) to discover recurrent classes. With the discovery of the recurrent classes, the MDP solver can derive an optimal overall policy by building an optimal policy in each recurrent class independently and then constructing and solving a reduced MDP consisting only of transient states (i.e., removing the recurrent classes in the MDP).

Of course, not all application problems can be exactly decomposed into independent sub-problems. However, many of them are composed of multiple weakly-coupled sub-problems where the number of states and transitions connecting two neighboring sub-problems is relatively small. A number of heuristic decomposition methods have been designed to exploit such weakly-coupled relationships. As an example, in the robot navigation domain (Parr, 1998; Precup and Sutton, 1998; Lane and Kaelbling, 2001), doorways (or similar connection structures, such as bridges) can be used to break a large environment into blocks of states, e.g., one block for each room. Two neighboring blocks are only connected by a small number of *doorway* states. Once a weakly-coupled state space is decomposed into several pieces, there are a few methods that can be used to efficiently build an overall policy based upon sub-problem policies. One common method is to let each sub-problem iteratively exchange information with its neighboring sub-problems, and repeatedly revises (if necessary) its sub-policy based upon its updated knowledge about utilities or values of its neighbors until an overall (approximately) optimal solution is derived (Dean and Lin, 1995).

Besides the application in stochastic planning, decomposition techniques have also been shown to be beneficial for resource management in many realistic application domains. Several resource allocation algorithms have been developed for the problem of allocating a set of heterogeneous resources with availability constraints to maximize a given utility function (Wu and Castanon, 2004; Palomar and Chiang, 2006; Reveliotis, 2005). For example, Wu and Castanon (2004) presented an approximation solution algorithm using decomposition combined with dynamic programming, and their experimental results showed that the algorithm produces near-optimal results with much reduced computational effort.

In addition to the Artificial Intelligence (AI) techniques discussed above, decomposition techniques, which are often integrated with hierarchical control (also called multilevel control in some literature), have received much attention in recent years in Operations Research, Operations Management, Systems Theory, Control Theory, and several other fields (Sethi et al., 2002; Antoulas et al., 2001; Xiao et al., 2004; Phillips, 2002; Teneketzis et al., 1980). Many manufacturing systems are large and complex; the management of such systems requires recognizing and reacting to a wide variety of events that could be deterministic or stochastic. Obtaining exact optimal policies to run these systems is often very difficult both theoretically and computationally. By exploiting the fact that real-world systems are often characterized by several decision sub-systems, e.g., a company consists of departments of marketing, production, personnel, and so on, one popular way to deal with the computational complexity challenge is to develop methods of hierarchical decision-making for these systems. The fundamental ideas are to reduce the overall complex problem into multiple smaller, manageable sub-problems, to solve these sub-problems, and to coordinate solutions of the sub-problems so that overall system objectives and constraints are satisfied (Sethi et al., 2002).

To summarize, it is well established that utilizing decomposition can greatly reduce computational costs in many situations. However, all the aforementioned prior decomposition techniques are not directly applicable for the S-RMP optimization problem. The underlying reason is that decomposition points that are good at reducing computational efforts are not necessarily (and, very likely, irrelevant with) the optimal points for constrained agents to reconfigure resources. It is worth emphasizing that S-RMP decomposition tackles capacity constraints instead of computation time constraints (where computation time constraints will be discussed and addressed in computation-driven mission-phasing (CMP) techniques that will be presented in Chapters IV and V). Indeed, in general, the mission decomposition in the S-RMP solution will not in itself reduce computational requirements because a policy in one phase can usually only be optimized with respect to the policies planned for possible subsequent phases.

Mode Transition. Finally, it is important to distinguish the resource-driven mission-phasing research from the "mode-transition" research implemented in the fields of Operations Research and Control Theory (Schrage and Vachtsevanos, 1999; Wills et al., 2001; Karuppiah et al., 2005). At first glance, these two research fields have a lot in common: they both work on transitions from one sub-problem to another, and both take into account resource reconfiguration. However, it should be pointed out that they emphasize distinct aspects, and are applicable for different application domains.

First of all, in the mode-transition approach, operational modes are usually tightly associated with some explicit actions (e.g., *hover* and *fly-forward* modes in the helicopter example described by Schrage and Vachtsevanos (1999)), corresponding to some particular states (e.g., *sleep*, *search*, *seed*, and *final* modes defined by Bojinov et al. (2002)), or characterized with some explicit purposes (e.g., passing through a narrow tunnel and then traversing rough terrain requires a self-reconfiguring robot to adjust its shape to achieve its goal better (Rus and Vona, 2001)). In contrast to the explicit definition or representation of modes in the mode-transition research, phases in the S-RMP problem are usually much more difficult to identify. The phasing information is hidden in the MDP model, and finding optimal phases is usually a challenging task.

Second, in the mode-transition research, mode transition and resource reconfiguration are often triggered by real-time events, e.g., responding to an unexpected disastrous event and reconfiguring resources for fault toleration (Drozeski, 2005). In contrast, the resource-driven mission-phasing study assumes that a decision-making agent has complete information about the environment prior to its execution, and one of its main objectives is to find the optimal points for reconfiguring resources and capacity usage. That is, phase switching in S-RMP is a choice of the agent instead of a reactive response to an exogenous event. More specifically, the S-RMP technique presented in this chapter utilizes sequential decision-making (for look-ahead) to identify optimal resource reconfiguration and policy switching states. It emphasizes how to reconfigure resources and switch policies so that the agent would not (or would be less likely to) enter into the predicament of encountering undesirable events, instead of studying how to reconfigure resources in real-time to respond to an unexpected event.

Finally, much prior mode-transition research, particularly in the Control Theory literature, investigates how to perform a smooth functional transition among modes, but the work in this dissertation simply assumes that there are *aggregate* resource (re)configuration actions, each of which can be a sequence of primitive actions of arranging resources. This dissertation does not address the details of how the agent *mechanically* implements mode-transition and resource-reconfiguration actions.

2.8 Summary

In this chapter, we have analyzed several variations of a single-agent resource-driven mission-phasing problem, corresponding to several cases of phase-switching constraints, and presented a suite of computationally efficient algorithms for finding and using mission phases. We have shown through analysis and experiments that our approach can considerably reduce the computational cost for finding an exact solution to a complex S-RMP optimization problem in comparison with prior approaches.

The contributions of the work presented in this chapter are summarized as follows:

- The work explicitly takes into account potential opportunities of a capacity-limited agent for reconfiguring its capacity usage in the midst of mission execution, and develops an abstract MDP solver to help the constrained agent optimize the use of the existing phase-switching states. As shown in our empirical results (Figure 2.6), exploiting such resource reconfiguration and policy-switching opportunities can considerably increase the reward of the constrained agent.

- The work designs a novel MILP-based algorithm to automate the process of finding and using mission phases in complex, stochastic environments, which eliminates the need of having phases predefined in the description of a mission. With this algorithm, the mission-phasing strategy can be generalized to a wide variety of application domains.

- The MILP-based algorithm presented in this chapter is computationally efficient. It formulates a complex mission-phasing problem into a compact mathematical formulation, and simultaneously solves three component problems —

mission decomposition, resource (re)configuration, and policy formulation — to exploit the problem structure. The empirical results (presented in Figure 2.10) have shown that the approach makes a significant reduction in computational cost, compared to the standard MDP-based approach.

- Much prior constrained-agent research either does not consider the possibility of reconfiguring the usage of agent capacity during execution, or only studies how to reactively reconfigure resources in response to an exogenous event, or uses some simple strategy to manage resources (and so, often sub-optimally). To the best of our knowledge, our problem model and solution algorithms, based upon the MDP model and sequential decision-making theory, are the first computationally efficient approach for optimally solving the coupled problems of mission decomposition, resource configuration, and policy formulation in constrained, stochastic environments.

CHAPTER III

Resource Reallocation in Multi-Agent Systems

In multi-agent systems, besides capacity limitations of each agent, there often exist other resource constraints, caused by a group of agents sharing a limited set of resources. Typically, the resources taken by one agent can affect the resources available to other agents. That is to say, in resource-limited environments, an individual agent may be unable to procure all of its desired resources (even when its capacity does not restrict the amount of resources it can hold) because some other agents may be interested in those resources as well. How the resources are allocated among the agents will dictate the actions each agent will be capable of performing, and thus how the agents will act and interact to accomplish their goals in their environments. Making effective resource-allocation decisions is of importance to such systems where a group of agents share scarce resources.

Commonly, the problem of determining an optimal resource allocation among agents operating in a stochastic world is computationally challenging. Assessing the value of a particular bundle of resources to an agent requires the agent to formulate an optimal policy based only on the actions that the bundle of resources enables. Since the number of resource bundles is exponential in the number of resources, the policy optimization process may have to be solved an exponential number of times.

Furthermore, once the agents have computed valuations for each of the resource bundles, identifying an optimal assignment of the bundles to the agents may require enumerating all possible combinations of resource bundles, which could lead to a doubly-exponential-time algorithm. Fortunately, a much more efficient algorithm has been designed that can solve the coupled problems of combinatorial optimization (for resource allocation) and stochastic optimization (for policy formulation) efficiently through exploiting problem structure (Dolgov and Durfee, 2005, 2006).

However, making optimal *one-shot* resource allocation decisions, though an important step towards improving the effectiveness of resource utilization, has not yet solved the problem of optimizing the use of the limited resources. One reason is that an agent may only need a resource for a particular task within a particular time period. As a Mars rover example where multiple rovers share limited instruments, a rover might no longer need a previously assigned instrument after accomplishing a particular scientific experiment, and so another rover can request this instrument to better carry out its remaining experiments (and vice versa). This suggests that reconfiguring resource assignments among a group of agents in the midst of mission execution could be a promising way to improve the system performance in resource-constrained multi-agent environments.

The prior work (Dolgov and Durfee, 2005, 2006) would generate an optimal allocation of resources assuming that, once the resources were distributed among the agents, the initial allocation would persist throughout the remainder of the agents' execution. The work in this chapter relaxes that assumption, and instead investigates the implications of allowing agents to redistribute resources among themselves in the midst of execution. This leads to solving the problem of *sequential* (as opposed to one-shot) resource allocation, along with the problem of optimizing agents' policies for the execution phases between neighboring reallocations.

To solve this sequential multi-agent resource-allocation problem, we extend our techniques from the S-RMP optimization problem presented in Chapter II. The work in that chapter analyzed how a single agent should select a different subset of the available resources at different times, and developed strategies for deciding on the optimal states at which to reconfigure resources and switch policies, given constraints and/or costs on such states. Analogously, the sequential resource allocation study in this chapter analyzes how a group of agents should sequentially reconfigure the distribution of the limited resources among themselves, and is to develop automated resource-driven mission-phasing techniques for solving three intertwined problems — problem decomposition, resource allocation, and policy formulation — in multi-agent environments with resource constraints and with uncertainties.

The rest of this chapter is organized as follows. Section 3.1 gives a formal problem definition, followed by a background introduction in Section 3.2 and complexity analysis in Section 3.3. Section 3.4 and Section 3.5 look at several increasingly general variations of sequential resource allocation problems, and for each, present, analyze, and illustrate solution algorithms. Both efficiency and optimality of our techniques are evaluated in Section 3.6. Finally, Section 3.7 summarizes the contributions of the work presented in this chapter.[1]

3.1 Problem Definition

Stochastic planning in multi-agent environments is typically much more challenging than that in single-agent environments, particularly when each agent has only a partial view of the global environment. Previous complexity analyses have shown that the general decentralized Markov decision process (Dec-MDP) is NEXP com-

[1]This chapter is largely based on work that was originally reported in (Wu and Durfee, 2007a).

plete (Bernstein et al., 2000; Goldman and Zilberstein, 2004), which means that exactly solving a Dec-MDP may be extremely difficult. Fortunately, in many application domains, the actions taken by one agent may not impact other agents' transitions. For example, when a few delivery robots operate in a large open environment, interactions may be rare and easily avoidable. The development of efficient algorithms for such *loosely-coupled* systems has gained much attention among many researchers (Meuleau et al., 1998; Becker et al., 2004; Dolgov and Durfee, 2005).

In keeping with the prior work, the work in this chapter focuses on loosely-coupled multi-agent systems[2] where a group of cooperative agents are coupled through sharing resources (i.e., actions selected for one agent might restrict the actions available to the others), but the actions executed by one agent cannot impact rewards and transitions of the others. As is a common assumption in the resource-allocation research literature, this work also assumes that, once the resources are distributed, the utility that each agent can achieve is only a function of its assignment of the resources and does not depend on what resources are given to other agents and how they use these resources. In addition, it is here assumed that a scheduled resource reassignment can always succeed at its scheduled time point. At the end of this dissertation, the discussion of future work will talk about the implications of relaxing this assumption. Finally, to simplify the presentation and make the discussion clearer, the discussion in the rest of this chapter will not consider and model capacity limits of the agents. However, the techniques presented in this chapter can be easily extended to also include agent capacity constraints.[3]

In a similar fashion to the S-RMP optimization problem that was presented in

[2]A general multi-agent mission-phasing problem can be solved exactly (but perhaps not efficiently) by using the S-RMP solution approach presented in Chapter II on the joint state and action space of the interacting agents, assuming that each agent has a full view of the joint state.

[3]A "dummy" agent may be created to hold unallocated resources.

Chapter II, a *multi-agent resource-driven mission-phasing* (M-RMP) problem can be defined as a constrained optimization problem with the following inputs $\mathcal{M}$, α, $\mathcal{C}$, and $\mathcal{R}$:

- ☐ $\mathcal{M} = \{\mathcal{M}^m\}$ is a set of classical MDPs, where $\mathcal{M}^m$ represents agent m's MDP and it can be modeled in the same way as described in Section 2.2. That is, $\mathcal{M}^m = \langle S^m, A^m, \{p^m_{i,a,j}\}, \{r^m_{i,a}\}\rangle$ where S^m is a finite state space of agent m, A^m is a finite action space of agent m, $p^m_{i,a,j}$ is the probability that agent m reaches its state j when it executes action a in its state i, and $r_{i,a}$ is the reward that agent m can receive when it performs action a in its state i.
- ☐ $\alpha = \{\alpha^m_i\}$ specifies the initial probability distribution over states, where α^m_i is the probability that agent m is initially in its state i.
- ☐ $\mathcal{C}$ represents resource constraints in the system, which can be represented as $\langle O, U, \hat{\Omega}\rangle$:
  - ◇ O is a finite set of shared, indivisible, non-consumable execution resources.
  - ◇ $U = \{u^m_{o,a,i}\}$ represents resource requirements of agents, where the binary parameter $u^m_{o,a,i} \in \{0, 1\}$ indicates whether agent m requires resource o to execute action a when it is in its state i.
  - ◇ $\hat{\Omega} = \{\hat{\omega}_o\}$ specifies resource limitations, where $\hat{\omega}_o$ is the maximum amount of resource o that could be shared by agents in the system.
- ☐ $\mathcal{R}$ specifies constraints on resource reallocation. We remain agnostic in this work as to the means by which resource redistribution occurs. In particular, we do not require agents to be in same (physical) state(s) to exchange resources. We assume that if the agents have agreed to redistribute resources at a particular

time then resources can always be successfully collected and reassigned at that time. We capture the efforts required for such resource reallocation activities as costs $\langle \{\psi_t\}, \hat{\psi} \rangle$:[4]

- ◇ $\{\psi_t\}$ indicates resource reallocation costs, where ψ_i denotes the cost for reconfiguring the resource assignment at time t.[5] Note that ψ_t is only associated with time t regardless of what resources and how many of them are reassigned. A variation of resource reallocation constraints where the reallocation cost depends on the amount of resources being transferred will be discussed and analyzed in Section 3.5.2 after presenting the solution algorithm to the M-RMP optimization problem defined in this section.
- ◇ $\hat{\psi}$ specifies the limit on the amount of cost that could be spent in resource reallocation. For a Mars rover example, $\psi_{t=any\_time} = 1$ and $\hat{\psi} = 4$ might say that at most four resource reconfiguration events could be scheduled during a particular mission execution.

M-RMP deals with a particular class of multi-agent stochastic planning problems where agents can communicate as much as they need before the start of the mission, but, once the mission starts, the distributed nature of the environment would prevent the agents from exchanging information further. That is to say, the central decision-making agent[6] has complete world information in the pre-execution planning stage, but each agent can observe only its local state during execution (but with common knowledge of global time). In general, a multi-agent planning problem where central

[4]This is different from *buffer pool* research (Lehman and Carey, 1986; Sacco and Schkolnick, 1982), which often assumes that buffer size can be changed immediately and free of charge.

[5]In cases where the costs of reconfiguring resources are associated with joint states of interacting agents instead of only their time points, if joint states are fully observable during execution, modeling a M-RMP problem as a S-RMP problem (which was solved in the previous chapter) on the joint state space is a reasonable solution approach since the joint state space has to be modeled and considered in such cases.

[6]The decision-making agent can be one of the cooperative agents operating in the environment, or the entire group of the agents (if using distributed mathematical programming techniques), or even some other agent outside the group.

planning is followed by distributed execution would lead to a NEXP-complete decentralized MDP problem because planning should account for history of interactions among agents in such cases (Bernstein et al., 2000; Goldman and Zilberstein, 2004). However, recall that M-RMP assumes that agents are transition independent and reward independent. That is to say, we only need to consider interactions of resource sharing among agents. Resource assignments change as time passes, and time t always reaches $t+1$ in the next step (i.e., in a deterministic way), which implies that we may be able to exploit this property and work out a particular, more efficient algorithm. It should be emphasized that M-RMP is an *open-loop* decision-making problem where the resource reallocations will occur as scheduled regardless of specific execution trajectories of each agent. That is to say, resource allocation decisions are made in the pre-execution planning stage, by accounting for uncertainty over the durations of needs of the resources and the probability distributions over states at the particular times of entering phases, but the way of allocating resources is not affected by particular execution trajectories. For example, resources will not be redistributed earlier even if every agent finishes using its assigned resources earlier than expectation in a particular run since M-RMP assumes that the agents are unable to communicate each other during execution.

The objective of the M-RMP optimization problem is to maximize the total expected reward of a group of agents within a finite time horizon by judiciously reallocating the limited, shared resources among the agents over time. To meet this need, the solution algorithm needs to determine when to reallocate resources, i.e., find an ordered set of time steps $\{t^k\}$ to decompose the overall problem into multiple phases. Phase k starts at time t^k (at which point the resource allocation changes) and lasts until the resource allocation changes again (and the agents enter the subsequent phase at time t^{k+1}). In addition, for each phase k, the algorithm needs to determine how to allocate resources, i.e., finding an optimal resource allocation for each phase. Resource allocation decisions are made in the pre-execution planning stage, by accounting for the joint state probability distributions at the particular times of entering the phases, but the way of allocating resources does not depend on specific joint states (given that joint states are not observable). Finally, the algorithm needs to determine how to use resources. Note that, when problem decomposition and resource allocation decisions are made, the M-RMP problem would be reduced to a transition-independent and reward-independent multi-agent MDP problem, which is indeed equivalent to multiple single-agent MDP problems where action choices of each agent only depend on the agent's own current state and are neither affected by the agent's historical states nor by actions of other agents.

Although much simpler than a general decentralized MDP problem, such an automated multi-agent mission-phasing problem is still computationally challenging because it needs to determine not only how to initially allocate limited shared resources, but also when to reallocate resources, what the best way of reallocating resources is, and what the best executable policies with respect to the reallocated resources are. These three component problems — mission decomposition, resource allocation, and policy formulation — are strongly intertwined. The utility of decomposing a problem into phases and the utility of allocating resources for each phase are unknown until executable policies are formulated and evaluated, but the policies cannot be formulated until the phases are built and the resources are allocated.

3.2 Background: Integrated Resource Allocation and Policy Formulation

This section is to familiarize readers with some background knowledge about how the prior approach (designed by Dolgov and Durfee (2005, 2006)) solves integrated resource-allocation and policy-formulation problems because the work in this chapter extends their work to also solve the problem of optimally decomposing a mission into phases and the problem of sequentially allocating resources.

Their work is under the same loosely-coupled multi-agent system assumption as in this work, and their algorithm is presented below in Eq. 3.1, where the constant parameters $p^m_{i,a,j}$, $r^m_{i,a}$, and α^m_j, respectively, represent the state transition probability function, the reward function, and the initial probability distribution of agent m. The constant parameter $u^m_{o,a,i}$ indicates whether agent m requires resource o to execute action a in its state i. The continuous variable $x^m_{i,a}$ represents the expected number of times agent m executes action a in its state i, and the binary variable Δ^m_o represents whether one unit of resource o is assigned to agent m prior to execution.

$$\max \sum_m \sum_i \sum_a x^m_{i,a} \times r^m_{i,a} \tag{3.1}$$

subject to:

probability conservation constraints:

$$\sum_a x^m_{j,a} = \alpha^m_j + \sum_i \sum_a p^m_{i,a,j} \times x^m_{i,a} \qquad : \forall m, \forall j$$

$$x^m_{i,a} \geq 0 \qquad : \forall m, \forall i, \forall a$$

resource constraints:

$$\frac{\sum_i \sum_a u^m_{o,a,i} \times x^m_{i,a}}{X} \leq \Delta^m_o \qquad : \forall m, \forall o$$

$$\sum_m \Delta^m_o = \hat{\omega}_o \qquad : \forall o$$

$$\Delta^m_o \in \{0,1\} \qquad : \forall m, \forall o$$

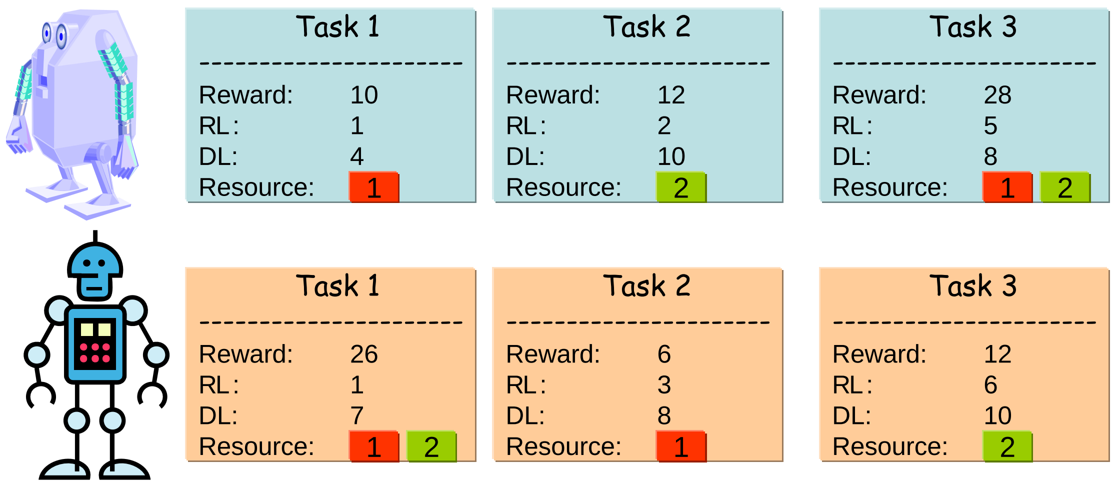
- The objective function $\sum_m \sum_i \sum_a x^m_{i,a} \times r^m_{i,a}$ represents the sum of cumulative rewards among all agents, based upon the assumption that the agents are loosely coupled (i.e., actions taken by one agent will not impact other agents' rewards and transitions).

- The constraint $\sum_a x^m_{j,a} = \alpha^m_j + \sum_i \sum_a p^m_{i,a,j} \times x^m_{i,a}$ guarantees probability conservation at every state for every agent, which is a multi-agent version of the probability conservation constraint in the single-agent MDP formulation (Eq. 2.1) described in the previous chapter.

- X is a constant equal to or greater than sup $\sum_i \sum_a x^m_{i,a}$ (where in a finite horizon MDP, X can be set to the finite horizon T since each agent can only execute T actions within that horizon). The constraint $\frac{\sum_i \sum_a u^m_{o,a,i} \times x^m_{i,a}}{X} \leq \Delta^m_o$ implies that $x^m_{i,a}$ must be zero (i.e., action a cannot be executed by agent m in state i) when $u^m_{o,a,i} = 1$ (i.e., agent m must have resource o to execute action a in its state i) and $\Delta^m_o = 0$. $x^m_{i,a}$ is unrestricted otherwise since X is no less than $\sum_i \sum_a u^m_{o,a,i} \times x^m_{i,a}$ by definition.

- The constraint $\sum_m \Delta^m_o = \hat{\omega}_o$ guarantees that the total amount of resource o allocated across all agents must equal the amount of available resource o (assuming the resources will be completely assigned). This constraint can be easily relaxed to the constraint $\sum_m \Delta^m_o \leq \hat{\omega}_o$ by introducing an additional *dummy* agent to keep unallocated resources.

The optimal joint policy can be easily derived from the solution to the above MILP in a similar way to that discussed in Section 2.2. That is, to maximize the total expected reward of the group of agents, agent m should choose a with probability $\pi^m_{i,a} = \frac{x^m_{i,a}}{\sum_a x^m_{i,a}}$ when it is in its state i.



	Task 1	Task 2	Task 3
Reward:	10	12	28
RL :	1	2	5
DL:	4	10	8
Resource:	1	2	1 2

	Task 1	Task 2	Task 3
Reward:	26	6	12
RL :	1	3	6
DL:	7	8	10
Resource:	1 2	1	2

Figure 3.1: A simple two-agent example.

3.2.1 A Multi-Agent Example

This subsection describes a simple multi-agent resource-allocation example problem, illustrating the above solution approach and examining the impact of resource limitations on system performance. This example problem will also be used later to illustrate the improvement in the system performance using the mission-phasing techniques presented in this chapter.[7]

In this example problem, two cooperative agents attempt to maximize their total expected reward within ten time steps. Each agent has three tasks. At each time step, an agent can choose to continue its previously started task (if there is one and if the required resources are still assigned to that agent), to start a new task (and abort its current task if there is one), or simply to do nothing. In addition, we say that a task that has been aborted previously (and thus has failed) can be re-tried, but no task can be accomplished more than once.

Figure 3.1 shows the detailed information of the tasks in the example problem, including release (RL) time (i.e., when the task becomes available), deadline (DL)

[7]A complete evaluation of our techniques (in both improving the utility and reducing the computational cost) will be presented in Section 3.6.

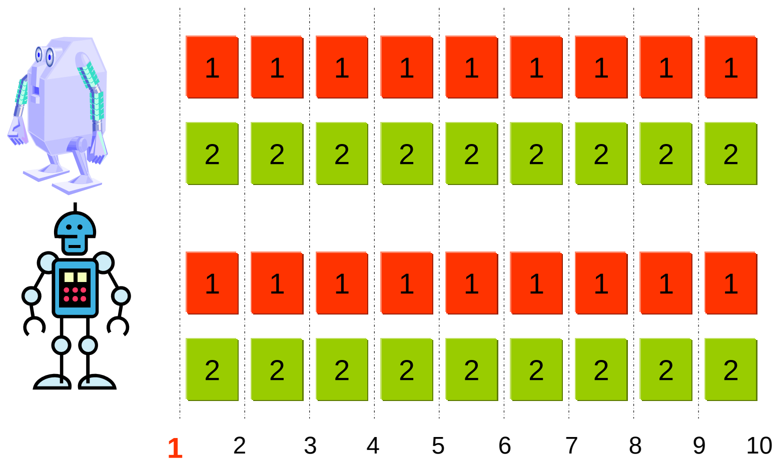


Figure 3.2: Optimal resource allocation when resources are unlimited.

(i.e., when the task becomes unavailable), reward, and resource prerequisite. For example, agent *1* can start (or continue) its task *1*, which will incur a reward 10 if accomplished, at any time step within the interval [1, 4) given that it has one unit of resource *1* at that time. The uncertainty in this problem is in the amount of time required to execute a task. Here, we say that, if an agent starts a task and does not abort it during execution, then the agent has probability 0.3, 0.4, and 0.3 of accomplishing it within one, two, and three time steps, respectively.

When the resources are sufficient (i.e., each agent has all of its desired resources as illustrated in Figure 3.2), this becomes an unconstrained MDP problem for each agent. Using a standard policy formulation algorithm (e.g., value iteration and policy iteration), we can easily compute the optimal unconstrained policy for each agent, which yields the total expected reward 93.64.

Suppose instead that the resources are scarce, i.e., there is only one unit of resource *1* and one unit of resource *2* in the system at any time step. It is not obvious how to distribute these resources among the two agents. Adopting Eq. 3.1, we can find that the optimal one-shot allocation is to give all resources to agent *1* and let agent *2*

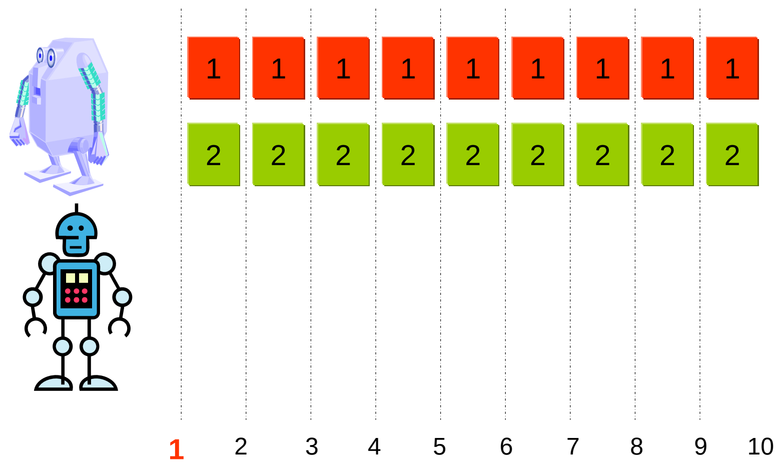


Figure 3.3: Optimal one-shot resource allocation when resources are scarce.

idle over the entire execution, as shown in Figure 3.3, and the total expected reward is 49.64 in this case, much lower than the reward (93.64) in the above unconstrained case. In the following sections, we will use this example as we go along to see the degree to which sequential allocation of the resources can improve the reward in this problem world with the limited shared resources.

3.3 Computational Complexity Analysis

The M-RMP optimization problem is a challenging decision-making problem, involving three intertwined components: mission decomposition, resource allocation, and policy formulation. This section starts by theoretically analyzing the computational complexity of the M-RMP optimization problem, and then explains why several related prior approaches are not computationally tractable for finding an exact solution to the M-RMP optimization problem.

Theorem III.1. *M-RMP optimization is NP-complete.*

Proof: It is trivial to prove that the M-RMP optimization problem is NP-complete. First, given that its special case — one-shot resource allocation and policy formulation — can be proven to be NP-complete through a reduction from the KNAPSACK problem (Dolgov, 2006), M-RMP optimization is NP-hard.

Second, given a solution to the M-RMP problem, the satisfaction of resource constraints and resource reallocation constraints can be verified in linear time. After that, for each agent, incorporating its policy into its MDP model, the M-RMP optimization problem becomes a Markov chain, which can be solved in polynomial time. That is, M-RMP optimization is in NP.

With both NP and NP-hard, M-RMP optimization is NP-complete. □

Decentralized MDP. As previously discussed in Chapter II, modeling resources into the MDP state representation and formulating resource-reconfiguration activities as actions is one possible way to solve a S-RMP optimization problem (although it is much slower than our MILP-based algorithms). However, the same idea is generally inapplicable to the M-RMP optimization problem, because even a small problem will result in a computationally intractable problem of solving a decentralized MDP (Dec-MDP) (Bernstein et al., 2000; Goldman and Zilberstein, 2004).

Note that the outcomes of a resource-reconfiguration action performed by one agent depend on whether interacting agents will perform corresponding resource-reconfiguration actions at the same time. Typically, a joint action of reconfiguring resource assignments among a group of agents can succeed only when the amount of released resources is equal to or greater than the amount of requested resources among all participant agents, which means that the resulting Dec-MDP is not transition independent. A general Dec-MDP is NEXP-complete (Bernstein et al., 2000), and so the decision-making agent may face a NEXP-complete problem with an exponential-size input (exponential in the size of the resource set) when using the idea of incorporating resources in the state representation. In general, this is an extremely difficult problem.

Combinatorial optimization and stochastic optimization. Each phase in a M-RMP problem is a one-shot resource-allocation and policy-formulation problem. However, directly using the prior integrated combinatorial optimization and stochastic optimization approach (Dolgov and Durfee, 2005, 2006) (recapped in Section 3.2) to compute resource allocations and executable policies for each phase independently and then piecing these phase policies together to obtain an overall policy is, in general, not a feasible solution approach for the M-RMP optimization problem. The underlying reason (besides the reason that it has to enumerate all possible ways of decomposing the problem) is that the MILP formulation in Eq. 3.1 requires the initial probability distribution α_j^m to be known *a priori*. However, α_j^m of a phase generally depends on the policy of its previous phase, but the policy of the previous phase usually can only be optimized with respect to the current and future phases.

Auction-based resource allocation. The last prior solution technique discussed in this section is based upon auction-based resource allocation techniques (Pekec and Rothkopf, 2003; de Vries and Vohra, 2003). In brief, each agent submits a set of valuations over its possible sequential resource assignments, which are often called *bids*, to a central decision-making agent.[8] The central decision-making agent then decides how to sequentially allocate resources among those agents.

This is a feasible solution for a simple M-RMP optimization problem, but it will

[8]Based upon the cooperative-agent assumption, it does not matter whether the central agent is in the group of agents that share resources to achieve goals, or it is out of the group.

quickly become intractable as the problem size increases. Let us illustrate this through an example. A group of $m = 5$ agents want to maximize their total expected reward within $t = 10$ time steps, there are $o = 5$ different types of resources in the environment, and resource assignments can be (re)configured $k = 3$ times (i.e., one initial allocation prior to execution and two reallocations in the midst of execution). In such a case, each agent needs to solve $C_{k-1}^{t-1} \times (2)^{o \times k} = 1,179,648$ non-trivial sequential decision-making problems to evaluate all possible sequential resource assignments. After that, the central agent needs to solve a winner determination problem (WDP) where each of five agents submits $1,179,648$ bids. Although, WDP is relatively simple when the participant agents are cooperative (instead of being self-interested) and many of the possible combinations of these bids can be pruned without further analysis (e.g., it is unnecessary to consider and evaluate a combination of bids whose resource-reallocation schedules are in conflict), WDP is still computationally challenging when the input size is large.

Unlike much of the prior work that considers each M-RMP component problem in isolation, the work in this chapter develops automated mission-phasing techniques (that will be presented in the next two sections) that can simultaneously solve the coupled problems of mission decomposition, resource allocation, and policy formulation to exploit interactions among them and to reduce computational cost.

3.4 Exploiting a Fixed Resource Reallocation Schedule

In a similar fashion to Chapter II, we begin with a simplified variation of the M-RMP optimization problem where the schedule of reallocating resources is dictated *a priori*, i.e., resource reallocation cost $\psi_t = 0$ if time step t is specified in a predefined schedule, $\psi_t > 0$ otherwise, and the cost limit $\hat{\psi} = 0$.

As was explained in Section 3.3, directly applying the (one-shot) integrated combinatorial optimization and stochastic optimization approach to compute an executable policy for each phase independently and then piecing phase policies together for an overall policy is usually not feasible. This section presents an alternative solution approach to address this issue. In brief, rather than dealing with each phase independently, we can link those phases together through modeling transition probability conservation. The detail is shown in the following MILP:

$$\max \sum_m \sum_i \sum_a x_{i,a}^m \times r_{i,a}^m \tag{3.2}$$

subject to:

probability conservation constraints:

$$\sum_a x_{j,a}^m = \alpha_j^m + \sum_i \sum_a p_{i,a,j}^m \times x_{i,a}^m \qquad : \forall m, \forall j$$

$$x_{i,a}^m \geq 0 \qquad : \forall m, \forall i, \forall a$$

resource constraints:

$$\frac{\sum_{i \in \mathbb{S}^k} \sum_a u_{o,a,i}^m \times x_{i,a}^m}{T} \leq \Delta_o^{m,k} \qquad : \forall k, \forall m, \forall o$$

$$\sum_m \Delta_o^{m,k} = \hat{\omega}_o \qquad : \forall o, \forall k$$

$$\Delta_o^{m,k} \in \{0, 1\} \qquad : \forall k, \forall m, \forall o$$

where transition probability $p_{i,a,j}^m$, reward $r_{i,a}^m$, initial probability distribution α_j^m (at the beginning of the execution), occupation measure $x_{i,a}^m$, resource prerequisite $u_{o,a,i}^m$, resource limit $\hat{\omega}_o$, finite horizon T, and the objective function $\sum_m \sum_i \sum_a x_{i,a}^m \times r_{i,a}^m$ are the same as in Eq. 3.1. New binary variables $\Delta_o^{m,k}$ indicate whether agent m is assigned one unit of resource o within phase k. The constraint $\sum_m \Delta_o^{m,k} = \hat{\omega}_o$ says that, for any resource type o within any phase k, the amount of the allocated resources must equal the amount of the available resources.

An agent can leave a phase and enter another phase as time passes, but, in a global view, the total expected number of times that agent m visits any state j must equal the probability that agent m is initially at state j plus the total expected number of times that agent m enters state j via all possible transitions. That is, the same constraint $\sum_a x^m_{j,a} = \alpha^m_j + \sum_i \sum_a p^m_{i,a,j} \times x^m_{i,a}$ as in Eq. 3.1 can be used to model probability conservation. On the other hand, a phase transition, which is triggered by a resource reallocation, might change the set of executable actions (in particular states). To model the characteristic that the set of executable actions might differ in different phases, we use the binary variable $\Delta^{m,k}_o$ to indicate whether agent m has resource o within phase k, and link $x^m_{i,a}$ and $\Delta^{m,k}_o$ with the constraint $\frac{\sum_{i \in \mathbb{S}^k} \sum_a u^m_{o,a,i} \times x^m_{i,a}}{T} \leq \Delta^{m,k}_o$ (where $\mathbb{S}^k$ represents the set of states within phase k). That is, $x^m_{i,a} \equiv 0$ (i.e., action a is not executable in state i by agent m within phase k) if $u^m_{o,a,i} = 1$ (i.e., requiring resource o) and $\Delta^{m,k}_o = 0$ (i.e., not having resource o) for any resource o. Otherwise, $x^m_{i,a}$ is not restricted since at most T actions can be executed during one execution with time horizon T.

Deriving an optimal sequential resource allocation and a joint policy from the solution to Eq. 3.2 is straightforward. At the start time of phase k, resources are redistributed in the following way: if $\Delta^{m,k}_o = 1$, then one unit of resource o is assigned to agent m. Every agent m should adopt its policy $\pi^m_{i,a} = \frac{x^m_{i,a}}{\sum_a x^m_{i,a}}$ for maximizing the total expected reward of the group of agents.

Running Example

We now return to the example presented in Section 3.2.1 to illustrate how the total expected reward can be improved when the resources can be reallocated in the midst of execution. Let us say that the resources can be redistributed at times *1*,

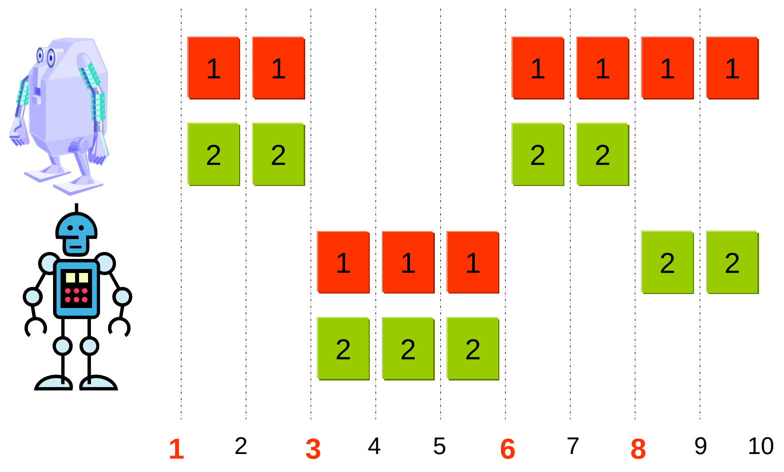


Figure 3.4: Optimal sequential resource allocation for four predefined phases.

3, *6*, and *8*; this resource-reallocation schedule decomposes the example problem horizon into four phases of roughly equal size. Formulating and solving this M-RMP problem with Eq. 3.2 yields the sequential allocation depicted in Figure 3.4. We can see that the resources are managed in a smarter way, where agent *2* no longer idles over the entire execution. As a result, the total expected reward increases to 65.04, 31% higher than that of using the one-shot resource allocation. However, it should be noted that, although evenly decomposing a problem into phases might be a good strategy in some situations, we might be able to do better by optimizing the reallocation schedule, which will be discussed in the next section.

3.5 Determining an Optimal Resource Reallocation Schedule

3.5.1 Solution Algorithm

This section investigates the general M-RMP optimization problem where there is no pre-determined schedule to reallocate resources, and so agents have to determine for themselves when to reconfigure their resource assignments for achieving their remaining goals better. As was defined in Section 3.1, given the inputs $\mathcal{M}$, α, $\mathcal{C}$, and $\mathcal{R}$, the objectives of the M-RMP optimization problem are to find an optimal resource reallocation schedule (subject to the resource reallocation constraint $\mathcal{R}$), which decomposes the overall problem into multiple phases, and to find the optimal resource allocation among agents (subject to the resource constraint $\mathcal{C}$) within each phase, as well as to derive optimal executable phase policies for each agent.

Obviously, a straightforward approach to the M-RMP optimization problem is to enumerate all possible schemes of decomposing a problem into phases, and, for each scheme, adopt the solution algorithm presented in Section 3.4 to determine the best solution. However, when the number of possible decompositions is large, this straightforward approach may become impractical. This section presents an alternative solution approach that extends the MILP formulation in Eq. 3.2 to also include the decision-making about problem decomposition.

The extended MILP is shown in Eq. 3.3, where the objective function and probability conservation constraints are the same as Eq. 3.2. The distinction is that, in order to characterize the limitations on resource reallocation cost and the occurrences of resource-reallocation events, this new formulation represents the resource constraints at each time step (instead of at each phase), and puts in supplementary constraints to model phase transitions.

$$\max \sum_m \sum_i \sum_a x^m_{i,a} \times r^m_{i,a} \tag{3.3}$$

subject to:

probability conservation constraints:

$$\sum_a x^m_{j,a} = \alpha^m_j + \sum_i \sum_a p^m_{i,a,j} \times x^m_{i,a} \qquad : \forall m, \forall j$$

$$x^m_{i,a} \geq 0 \qquad : \forall m, \forall i, \forall a$$

(additional constraints on the next page)

resource constraints:

$$\sum_{i \in \mathbb{S}^t} \sum_{a} u^m_{o,a,i} \times x^m_{i,a} \leq \Delta^{m,t}_o \qquad : \forall t, \forall m, \forall o$$

$$\sum_{m} \Delta^{m,t}_o = \hat{\omega}_o \qquad : \forall o, \forall t$$

$$\Delta^{m,t}_o \in \{0,1\} \qquad : \forall t, \forall m, \forall o$$

reallocation constraints:

$$\Delta^{m,t}_o - \Delta^{m,t-1}_o \leq \Psi_t \qquad : \forall o, \forall t > 1, \forall m$$

$$\Psi_{t=1} = 1$$

$$\sum_{t} \psi_t \times \Psi_t \leq \hat{\psi}$$

$$\Psi_t \in \{0,1\} \qquad : \forall t$$

where $p^m_{i,a,j}$, $r^m_{i,a}$, α^m_j, $x^m_{i,a}$, $u^m_{o,a,i}$, $\hat{\omega}_o$, and T have the same definitions as before. $\mathbb{S}_t$ represents the set of states associated with time t. New binary variable $\Delta^{m,t}_o$ indicates whether resource o is assigned to agent m at time t. The second portion of the Eq. 3.3's constraints (i.e., resource constraints), based upon $\Delta^{m,t}_o$, guarantees that the total amount of allocated resources must equal the total amount of available resources at any time point.

To model the cost limit in resource reallocation, this approach introduces new binary variable Ψ_t to symbolize whether the resources are to be redistributed at time t. To sidestep the question of what the default resource allocation might be, it is assumed that the resources are always initially allocated at the beginning of the execution, i.e., $\Psi_{t=1} = 1$. Note that $\Delta^{m,t}_o - \Delta^{m,t-1}_o$ can never be greater than one since $\Delta^{m,t}_o$ and $\Delta^{m,t-1}_o$ are binary values in $\{0,1\}$; the constraint $\Delta^{m,t}_o - \Delta^{m,t-1}_o \leq \Psi_t$ thus points out that Ψ_t must be one if any agent m procures any extra resource at time t compared to time $t-1$. In other words, any resource reassignment at time t

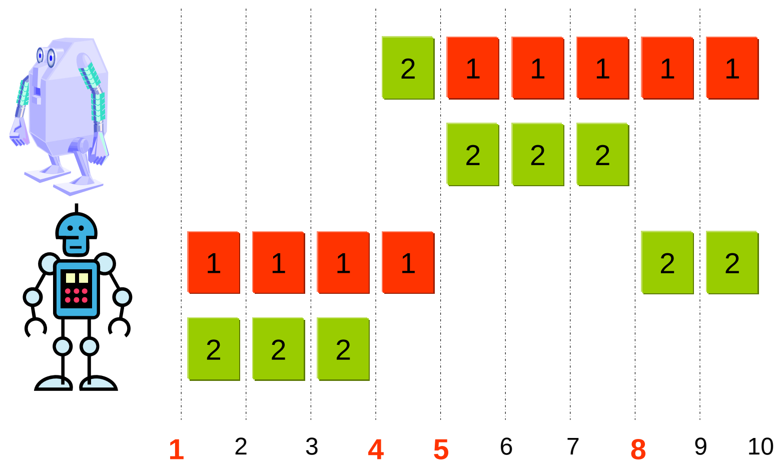


Figure 3.5: Optimal sequential resource allocation for four phases without a predefined schedule.

will lead to $\Psi_t = 1$, which means that we can use the constraint $\sum_t \psi_t \times \Psi_t \leq \hat{\psi}$ to limit the total cost for resource reallocation.

Clearly, by definition, there is a one-to-one mapping between possible sequential resource allocations and possible integer solutions. In addition, given a particular sequential resource allocation, the MILP would be reduced to a linear program whose solution space is equivalent to executable policy space (because resource constraints would prune inexecutable actions). In other words, the MILP solution space includes the best way of allocating resources together with the best way of utilizing the allocated resources, and so finding an optimal solution to the MILP is equivalent to finding an optimal way of sequentially allocating and utilizing resources.

Running Example

Recall that a fixed set of reallocation times $\{1, 3, 6, 8\}$ are chosen in Section 3.4, which results in a reward of 65.04. Now, let us say that the agents will determine for themselves a set of reallocation times given an upper bound of four for the size of this set, i.e., $\psi_{t=1} = 0$, $\psi_{t \neq 1} = 1$, and $\hat{\psi} = 3$. Using Eq. 3.3, the optimal schedule to reallocate resources is computed as $\{1, 4, 5, 8\}$. Figure 3.5 depicts the detailed allocation. We can see that this schedule is more sophisticated; it gives high priority and allots sufficient time for agents to accomplish their high-reward tasks (i.e., task *3* of agent *1*, and task *1* of agent *2*). As a result, the total expected reward for those two agents increases to 72.25, which is 11.1% higher than the simple heuristic of evenly decomposing the problem into phases, and 45.5% higher than for not taking resource reallocations into account.

3.5.2 Variation: Maximizing the Total Reward, Accounting for Cost

In a similar layout to Chapter II, this subsection extends the MILP-based algorithm to a variation of the M-RMP optimization problem where neither the resource-reallocation schedule is predefined (Section 3.4) nor the number of times for reallocating resources is restricted due to the bounded cost of creating phases (Section 3.5.1). Instead, it is now assumed that resource reallocation can occur at any time, and as many times as desired, but a cost should be paid for transferring resources among agents and this cost is calibrated with the utility of the MDP policy, and thus the optimization problem is to maximize the total expected reward, accounting for the cost of redistributing resources in the midst of execution.

We begin by examining a binary-cost case where, if a resource reallocation is scheduled at time t, it will charge the group of agents a constant fee ψ_t regardless of what resources and how many of them are redistributed in that reallocation process. In general, coping with such binary reallocation costs is relatively easy because Eq. 3.3 has paved the way to characterize time steps for reconfiguring resource assignments.

Eq. 3.4 shows the solution algorithm to such a sequential resource allocation problem, which is a slight revision of Eq. 3.3, i.e., adopting a new objective function $\sum_m \sum_i \sum_a x^m_{i,a} \times r^m_{i,a} - \sum_t \psi_t \times \Psi_t$, and removing the constraint $\sum_t \psi_t \times \Psi_t \leq \hat{\psi}$ that is no longer applicable since the agents can now reallocate resources as frequently as they desire.

$$\max \sum_m \sum_i \sum_a x^m_{i,a} \times r^m_{i,a} - \sum_t \psi_t \times \Psi_t \tag{3.4}$$

subject to:

probability conservation constraints:

$$\sum_a x^m_{j,a} = \alpha^m_j + \sum_i \sum_a p^m_{i,a,j} \times x^m_{i,a} \qquad : \forall m, \forall j$$

$$x^m_{i,a} \geq 0 \qquad : \forall m, \forall i, \forall a$$

resource constraints:

$$\sum_{i \in \mathbb{S}^t} \sum_a u^m_{o,a,i} \times x^m_{i,a} \leq \Delta^{m,t}_o \qquad : \forall t, \forall m, \forall o$$

$$\sum_m \Delta^{m,t}_o = \hat{\omega}_o \qquad : \forall o, \forall t$$

$$\Delta^{m,t}_o \in \{0, 1\} \qquad : \forall t, \forall m, \forall o$$

cost constraints:

$$\Delta^{m,t}_o - \Delta^{m,t-1}_o \leq \Psi_t \qquad : \forall o, \forall t > 1, \forall m$$

$$\Psi_{t=1} = 1$$

$$\Psi_t \in \{0, 1\} \qquad : \forall t$$

In the following discussion, we consider a more difficult variation of the M-RMP optimization problem where the cost incurred in redistributing resources is determined by the *amount* of resources being transferred among the agents. Since it is assumed that the agents are cooperative, it does not matter which agent pays the resource transfer costs. Without loss of generality, let us say that agent m pays the cost $c^{m,t}_o$ when it obtains one unit of resource o at time t from someone else, and the agent releasing that resource pays no cost.

Similarly to the above, $\Delta_o^{m,t}$ is used to represent whether resource o is currently held by agent m at time t. The cost that agent m should pay for getting resource o at time t can then be represented as $c_o^{m,t} \times \Theta(\Delta_o^{m,t} - \Delta_o^{m,t-1})$ where function $\Theta(z)$ is a piecewise linear function, defined as:

$$\Theta(z) = \begin{cases} z & z > 0 \\ 0 & otherwise \end{cases}$$

This piecewise linear constraint can be equivalently represented using multiple linear constraints by introducing continuous variables $\epsilon_o^{m,t}$. The new MILP formulation is shown below:

$$\max \sum_m \sum_i \sum_a x_{i,a}^m \times r_{i,a}^m - \sum_o \sum_m \sum_t c_o^{m,t} \times \epsilon_o^{m,t} \tag{3.5}$$

subject to:

probability conservation constraints:

$$\sum_a x_{j,a}^m = \alpha_j^m + \sum_i \sum_a p_{i,a,j}^m \times x_{i,a}^m \qquad : \forall m, \forall j$$

$$x_{i,a}^m \geq 0 \qquad : \forall m, \forall i, \forall a$$

resource constraints:

$$\sum_{i \in \mathbb{S}^t} \sum_a u_{o,a,i}^m \times x_{i,a}^m \leq \Delta_o^{m,t} \qquad : \forall t, \forall m, \forall o$$

$$\sum_m \Delta_o^{m,t} = \hat{\omega}_o \qquad : \forall o, \forall t$$

$$\Delta_o^{m,t} \in \{0, 1\} \qquad : \forall t, \forall m, \forall o$$

cost constraints:

$$\epsilon_o^{m,t=1} = \Delta_o^{m,t=1} \qquad : \forall o, \forall m$$

$$\epsilon_o^{m,t} \geq \Delta_o^{m,t} - \Delta_o^{m,t-1} \qquad : \forall o, \forall t > 1, \forall m$$

$$\epsilon_o^{m,t} \geq 0 \qquad : \forall o, \forall t, \forall m$$

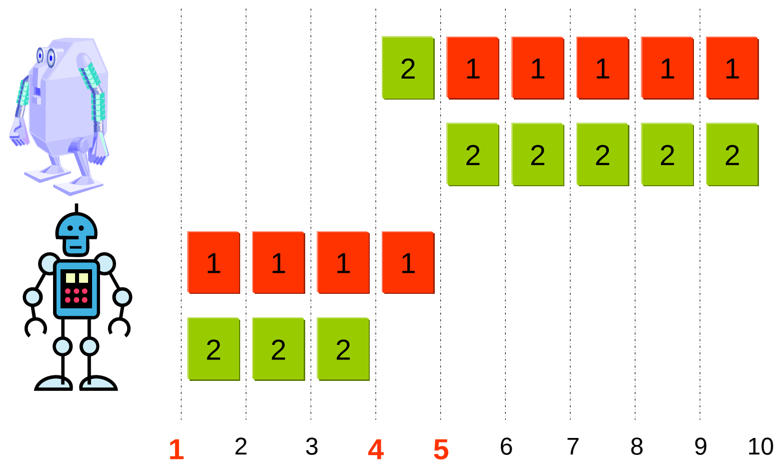


Figure 3.6: Optimal sequential resource allocation, given that the transfer cost is 5 per unit.

That is, when $t > 1$, $\epsilon_o^{m,t}$ is constrained by $\epsilon_o^{m,t} \geq 0$ and $\epsilon_o^{m,t} \geq \Delta_o^{m,t} - \Delta_o^{m,t-1}$. In other words, $\epsilon_o^{m,t} \geq 1$ when $\Delta_o^{m,t} > \Delta_o^{m,t-1}$ (i.e., $\Delta_o^{m,t} = 1$, and $\Delta_o^{m,t-1} = 0$), and $\epsilon_o^{m,t} \geq 0$ under other circumstances. Note that the objective function of Eq. 3.5 is to maximize $\sum_m \sum_i \sum_a x_{i,a}^m \times r_{i,a}^m - \sum_o \sum_m \sum_t c_o^{m,t} \times \epsilon_o^{m,t}$, which implies that the second term $\sum_o \sum_m \sum_t c_o^{m,t} \times \epsilon_o^{m,t}$ should be as small as possible for an optimal solution that yields the highest expected utility. That is, $\epsilon_o^{m,t}$ should reach its lower bound for any optimal solution to Eq. 3.5, i.e., $\epsilon_o^{m,t} = 1$ when $\Delta_o^{m,t} > \Delta_o^{m,t-1}$ (i.e., when agent m acquires resource o at time t) and $\epsilon_o^{m,t} = 0$ otherwise, which exactly matches our expectation of using $\epsilon_o^{m,t}$ to represent the piecewise linear cost function $\Theta(\Delta_o^{m,t} - \Delta_o^{m,t-1})$.

Running Example

We now revisit our running example to illustrate how the above algorithm manages the transfer of resources among the agents when considering the transfer cost. Not surprisingly, as the transfer cost increases, the amount of resources to be transferred decreases. As an example, when the cost of transferring one unit of any resource is 5, the optimal sequential resource allocation, which is shown in Figure 3.6, is to

Case	Resource Allocation Schedule	Utility
non-phasing, Section 3.2.1	transfer 2 resources at T_1	39.64
4 fixed phases, Section 3.4	transfer 2 resources at T_1 transfer 2 resources at T_3 transfer 2 resources at T_6 transfer 1 resource at T_8	30.04
three additional phases, Section 3.5.1	transfer 2 resources at T_1 transfer 1 resource at T_4 transfer 1 resource at T_5 transfer 1 resource at T_8	47.25
unlimited phases, accounting for cost, Section 3.5.2	transfer 2 resources at T_1 transfer 1 resource at T_4 transfer 1 resource at T_5	48.72

Figure 3.7: Comparison of the resource reallocation schedules to the example problem, given that the transfer cost is 5 per unit.

transfer only four units of resources over the entire execution (with two units at the initial time *1*, one unit at time *4*, and one unit at time *5*).

Figure 3.7 shows and compares this schedule and the schedules depicted in the previous sections. As expected, the algorithm in Eq. 3.5 yields a reallocation schedule with the highest utility, i.e., a utility of 48.72.

3.6 Experimental Evaluation

The computational complexity of the M-RMP optimization problem has been theoretically analyzed in Section 3.3. This section is intended to empirically evaluate the effectiveness and computational efficiency of the MILP-based solution algorithms presented in this chapter, using a grid world environment similar to that used in the previous chapter.[9]

[9]An empirical evaluation in the domain with problems similar to (but more complex than) the example shown in Figure 3.1 can be found in our published paper (Wu and Durfee, 2007a).

3.6.1 Experimental Setup

Each test problem instance includes m cooperative agents where each agent operates in its own $n \times n$ grid world that is independent of all others. The starting location of each agent is always at the center of its grid world. The objective of the group of agents is to maximize their total expected reward within T time steps. In a similar fashion to single-agent test problems used in Section 2.6, where a grid world is generated, 40% of the locations are randomly chosen as wall locations, and 10% of the locations are randomly chosen as task locations. The rewards of the tasks are randomly set, i.e., the i^{th} task (in a random order) is associated with a reward i.

The tasks are temporally constrained by their release times and deadlines. The release time of a task indicates the time step when the task becomes available, i.e., attempting the task before its release time will return zero reward. The deadline of a task indicates the time step when the task becomes unavailable, i.e., doing the task after its deadline will also return zero reward. The temporal constraints are randomly set. The release time of a task is an integer uniformly and randomly selected in the range $[1,\ T-2]$ where T is the time horizon, and a task will always expire in three time steps. That is to say, the time window of task i is $[t_i,\ t_i+3)$ where t_i is a random integer in $[1,\ T-2]$. A task can be repeated multiple times (and each time it will give the same reward) within its time window.

The action space of each agent is $\{$*wait*, *up*, *left*, *down*, *right*, *safe-up*, *safe-left*, *safe-down*, *safe-right*, *do*$\}$. All actions except action *do* have the exactly same definitions as before (Section 2.6). The resource prerequisite of the *do* action is also the same as before, but its outcomes are defined in a new way (to make test problems more interesting): after the *do* action is executed, the agent will stay at the same location with probability 0.95 and will be out of the system with probability 0.05. That is to

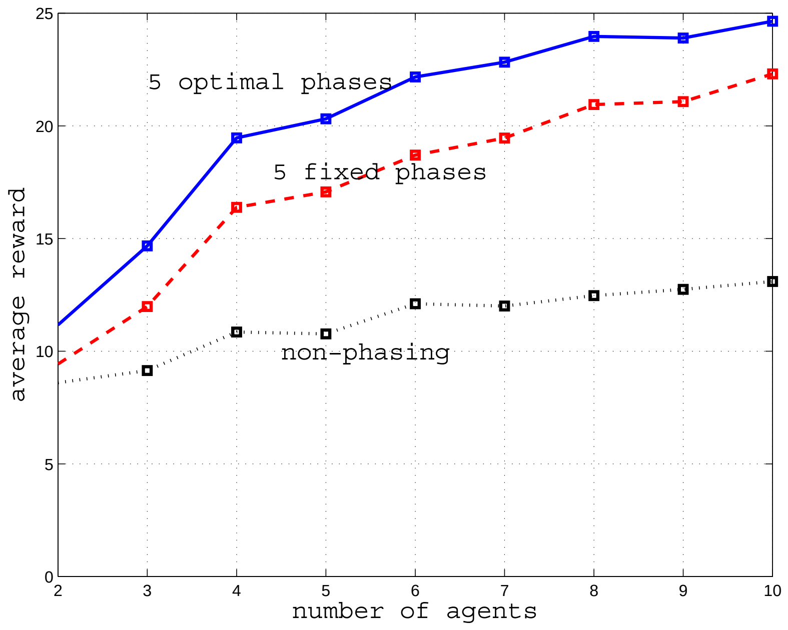


Figure 3.8: Exploiting fixed phases increases the reward, and finding optimal phases further increases the reward.

say, instead of terminating the execution after completing a task, an agent may now stay in the system to do more tasks until the time horizon T is reached.

The system is constrained by resource limitations. There are $|O|$ different types of resources in the system. Each resource type has only one unit, which is shared by m agents.

3.6.2 Optimality

Figure 3.8 demonstrates the improvement of our sequential resource allocation approaches over the prior one-shot resource allocation approach. The x-axis of the figure represents the number of agents in the world, and the y-axis specifies the total expected reward of the group of agents.[10] Other parameters are set as follows: $T = 10$, $n = 5$, and $|O| = 5$. We can see that, by taking into account resource reallocation opportunities in the midst of execution, the agents could gain a considerably higher

[10]In this section, each average data point is computed from 20 random test problems.

reward. For example, in the case that five fixed resource (re)allocation times (one at the initial time step and the other four randomly and uniformly selected when the test problem is defined) are available in the midst of execution, our mission-phasing approach, using Eq. 3.2 and denoted as *5-fixed-phases*, achieves a reward 50% higher than that of not exploiting resource-reallocation opportunities. We can also see that (as expected) finding and using the optimal resource-allocation and phase-switching time points can further improve the system performance, e.g., the *5-optimal-phases* approach (using Eq. 3.3 and assuming that four additional phase-switching points besides the one at the initial time step can be created under the phase-switching cost limit) achieves a reward about 20% higher than the aforementioned 5-fixed-phases solution.

Another interesting discovery from Figure 3.8 is that the improvement of sequential resource allocation over one-shot resource allocation increases as the number of agents increases. This is because, given that the number of resources is fixed (i.e., a fixed number of five resources), the more agents there are, the scarcer the resources are. That is to say, assigning a resource to the right agent at the right time becomes increasingly important to the system performance as the constrainedness of the system increases.

Figure 3.9 uses the same parameters as in Figure 3.8 (i.e., $T = 10$, $n = 5$, $m = 5$, and $|O| = 5$), but looks at the problem from the perspective of the phase-switching cost limit. The results show that the reward of the agents can considerably increase by creating phase-switching points in the midst of execution. However, it should be pointed out that, unlike the S-RMP optimization problem, even if the group of agents can reallocate resources among themselves at every time step, they are usually unable to achieve the same reward (which is 37.2 on average in our test problems)

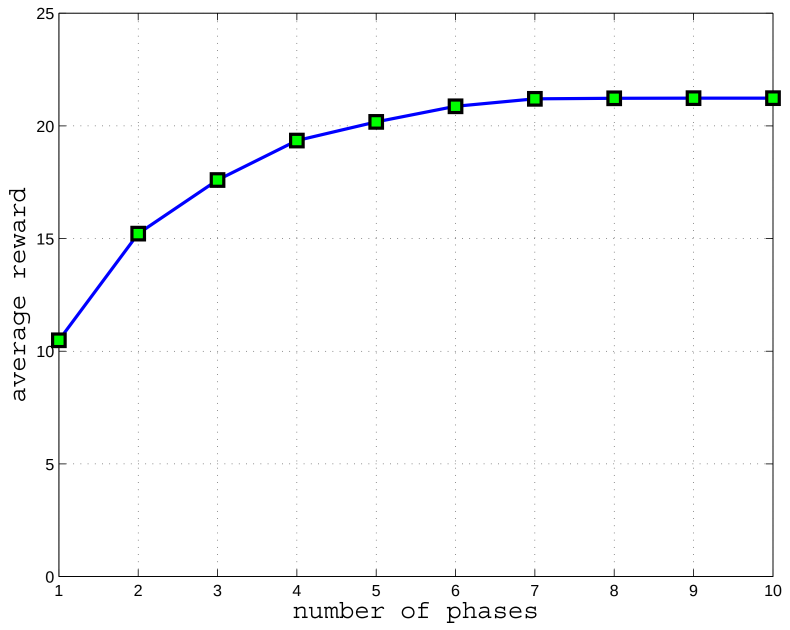
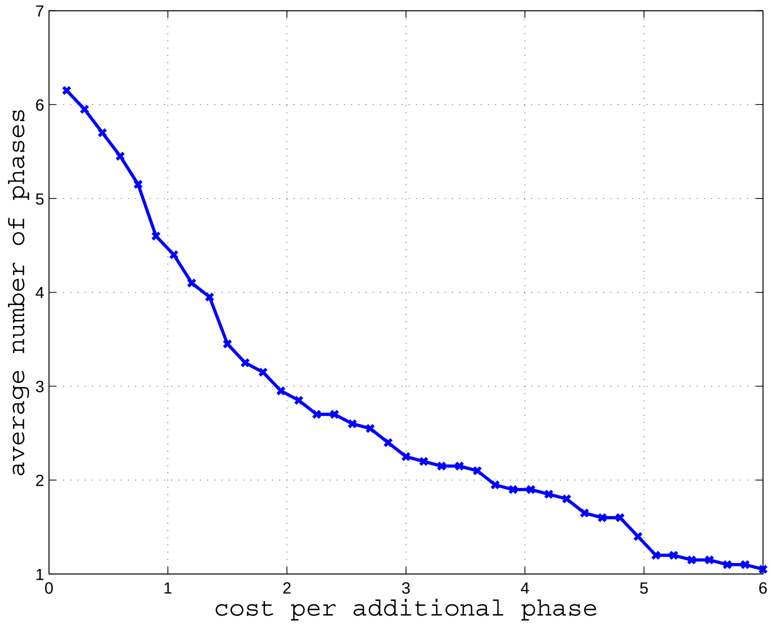


Figure 3.9: The reward increases as the phase-switching cost limit (that determines the number of phases) increases.

as in the unconstrained case with unlimited resources. This is because, at each time step, the joint action space is restricted by the limitation of resources (i.e., assigning a resource to one agent may prevent another agent executing some of its possible actions at that time), though this restriction can change as time passes.

Unlike Figure 3.9 where the phase-switching cost limit is pre-specified to restrict the number of phases, Figure 3.10 examines the M-RMP optimization problem where the phase-switching cost is modeled in the objective function but does not directly limit the number of phases that could be created. As shown, our MILP-based solution approach (using Eq. 3.4) can judiciously determine the number of phases to create, accounting for the cost of creating them. For example, resources are (re)allocated more than six times on average when the cost of creating each additional phase is very low, but the resources are usually only allocated once (at the initial time step) when that cost is high.



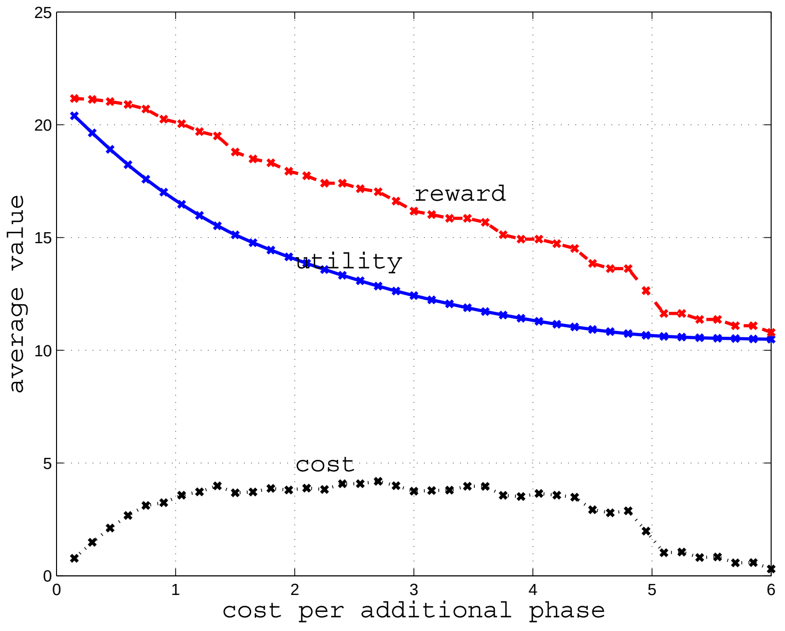


Figure 3.10: The impact of phase-switching cost on phases and utility. Top Figure: the optimal number of phases decreases as the cost of creating each additional phase increases. Bottom Figure: the expected utility decreases as the cost of creating each additional phase increases.

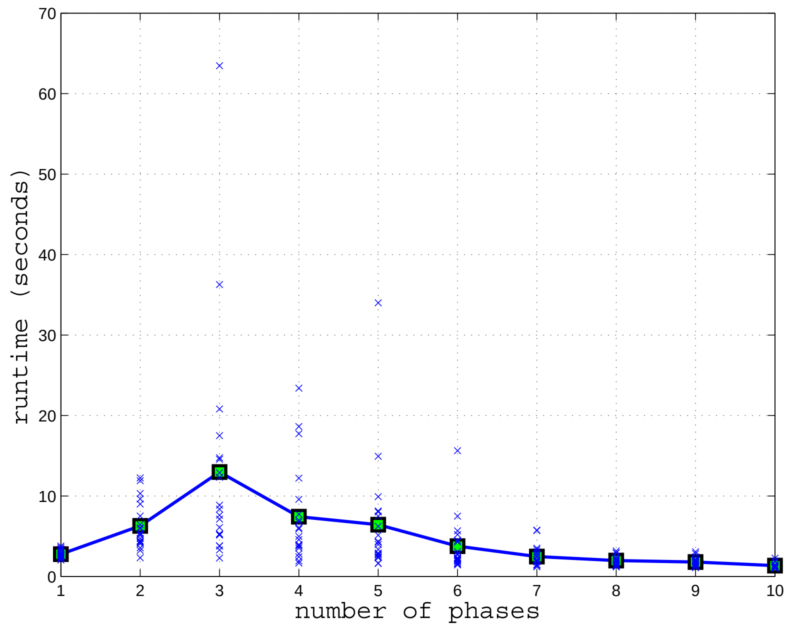


Figure 3.11: The runtime increases and then decreases as the number of times of resource reallocation increases.

3.6.3 Computational Efficiency

To understand the impact of the number of phases on the computational cost and to choose "hard" M-RMP test problems for the following computational efficiency evaluation, we now run experiments with the same parameters as in Figure 3.9, but collect and examine the results of average runtime for finding exact solutions to the test problems. As shown in Figure 3.11, the MILP-based solution approach can exploit the aspects both when the M-RMP optimization problem is over-constrained (when the number of phases is small) and when the M-RMP problem is under-constrained (when the number of phases is large), and reduce computational costs for solving the problems in both cases. According to this complexity profile, the phase-switching cost limit $\hat{\psi}$ is set to 3 in the following experiments[11] (which means that the problem can be decomposed into four phases, under the assumption that

[11]The reason for setting $\hat{\psi} = 3$ instead of $\hat{\psi} = 2$ is that some of test problems in the following evaluation are larger and more complex.

$\psi_{t=1} = 0$ and $\psi_{t\neq1} = 1$), unless we would like to examine the effects of varying the number of phases.

We compare our MILP-based algorithm with the WDP-based algorithm (using the auction-based resource allocation strategy), which is the most computationally-efficient approach among the three prior related approaches discussed in Section 3.3 for solving M-RMP optimization problems. Recall that the WDP-based algorithm involves two steps. First, each agent submits its valuations of its possible sequential resource allotments to a central agent. The number of bids is $C_{K-1}^{T-1} \times (2)^{|O|\times K}$ (as explained in Section 3.3). Second, the central agent solves a winner determination problem. Let us assume that the central agent has a perfect filtering method (although it usually does not), and so it only needs to consider and evaluate "valid" combinations of bids. This assumption reduces the number of possible combinations from $(C_{K-1}^{T-1} \times (2)^{|O|\times K})^m$ to $C_{K-1}^{T-1} \times (m)^{|O|\times K}$ where the reason of using the base m in the exponentiation $(m)^{|O|\times K}$ is that there are m different ways to allocate one resource in the group with m agents.

However, even with this enhancement, the WDP-based algorithm is still computationally intractable for moderately complex M-RMP optimization problems (where it is often unable to find an exact solution even given 100 hours of cpu time). Note that the lower bound of the running time of the WDP-based algorithm can be approximated as $C_{K-1}^{T-1} \times (2)^{|O|\times K} \times t_{bid} + C_{K-1}^{T-1} \times (m)^{|O|\times K} \times t_{eval}$ where t_{bid} is the average runtime of evaluating a sequential resource allotment (i.e., a bid) by modeling and solving an unconstrained finite-horizon MDP, and t_{eval} is the average runtime of evaluating a feasible combination of agents' bids. This work uses a sampling method to estimate the runtime, i.e., t_{bid} and t_{eval} are estimated from 100,000 random runs.

Figure 3.12 shows and compares the runtime results under various parameter

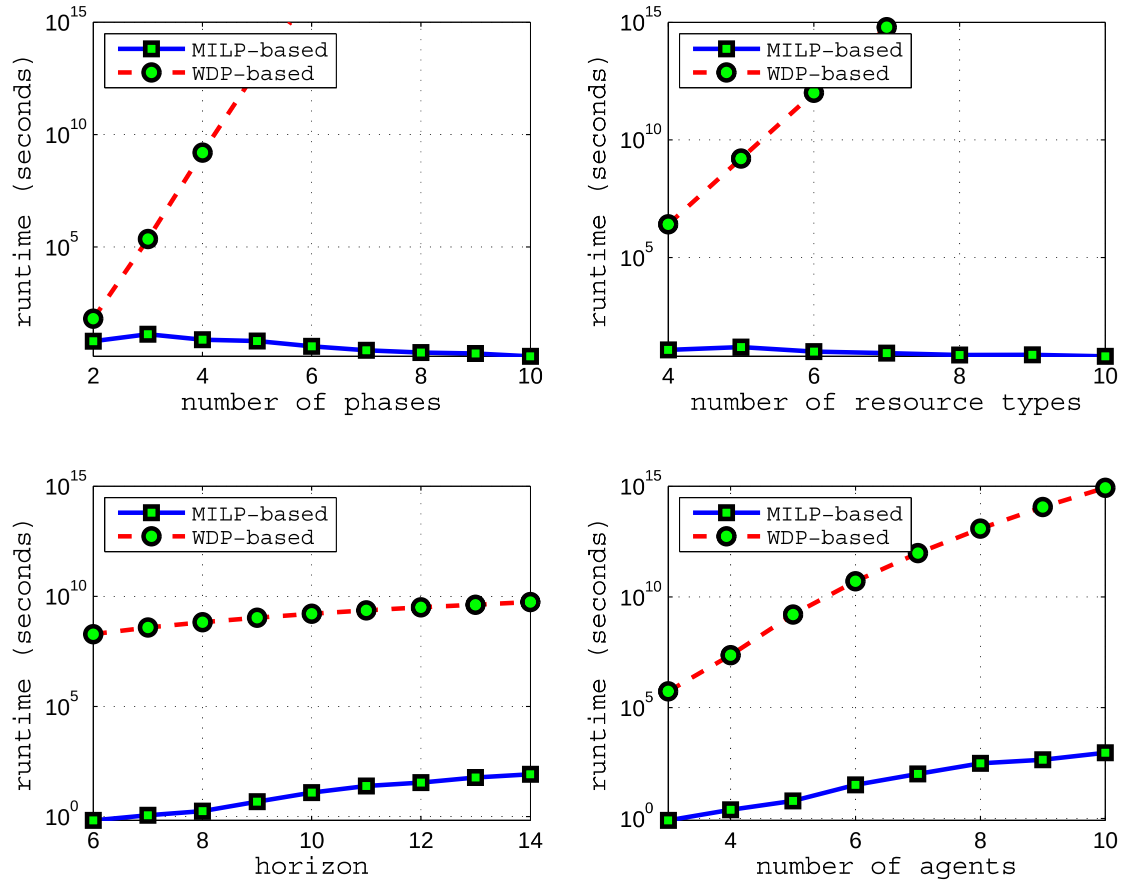


Figure 3.12: Runtime comparison between the MILP-based algorithm and the WDP-based algorithm. Parameters are set as follows: Top-left figure $n = 5$, $T = 10$, $m = 5$, $|O| = 5$, and $\hat{\psi} = \{1, 2, ..., 9\}$. Top-right figure $n = 5$, $T = 10$, $m = 5$, $|O| = \{4, 5, ..., 10\}$, and $\hat{\psi} = 3$. Bottom-left figure $n = 5$, $T = \{6, 7, ..., 14\}$, $m = 5$, $|O| = 5$, and $\hat{\psi} = 3$. Bottom-right figure $n = 5$, $T = 10$, $m = \{3, 4, ..., 10\}$, $|O| = 5$, and $\hat{\psi} = 3$.

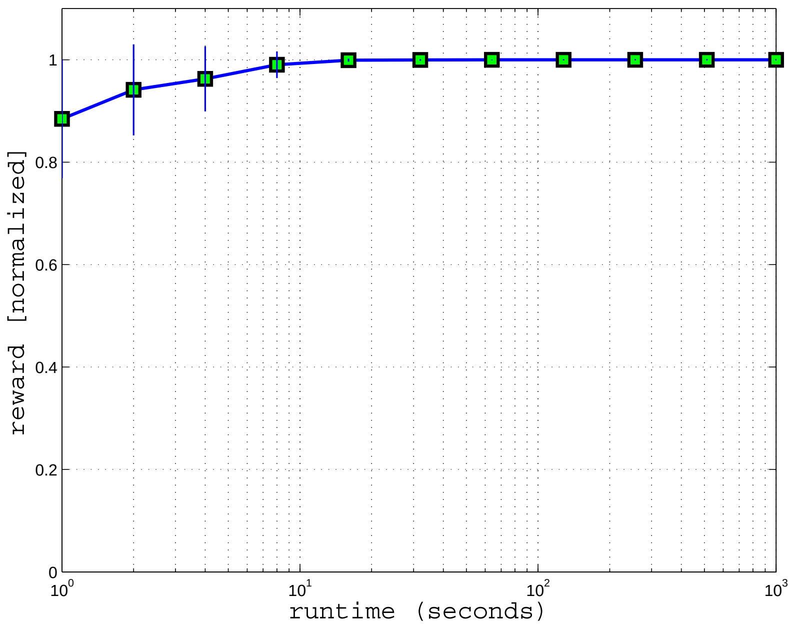
settings.[12] Note that the y-axis is in a logarithmic scale. These results illustrate and emphasize that the MILP-based algorithm, which formulates and simultaneously solves the coupled problems of mission decomposition, resource allocation, and policy formulation using a single compact MILP formulation, can effectively and fruitfully exploit the inter-relationship among these component problems. As a result, it is significantly faster than the WDP-based approach that considers the component problems in isolation.

3.6.4 Anytime Performance

The MILP-based approach presented in this chapter is designed to find an exact solution to the M-RMP optimization problem. Nevertheless, since the approach can adopt an anytime MILP solver for its formulated MILP,[13] it can also serve as an anytime M-RMP solver.

This evaluation section concludes by empirically analyzing anytime performance of the MILP-based algorithm (Eq. 3.3). As shown in Figure 3.13, the anytime performance of our algorithm is (at least) reasonably good, given that it does not depend on any domain-specific heuristic and knowledge. For example, in the simpler test problems with parameters $n = 5$, $T = 10$, $m = 5$, $|O| = 5$, and $\hat{\psi} = 3$, the algorithm finds a near-optimal solution (with above 95% of the optimal reward) within 4 seconds on average, while finding an optimal solution takes about 20 seconds. As another example, in the more complex problems with $n = 6$, $T = 12$, $m = 6$, $|O| = 6$, and $\hat{\psi} = 3$, it takes about 20 seconds to find a near-optimal solution on average, compared to about 650 seconds for finding an exact solution.

[12] Neither MILP nor WDP uses parallel computation.
[13] The *cplex* solver used in this work is an anytime solver that usually has good anytime performance.



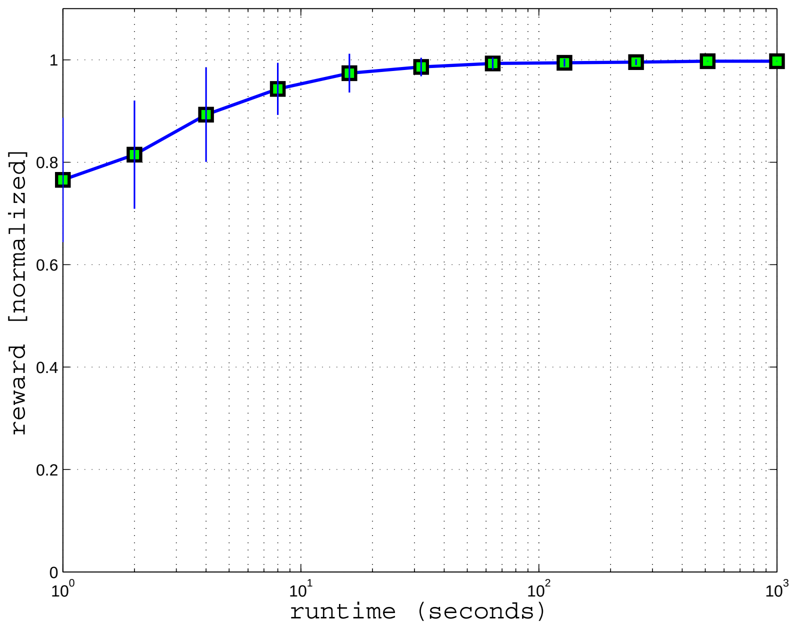


Figure 3.13: Anytime performance of the MILP-based algorithm. Parameters are set as follows. Top figure: $n = 5$, $T = 10$, $m = 5$, $|O| = 5$, and $\hat{\psi} = 3$. Bottom figure: $n = 6$, $T = 12$, $m = 6$, $|O| = 6$, and $\hat{\psi} = 3$.

3.7 Summary

In this chapter, we have presented, analyzed, and empirically evaluated a MILP-based approach that automates the process of finding and using optimal resource reallocation schedules for a group of agents operating in complex environments with resource limitations and with uncertainties. Our analytical and experimental results have shown that the approach can greatly reduce computational cost compared to prior approaches.

The contributions of the work in this chapter are summarized as follows:

- This work extends the prior techniques for solving the one-shot resource-allocation-and-policy-formulation problem to also solve the problem of optimally decomposing the agents' overall activities into a sequence of phases. It generalizes the integrated resource-allocation and policy-formulation approach.

- This work extends our single-agent resource-driven mission-phasing techniques to multi-agent environments. The presented approach can explicitly take into account resource reallocation opportunities in the midst of execution to redistribute resources among agents over time, and can automate the process of finding and using such opportunities in complex constrained and stochastic environments. The experimental results (shown in Figure 3.8 and Figure 3.9) have shown and emphasized that the approach is an effective way to improve the agents' reward in resource-constrained systems.

- Similarly to our S-RMP techniques, the M-RMP techniques presented in this chapter can effectively exploit interactions among the coupled problems of mission decomposition, resource allocation, and policy formulation by representing all of them in a single compact formulation and solving them simultaneously there. The results (shown in Figure 3.12) highlighted the significance of our techniques in reducing computational cost, compared to the approach that considers mission decomposition, resource allocation, and policy formulation in isolation.

CHAPTER IV

Scheduling Phase Decision Procedures

The previous two chapters have examined and illustrated the effectiveness of our automated resource-driven mission-phasing (RMP) techniques in resource-constrained systems. The remainder of this dissertation will focus on another family of constraints — computational time limitations,[1] and present computation-driven mission-phasing (CMP) techniques for improving agent performance in time-critical environments.[2]

4.1 Computation-Driven Mission-Phasing Overview

4.1.1 Introduction

Computational time limitations commonly reside in online application domains. For example, an autonomous aircraft flying a prolonged mission might not have time to prepare a plan that specifies actions and reactions for all possible contingencies over the entire mission before it must start to execute the plan. This raises challenges for finding the best possible solutions within the time limits.

The essential ideas of our computation-driven mission-phasing techniques are presented in Figure 4.1. In a wide variety of research fields, enormous efforts have been devoted to speeding up planning through "divide and conquer" strategies. These

[1]Although some approaches presented in this dissertation are applicable for problems with both resource constraints and computation constraints, a thorough study of such problems is beyond the scope of this dissertation and will be addressed in the future research.

[2]This chapter is largely based on work that was originally reported in (Wu and Durfee, 2006b).

techniques decompose a large complex problem into multiple (approximately) independent pieces and can often gain significant speedup by solving each sub-problem separately and then combining sub-problem solutions together into an overall solution. However, given a large complex online problem, even with a decomposition method that can properly decompose the problem into independent sub-problems (phases), the resulting phases may still be too large to be solved completely under computation bounds.

One potential way to address this issue is to adopt an anytime policy formulation method, building partial policies for each phase. To this end, our CMP work designs a heuristic search algorithm which is prone to explore and expand states that are likely to be reached by following high-quality policies.

Note that, in many application domains, the value and importance of sub-problems in a mission may vary a lot, which suggests that a sophisticated approach should be able to bias its computational efforts on high-value sub-problems. The CMP approach presents an automated deliberation-scheduling algorithm to selectively distribute limited computation time among phases, based upon their predicted contributions to the utility of the overall solution. Furthermore, besides the ability to utilize the available time prior to mission execution, the deliberation scheduling algorithm also explicitly takes into account possible additional computation time in the midst of mission execution. Intuitively, an agent can often do better to focus computation only on near-term high-value phases. Then, while executing the plans for earlier phases, the agent could use available computation time during execution to reconsider aspects of the problem and improve its solutions for the current and future phases.

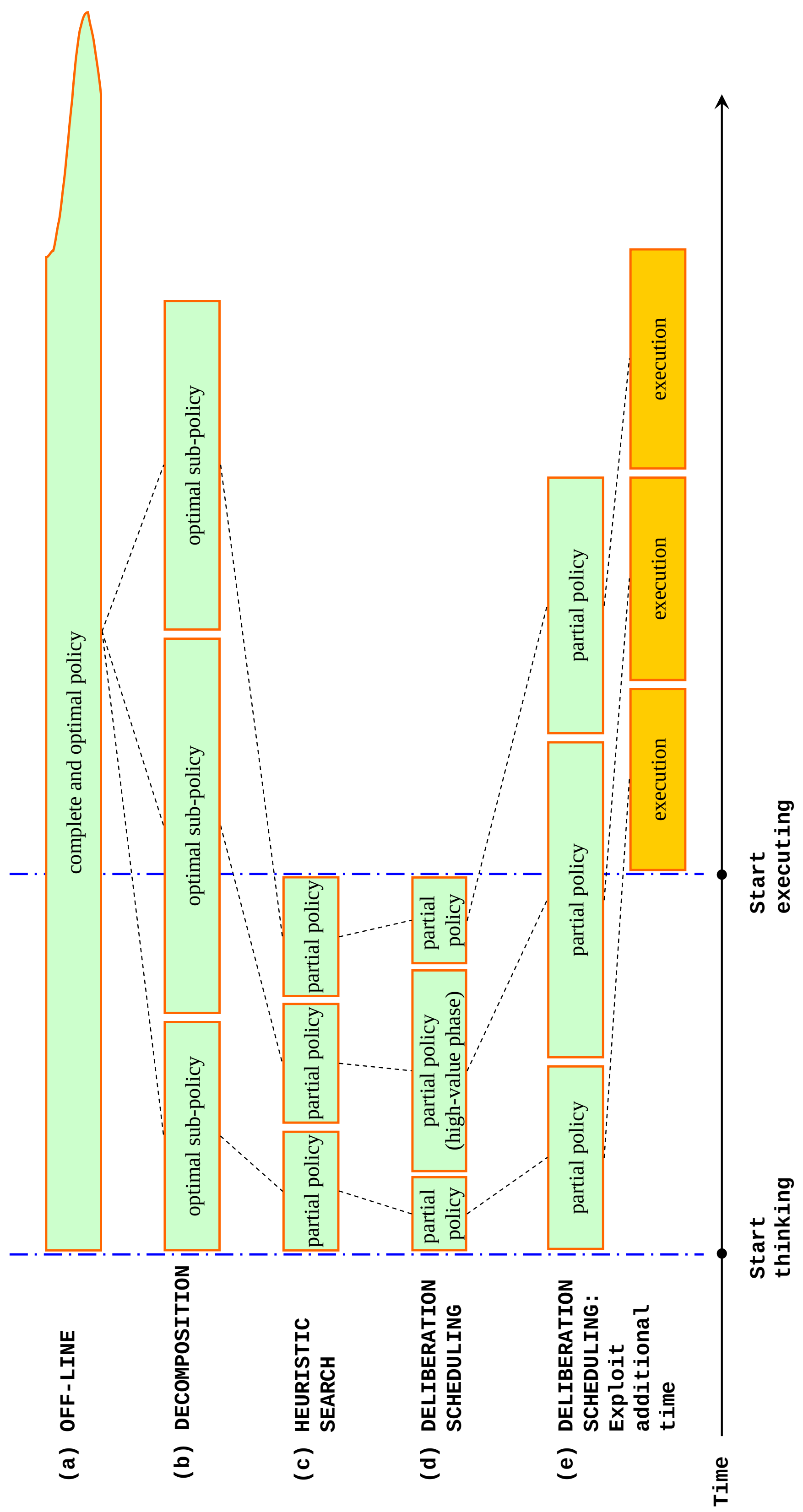


Figure 4.1: Illustration for computation-driven mission-phasing techniques.

4.1.2 CMP vs. RMP

To provide readers with a better understanding of why the RMP approaches previously presented in Chapter II and Chapter III are not applicable for CMP problems, we here discuss the differences between computation-driven mission-phasing techniques and resource-driven mission-phasing techniques. In short, the RMP work developed efficient off-line techniques where phasing is driven by the need to reallocate resources, while the CMP work focuses on online techniques where phasing is driven by the need to focus on high-value sub-problems and the need to exploit possible available computation time in the midst of execution.

Although either of the phasing approaches consists of three component problems (i.e., mission decomposition, resource/time allocation, and policy formulation), the solution techniques differ considerably because of the fundamental distinctions between computational time limitations and resource constraints. Usually, the RMP work does not explicitly consider computation bounds (although computationally efficient algorithms are required), and its objective is to find an exact solution indicating optimal problem decomposition, optimal resource allocations, and optimal executable policies. In contrast, the CMP work is driven by computational time limits, which means that achieving optimality in that way is usually impossible since thinking how to use the limited computation time indeed consumes the time that can otherwise be used for actual problem solving. Therefore, the CMP work in this dissertation faces the following requirements:

- Quickly and properly decompose a large complex mission into multiple (nearly) independent phases
- Quickly and properly distribute the limited computation time among phases

- Effectively utilize the allocated computation time for each phase

4.1.3 Overview

One major reason that planning in environments with uncertainties is difficult is that an agent starts from a known initial state but goes into branching futures. Decomposition techniques can reduce computational cost by finding "known" intermediate goal states and encouraging the agent to work off from these intermediate states. There have been a number of existing decomposition methods using this idea, e.g., the "doorway" decomposition heuristic in robot navigation domains (Parr, 1998; Precup and Sutton, 1998; Lane and Kaelbling, 2001), and the mission decomposition techniques in autonomous aircraft domains (Goldman et al., 2001). These techniques have been shown to be able to significantly reduce the runtime for finding an approximately optimal solution.

On the down side of the decomposition techniques, an incorrect guess of intermediate goal states may cause "confusion" at the transition points between phases (i.e., the agent might be unable to reach the intermediate state it was expected to reach), and result in a negative impact on the agent's performance. Therefore, like much of the prior work using the decomposition strategy, our CMP work emphasizes the problems in which sub-problems are weakly connected so that the intermediate goal states can be determined in a reasonably straightforward manner (by exploiting domain-specific knowledge).[3] Though it is not the focus of this work, we will illustrate the idea of exploiting such weakly-connecting relationships through a realistic application problem in Section 5.5.1 where our CMP techniques are to be evaluated.

The novelty and the main contributions of our CMP work are its deliberation

[3]Nevertheless, as will be presented in the next chapter, the heuristic search component of our CMP approach is often able to quickly find, by itself, a high-quality solution to a large complex problem with strongly interacting sub-problems.

scheduling techniques (for time allocation) and heuristic search techniques (for anytime policy formulation), which will be presented in this and the next chapter, respectively.

The rest of this chapter is organized as follows. Section 4.2 introduces related work in the deliberation scheduling literature. Section 4.3 gives a fundamental definition of the deliberation scheduling problem of interest in this work, followed by its solution algorithm presented in Section 4.4. The solution algorithm is extended to deal with more complex objective functions and non-deterministic phase transitions in Section 4.5 and Section 4.6, respectively. Experimental results are shown in Section 4.7, where the efficiency and optimality of our approach is evaluated. Finally, Section 4.8 summarizes the contributions of our work presented in this chapter.

4.2 Background: Deliberation Scheduling

In the planning research literature, the process of scheduling decision procedures to maximize overall system performance is often called deliberation scheduling (Boddy and Dean, 1989, 1994; Goldman et al., 2001; Horvitz, 2001; Musliner et al., 2005). Deliberation scheduling starts with the premise that an anytime algorithm is able to produce improving plans given increased computation time, and needs to carefully manage the distribution of the computation time among decision procedures in environments where multiple decision procedures share limited computation time.

A fundamental construct in the deliberation scheduling research is the *performance profile*. The performance profile of an anytime algorithm indicates the predicted utility of a solution (derived by that algorithm) as a function of the algorithm's runtime to derive that solution. Typically, the performance profile of an anytime algorithm is learned through applying that algorithm to solve many similar problem instances (similar to the online problem of concern) in an off-line planning stage, and then the results derived at various time points are collected and averaged to predict how much utility an online decision procedure can achieve without actually solving that online problem.

Based upon the construct of performance profiles, several deliberation scheduling methods have been developed to answer “planning when to plan” questions. Boddy and Dean (1989, 1994) proposed an optimal deliberation scheduling method for a particular family of decision procedures with piecewise linear concave performance profiles. Their algorithm works backwards from the occurrence time of the last event, and, at every iteration through the main loop, the algorithm allocates some interval of computation time to the decision procedure that is expected to incur the largest gain. Under the simplifying assumption that performance profiles of all considered decision procedures are strictly concave, this myopic algorithm can guarantee to find an optimal deliberation schedule.

Horvitz (2001) explored policies for proactive allocation of idle time for potential future decision procedures. That work explicitly considered uncertainty (i.e., the probability of the occurrence of future decision procedures), explored several families of performance profiles (but still not as general as the work presented in this chapter), and presented methods to derive ideal policies for guiding pre-computation in several settings.

Goldman et al. (2001) proposed a greedy deliberation scheduling algorithm, which myopically looks one-step ahead along all of its immediate deliberation action choices to find the action that results in a plan with the highest expected utility. They compared the performance of this greedy method with an optimal (but very slow) deliberation scheduling algorithm based upon MDP models, and showed through experiments that their myopic method can often find a fairly good solution within a short time (Goldman et al., 2001; Musliner et al., 2005).

However, these prior approaches are only optimal with respect to limited types of performance profiles of decision procedures, and/or ignore some important aspects of online problems that may be exploited to improve agent performance (e.g., an agent may choose to pay a cost for some additional time to derive a better solution). The work in this chapter is directed at addressing some of those issues.

4.3 Problem Definition

The deliberation scheduling procedure is the core component of our computation-driven mission-phasing techniques. It is based on the premise that an online, complex problem can be decomposed into multiple nearly-independent phases, and its goal is to help a time-limited agent focus its computation on high-value phases as well as help the agent exploit possible additional time in the midst of execution to reconsider system aspects and improve solutions to future phases.

In general, to find a good way for distributing the limited deliberation time among multiple phases, an autonomous agent should have some prior knowledge to predict how much utility a phase decision procedure can achieve, but without actually spending much time solving that decision procedure. The reason is obvious. Once computation time is spent, it is useless to schedule its use. The work in this chapter uses the same assumption as in the prior deliberation scheduling work introduced in Section 4.2 — performance profiles of the phase decision procedures are known *a priori*. A detailed discussion about how to construct and use performance profiles of our test problems is postponed to the CMP evaluation section (Section 5.5.2) in the next chapter.

Clearly, a sophisticated deliberation scheduling approach should not only be able to deal with the cases where the intervals of computation time are pre-specified, but should also be able to address more general cases where an agent has the choice of using as much computation time as it desires but there is a cost associated with using the time. The deliberation costs at different phases may be different. For a Mars rover example, more deliberation before executing the mission will incur a cost of delaying the mission, and more deliberation during execution will incur a cost of distracting the rover from responding to external events. In this work, we refer to the functions that characterize the relationship between the amount of time used by an agent and the cost the agent should pay as *deliberation cost functions*.

With the constructs of the performance profiles and the deliberation cost functions, it is time to formulate the deliberation scheduling problems of interest in this chapter. This section gives the definition of a fundamental deliberation scheduling problem, in which the objective function is linear and phase transitions are deterministic. After laying out our solution algorithm to this problem, the formulation will be extended to represent more complicated nonlinear objective functions (Section 4.5) and non-deterministic phase transitions (section 4.6).

A deliberation scheduling problem is an optimization problem with the inputs $\langle \mathcal{B}, \mathcal{V}, \mathcal{C} \rangle$:

- ☐ Problem $\mathcal{B}$ consists of a sequence of phases $\{phase_1, phase_2, \ldots, phase_n\}$. Once an agent leaves $phase_i$, it will enter $phase_{i+1}$.

- ☐ $\mathcal{V} = \{V_i(t)\}$ define performance profiles where $V_i(t)$ predicts the utility of phase i given the amount of its assigned computation time t.

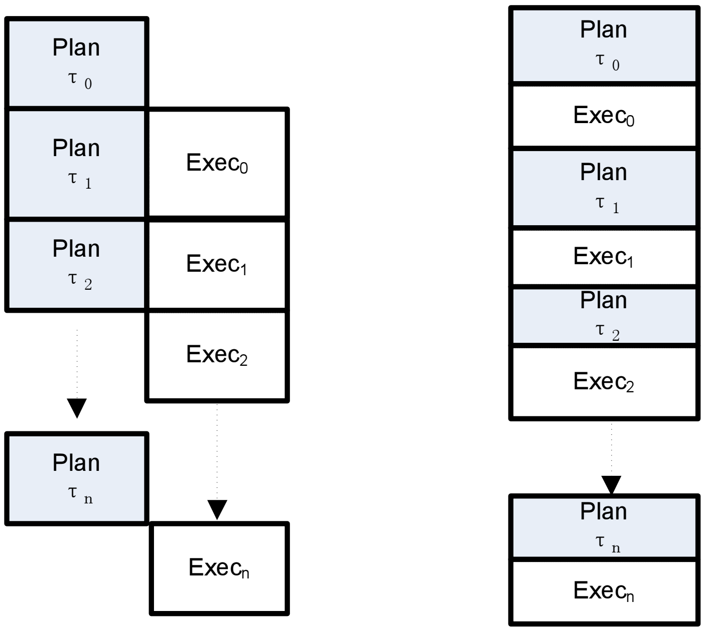


Figure 4.2: Deterministic phase transitions in two situations: simultaneous planning and execution (left) and interleaved planning and execution (right).

□ $\mathcal{C} = \{C_i(\tau)\}$ define deliberation cost functions where $C_i(\tau)$ denotes the cost of using the amount of computation time τ at phase i.

The objective of the problem is to maximize the cumulative utility across all phases by determining a deliberation schedule specifying at which time intervals to "think" about which phases. The solution schedule will help the agent use its limited computation time in a clever way, i.e., spending more time performing policy formulation in high-value phases, balanced by less completed policies in the other phases.

4.4 Solution Algorithm

4.4.1 Nonlinear Formulation

Given that the transitions among phases are deterministic, a mission can be represented as a chain of phases as shown in Figure 4.2. The left side of the figure depicts the case where deliberation and execution can occur simultaneously, such that deliberation about future phases can be done during execution of the current phase. The right side of the figure shows interleaved planning and execution, where an interval for deliberation is followed by an interval for execution, which in turn is followed by time for deliberation, and so on. This chapter describes and illustrates our deliberation scheduling approach in situations where planning and execution interleave. Nonetheless, the presented algorithms are also applicable in the situations where an agent can plan and execute in parallel, assuming that the agent would not revise its phase policy that it is currently executing.

Let τ_0 be the amount of computation time that the autonomous agent initially has when the problem is presented, and τ_i $(i \geq 1)$ be the amount of additional computation time that the agent can have after finishing the execution of the previous phase and before beginning the execution of $phase_i$. Let us temporarily assume that τ_i is fixed and specified *a priori*; this assumption will be relaxed shortly. The deliberation scheduling problem is then to schedule decision procedures within these available computation time intervals so that the expected utility of the mission solution is maximized. A straightforward strategy is to allocate τ_i to $decision_i$ (where $decision_i$ represents the decision procedure for $phase_i$), but this myopic approach is usually suboptimal since it might be fruitful to use some of the time to get a head start on decision procedures for future phases.

Let t_i denote the total amount of computation time scheduled for the decision procedure of $phase_i$. The deliberation scheduling problem with deterministic phase transitions can then be represented in the following mathematical formulation:

$$\max \sum_i v_i \tag{4.1}$$

subject to:

$$\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i \qquad : \forall k$$

$$v_i = V_i(t_i) \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

where $\forall k \in \{0, 1, \ldots, n\} : \sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i$ indicates the fact that the amount of scheduled computation time can never exceed the amount of available computation time at any point. v_i is the expected utility of the solution to $phase_i$, and the objective function $\sum_i v_i$ represents the total expected utility.

If $V_i(t)$ is a linear function of t for any $phase_i$, then the constraints $v_i = V_i(t_i)$ in Eq. 4.1 are trivial linear constraints, and the deliberation scheduling problem can be formulated as a linear program that is solvable in polynomial time. However, for most anytime algorithms (e.g., the RTDP algorithm (Barto et al., 1995), the LAO* algorithm (Hansen and Zilberstein, 2001)), performance profiles $V(t)$ are nonlinear. Nonlinear optimization problems are usually computationally intractable. In the following discussion, we present how to use approximation techniques for linearization.

4.4.2 Linearization

- **Continuous concave performance profile**

  For many anytime algorithms (e.g., the RTDP algorithm), the rate of refinement of the solution slows down with increasing computational activity, which means that $V_i(t)$ is a continuous concave function by definition. It has been well established that a continuous concave function can be approximated as a piecewise

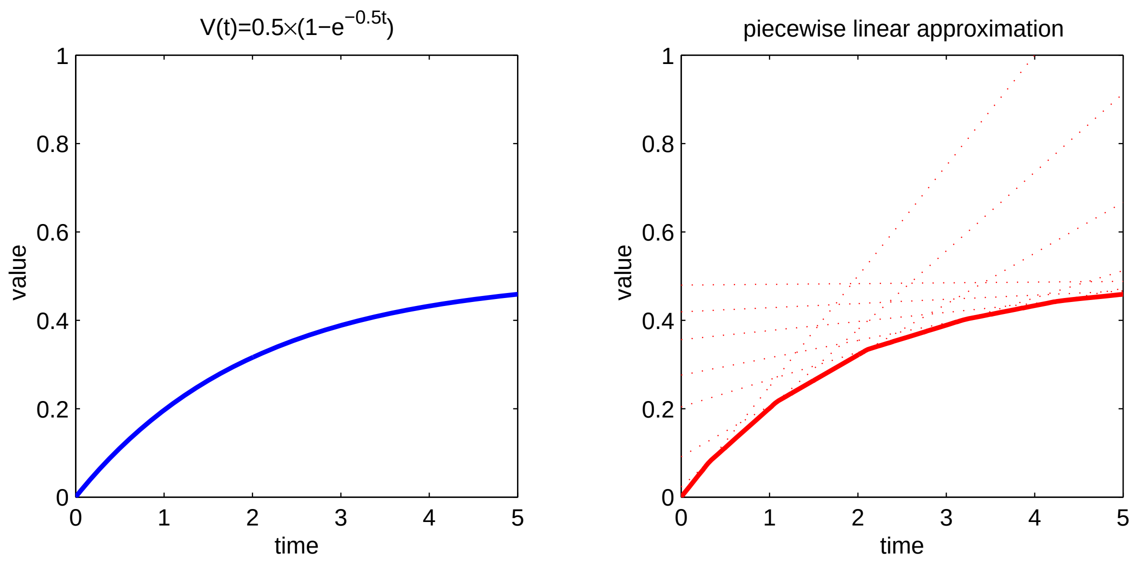


Figure 4.3: A piecewise linear approximation example: a continuous concave function (left), and its piecewise linear approximation (right).

linear concave function (e.g., Powell, 1981). Using a sufficiently large number of pieces, such an approximation usually performs well. Figure 4.3 shows an example of approximating function $V(t) = 0.5 \times (1 - e^{-0.5t})$ with a piecewise linear concave function that is composed of eight linear pieces.

In this work, we adopt a naive but fast algorithm to construct piecewise linear functions, which, at each iteration, myopically adds a linear piece that will most reduce the approximation error (defined as the maximum gap between the input function and its approximation function). Our empirical results show that this myopic algorithm can, in general, approximate a function within several milliseconds, and thus is well suited for online applications.

Let $V_{i,j}(t) = a_{i,j} \times t + b_{i,j}$ be the linear function used to represent the j^{th} segment of the piecewise linear concave curve. Then, the continuous concave function $V_i(t)$ can be approximated as $V_i(t) = \min_j a_{i,j} \times t + b_{i,j}$. In turn, the constraint $v_i = V_i(t_i)$ in Eq. 4.1 becomes $v_i = \min_j a_{i,j} \times t_i + b_{i,j}$, and the deliberation scheduling problem can be formulated into the following program:

$$\max \sum_i v_i \tag{4.2}$$

subject to:

$$\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i \qquad : \forall k$$

$$v_i = \min_j a_{i,j} \times t_i + b_{i,j} \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

Eq. 4.2 is mathematically equivalent to the following linear program (Eq. 4.3) because v_i always reaches its upper bound $\min_j a_{i,j} \times t + b_{i,j}$ when the objective function $\sum_i v_i$ is maximized.

$$\max \sum_i v_i \tag{4.3}$$

subject to:

$$\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i \qquad : \forall k$$

$$v_i \leq a_{i,j} \times t_i + b_{i,j} \qquad : \forall i, \forall j$$

$$t_i \geq 0 \qquad : \forall i$$

A linear program can be solved fast (i.e., in polynomial time), which explains why much prior work has focused on piecewise linear concave performance profiles and there exist fast algorithms (e.g., Boddy and Dean, 1994) that can find optimal deliberation schedules for such problems.

- **General nonlinear performance profile**

  In the previous discussion, we have approximated deliberation scheduling problems that have decreasing return rate performance profiles into linear programs.

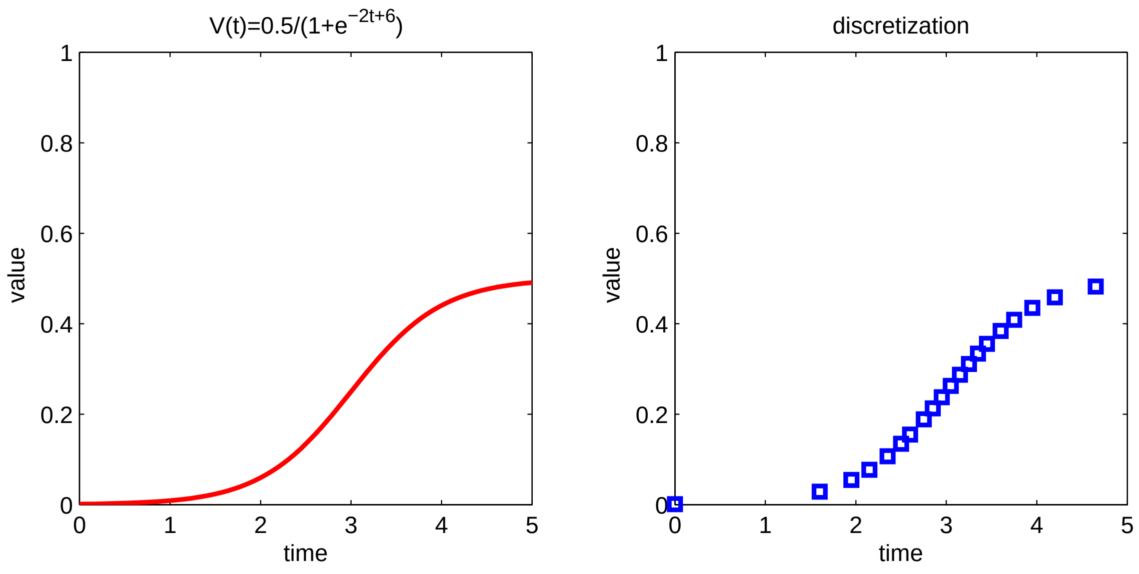


Figure 4.4: A discretization example: a general nonlinear function (left), and its discrete function (right).

Now, we consider more general nonlinear performance profiles, and use discretization to remove nonlinearity in such functions. Figure 4.4 shows an example of discretization. For a detailed discussion of the discretization techniques, we refer to (Powell, 1981).

Let $T_{i,j}$ and $V_{i,j}$ represent the j^{th} time point and its corresponding value on the discretized function of $V_i(t)$, and let binary variable $\delta_{i,j}$ represent whether time point $T_{i,j}$ is selected. The deliberation scheduling problem can then be formulated into the following mixed integer linear program:

$$\max \sum_i v_i \tag{4.4}$$

subject to:

$$\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i \qquad : \forall k$$

(additional constraints on the next page)

$$t_i = \sum_j T_{i,j} \times \delta_{i,j} \qquad : \forall i$$

$$v_i = \sum_j V_{i,j} \times \delta_{i,j} \qquad : \forall i$$

$$\sum_j \delta_{i,j} = 1 \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

$$\delta_{i,j} \in \{0, 1\} \qquad : \forall i, \forall j$$

The constraint $\sum_j \delta_{i,j} = 1$ says that a certain amount of computation time is scheduled for decision procedure $decision_i$. The constraints $t_i = \sum_j T_{i,j} \times \delta_{i,j}$ and $v_i = \sum_j V_{i,j} \times \delta_{i,j}$ model the performance profile $V_i(t)$ through binary variable $\delta_{i,j}$.

It should be noted that, for the phases with continuous concave performance profiles, we can approximate those performance profiles with piecewise linear functions. That is to say, we only need to discretize non-concave performance profiles. This strategy reduces the number of binary variables used in the MILP and thus often improves the computational efficiency.

4.4.3 Determining Optimal Deliberation Intervals

In the discussion so far, it is assumed that τ_i is known *a priori*, but, in many online application domains, τ_i is associated with a cost function $C_i(\tau_i)$ rather than being pre-specified. Let us consider as an example an information gathering agent that responds to a user query, which may choose to immediately return a cached answer, whose computational cost is low but whose solution utility may also be low because the returned information is not up-to-date. The agent can also choose to spend some time querying remote servers, which may result in a high-quality answer but runs the risk of the user losing patience and no longer being interested in the answer. In this and similar situations, an agent needs to determine the amount of computation time it should use, accounting for the cost of using it.

Thanks to the mathematical programming formulation, it is fairly easy to make this extension. We just need to model deliberation cost functions $C_i(\tau)$ with some additional constraints, and account for the cost $\sum_i c_i$ in the objective function. That is,

$$\max \sum_i (v_i - c_i) \tag{4.5}$$

subject to:

$$\begin{aligned}
&\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i && : \forall k \\
&v_i = V_i(t_i) && : \forall i \\
&c_i = C_i(\tau_i) && : \forall i \\
&\tau_i \geq 0 && : \forall i \\
&t_i \geq 0 && : \forall i
\end{aligned}$$

Notice that the deliberation cost functions $C_i(\tau)$ are analogous to the performance profiles $V_i(t)$ in this formulation, which means that we can use the approximation techniques described previously to linearize the constraint $c_i = C_i(\tau_i)$ as well. Specifically, when $C_i(\tau)$ is a continuous convex function (i.e., increasing cost rate), we can approximate it as a piecewise linear convex function, i.e., $C_i(\tau) = \max_j c_{i,j} \times \tau + d_{i,j}$, while, when $C_i(\tau)$ is a general nonlinear function, we can build a discrete function instead.

4.4.4 An Example

This section concludes by examining the algorithm presented above on a simple example problem (more empirical results will be shown in Section 4.7). In this example, there are four phases $phase_{i \in \{0,1,2,3\}}$ whose performance profiles and deliberation cost functions are defined below and illustrated in Figure 4.5.

$$V_0(t) = 3.1319 \times (1 - e^{-0.8233t})$$
$$V_1(t) = 4.0886 \times (1 - e^{-0.3603t})$$
$$V_2(t) = 0.2965 \times (1 - e^{-0.8393t})$$
$$V_3(t) = 2.3293 \times (1 - e^{-0.3057t})$$

$$\begin{aligned} & C_0(\tau) = 0.2037 \times (\tau - 1)^{1.5115} && \text{when } \tau \geq 1 \\ & C_1(\tau) = 0.4808 \times \tau^{1.1843} && \text{when } \tau \geq 0 \\ & C_2(\tau) = 0.4038 \times (\tau - 3)^{1.8348} && \text{when } \tau \geq 3 \\ & C_3(\tau) = 0.2129 \times (\tau - 1)^{1.9415} && \text{when } \tau \geq 1 \\ \text{and } & C_i(\tau) = 0 && \text{otherwise} \end{aligned}$$

Approximating each $V_i(t)$ and $C_i(\tau)$ as a piecewise linear function with 20 pieces, this deliberation scheduling problem can then be formulated as a linear program. With the LP solver *cplex* (www.ilog.com), the total expected utility is 5.40, and solving the LP takes 0.012 seconds.

The deliberation schedule is shown in Figure 4.6. In detail, the agent spends 4.068 time units on its decision procedures for $phase_0$ and $phase_1$ before it starts to execute the mission. After $phase_0$ is completed, it uses 0.6844 additional time units to improve the solution to $phase_1$. Since $phase_2$ has a much lower expected utility

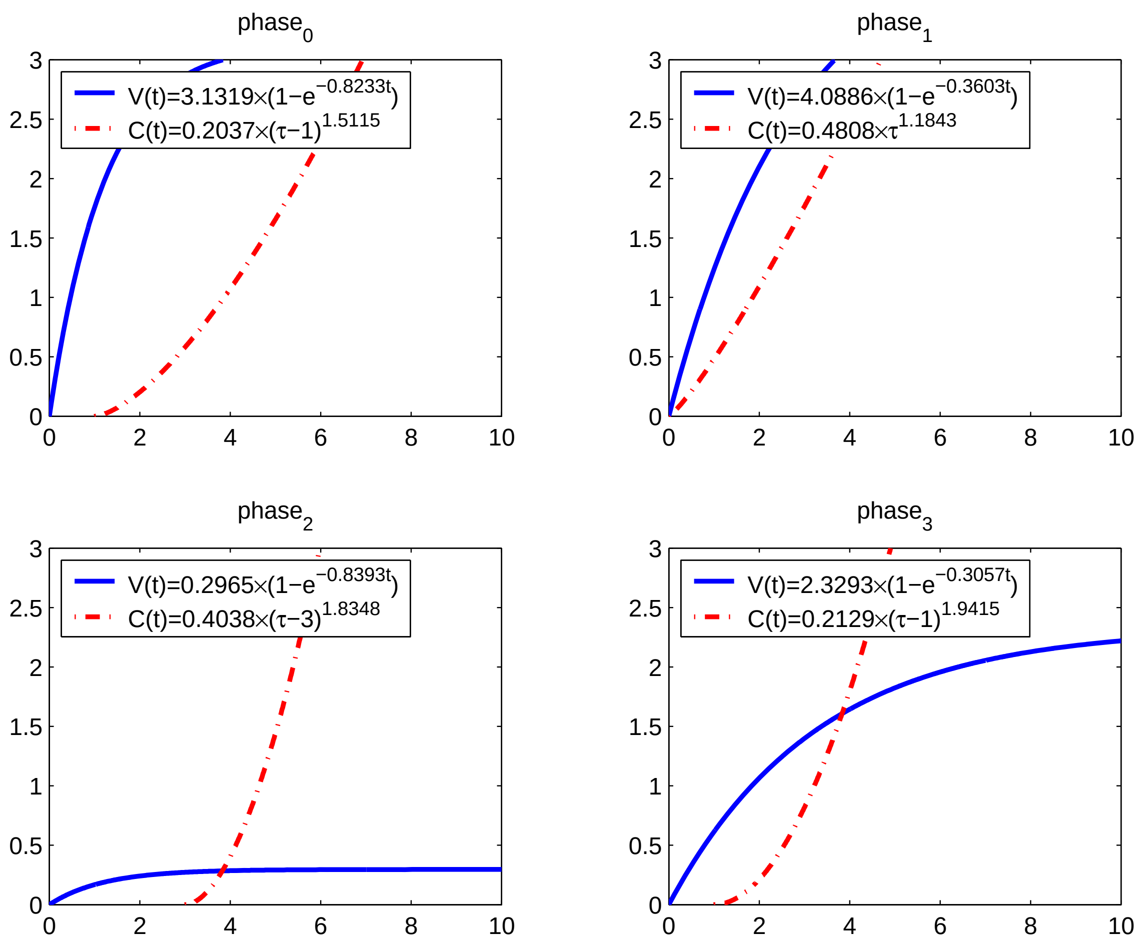


Figure 4.5: A simple example with four phases.

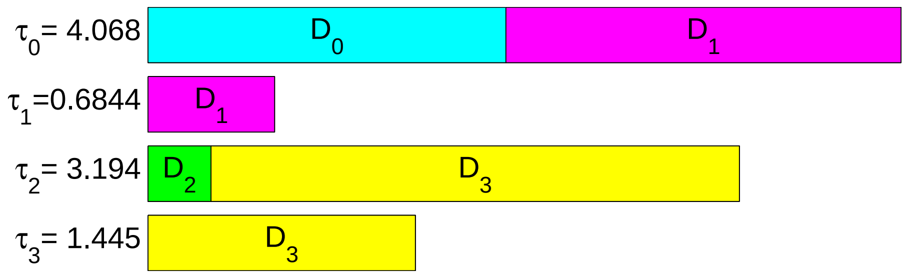


Figure 4.6: Optimal deliberation schedule to the example problem. D_i represents the decision procedure for $phase_i$.

than $phase_3$ (i.e., 0.2965 vs. 2.3293 in the maximum utility), most of the available computation time before executing $phase_2$ is used for the decision procedure for $phase_3$. The resulting schedule achieves 20% higher expected utility than using a myopic algorithm that only runs the decision procedure of $phase_i$ right before that phase.

4.5 Extension: Nonlinear Objective Functions

In the previous section, we have presented solution algorithms for the linear objective function $\sum_i v_i$ and its generalized version $\sum_i(v_i - c_i)$, which fit many application domains where the utility of a mission is the cumulative utility throughout all its phases. An intuitive example is that of an autonomous delivery robot making several rounds of deliveries; its total utility is the sum of the utilities of individual package deliveries.

However, the interests in some application domains might not be the cumulative utility. For example, in the Coordinator domain (Musliner et al., 2006; Wu and Durfee, 2007b), the utility of a task might be the minimum utility of its subtasks. Or, as another example, in the autonomous aircraft domain (Goldman et al., 2001), the utility can be defined as the probability of successfully completing the mission, and thus the overall utility is the product of the probabilities of successfully completing each phase.

Our solution algorithms can be easily revised to suit these (and similar) domains. The underlying idea is that many nonlinear objective functions can be linearized (but not necessarily as $\sum_i v_i$) through mathematical reformulation. Hence, the techniques presented in Section 4.4 are also applicable for them. In the rest of this section, we illustrate this idea by showing how to deal with the two example types of nonlinear objective functions mentioned above.

4.5.1 Minimum of Phase Utilities

In some application domains (e.g., the Coordinator domain (Musliner et al., 2006; Wu and Durfee, 2007b)), the utility of a mission is the minimum utility of individual phases. That is, the problem is to

$$\max \min_i v_i \tag{4.6}$$

subject to:

$$\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i \qquad : \forall k$$

$$v_i = V_i(t_i) \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

In general, directly solving such a nonlinear optimization problem (with a nonlinear objective function $\min_i v_i$) is difficult. However, the nonlinear objective function can be easily reformulated as a linear objective function v with additional linear constraints $\forall i : v \leq v_i$. That is, Eq. 4.6 can become:

$$\max v \tag{4.7}$$

subject to:

$$\begin{aligned} &\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i && : \forall k \\ &v_i = V_i(t_i) && : \forall i \\ &v \leq v_i && : \forall i \\ &t_i \geq 0 && : \forall i \end{aligned}$$

We can then convert Eq. 4.7 into a linear program or a mixed integer linear program through the approximation techniques described in Section 4.4.[4]

4.5.2 Product of Phase Utilities

The objective function $\prod_i v_i$ is often used in application domains where the probability of successfully completing a mission is concerned, e.g., the aforementioned autonomous aircraft domain (Goldman et al., 2001). In such domains, the deliberation scheduling problem can be formulated as:

$$\max \prod_i v_i \tag{4.8}$$

subject to:

$$\begin{aligned} &\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i && : \forall k \\ &v_i = V_i(t_i) && : \forall i \\ &t_i \geq 0 && : \forall i \end{aligned}$$

[4] In a similar manner, we can solve the problem where the intervals of computation time are associated with deliberation cost functions and the objective is to maximize $\min(v_i - c_i)$.

By using a logarithmic transformation ($\varepsilon_i = \ln v_i$), Eq. 4.8 can be reformulated into Eq. 4.9:

$$\max e^{\sum_i \varepsilon_i} \tag{4.9}$$

subject to:

$$\sum_{i=0}^{k} t_i \leq \sum_{i=0}^{k} \tau_i \qquad : \forall k$$

$$\varepsilon_i = V_i'(t_i) \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

where $V_i'(t) \equiv \ln V_i(t)$.

Note that the function e^x is a monotonically increasing function, maximizing $e^{\sum_i \varepsilon_i}$ is equivalent to maximizing $\sum_i \varepsilon_i$, and so the objective function is linearized (while the nonlinearity is moved to performance profiles that can be linearized using the previously presented approximation techniques).[5]

4.6 Extension: Non-Deterministic Phase Transitions

Section 4.5 discussed how to extend the fundamental solution algorithms presented in Section 4.4 to also work for application domains with nonlinear objective functions. This section extends the algorithms in another way — transitions among phases.

4.6.1 Uncertain Phase Transitions

The solution algorithms presented in Section 4.4 are built upon deterministic phase transitions. We now consider deliberation scheduling problems where phase transitions are uncertain and these transitions are not controllable by agents, and we will discuss more general phase transitions in Section 4.6.2 and Section 4.6.3. In

[5]Similarly, we can also use the logarithmic transformation for the problem where the intervals of computation time are associated with deliberation cost functions and the objective to maximize $\prod_i v_i / \prod_i c_i$.

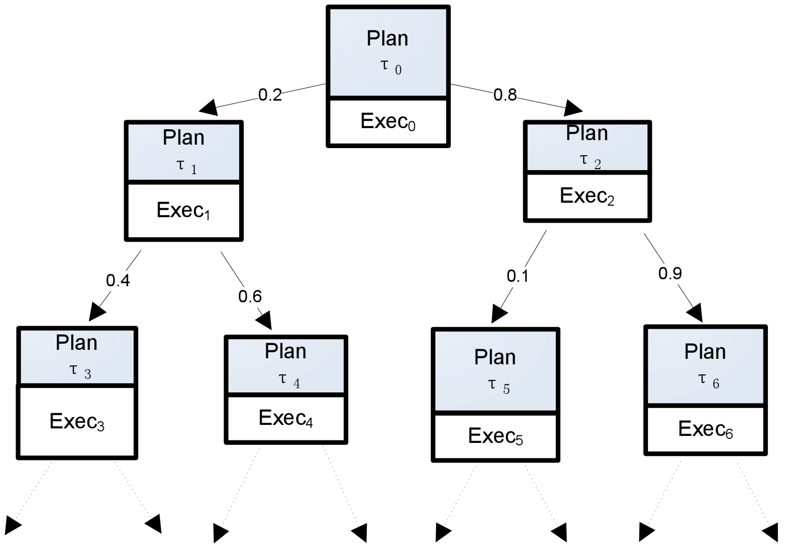


Figure 4.7: Uncertain phase transitions.

this work, we focus on situations where phase transitions can be represented as a tree. Figure 4.7 shows one such problem. When the agent leaves a phase, it will reach one of the subsequent phases with some probability. This study of uncertain phase transitions is similar to Horvitz's previous work (Horvitz, 2001), in which future instances are non-deterministic, but our techniques explore this topic further by explicitly taking into account deliberation costs and more general performance profiles.

We here assume that phase transitions are uncertain but known *a priori*.[6] Let $p_{i,j}$ represent the transition probability from $phase_i$ to $phase_j$, and then the probability of reaching $phase_j$ in the mission, denoted as P_j, can be computed from $p_{i,j} \times P_i$ where $P_0 = 1$. Let $\mathcal{A}_i$ denote the set composed of $phase_i$ and its ancestor phases, and let $\mathcal{D}_i$ denote the set composed of $phase_i$ and its possible descendant phases. Since computation time scheduled for decision procedure $decision_i$ can be from any phase in $\mathcal{A}_i$, $t_i = \sum_{k \in \mathcal{A}_i} \psi_{k,i}$ where $\psi_{k,i}$ is the period of computation time that is

[6]Nevertheless, the solution algorithm presented in this section can be extended to environments where the agent might not know all future phases *a priori*, i.e., by introducing a *leak* phase (with predefined reaching probability and performance profile) to approximately model unknown future phases.

located at $phase_k$ and will be used for $phase_i$'s decision procedure. The deliberation scheduling problem with uncertain phase transitions then becomes

$$\max \sum_i P_i \times (v_i - c_i) \tag{4.10}$$

subject to:

$$\begin{aligned}
t_i &= \sum_{k \in \mathcal{A}_i} \psi_{k,i} && : \forall i \\
\tau_k &\geq \sum_{i \in \mathcal{D}_k} \psi_{k,i} && : \forall k \\
v_i &= V_i(t_i) && : \forall i \\
c_i &= C_i(\tau_i) && : \forall i \\
\tau_i &\geq 0 && : \forall i \\
t_i &\geq 0 && : \forall i \\
\psi_{k,i} &\geq 0 && : \forall k, \forall i
\end{aligned}$$

where the constraints $\forall k : \tau_k \geq \sum_{i \in \mathcal{D}_k} \psi_{k,i}$ guarantee that the amount of scheduled computation time cannot exceed the amount of available computation time at any point within the mission.

The constraints $t_i = \sum_{k \in \mathcal{A}_i} \psi_{k,i}$ and $\tau_k \geq \sum_{i \in \mathcal{D}_k} \psi_{k,i}$ do not introduce additional nonlinearities. Therefore, using the techniques described in Section 4.4, Eq. 4.10 can be approximated into a linear program whenever, for any $phase_i$, $V_i(t)$ is a continuous concave function and $C_i(\tau)$ is a continuous convex function. Otherwise, Eq. 4.10 can be approximated into a mixed integer linear program using discretization techniques. After solving the linear program or the mixed integer linear program, it is trivial to derive deliberation schedules from $\psi_{k,i}$ (since $\psi_{k,i}$ actually explains the schedule by definition).

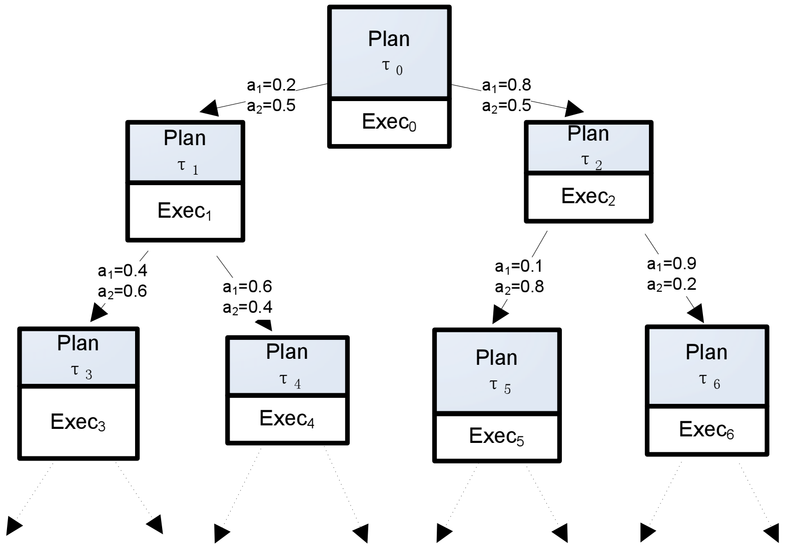


Figure 4.8: Controllable phase transitions.

4.6.2 Controllable Phase Transitions

In the previous sections it is assumed that transition probabilities (either deterministic or uncertain) between phases are specified *a priori*. We here extend our mathematical programming formulation to a more complicated deliberation scheduling problem (shown in Figure 4.8) where phase transitions may be controllable by the agent itself. For example, when there are multiple paths to the same destination, action choices of the agent determine (possibly stochastically) which phase may be reached next, and in turn affect the utility that the agent can receive in its subsequent phases.

With such an extension, deliberation scheduling problems become more challenging because an agent not only needs to control its reasoning, but also needs to find a policy that maps each phase (abstract state) to an action choice.[7] A straightforward way for solving such problems is to enumerate all possible policies, and then, for each policy, adopt the algorithms presented in the previous sections. However,

[7]The actions mentioned here are high-level, abstract actions controlling phase transitions instead of the actions performed in specific states.

when the number of policies is large, this straightforward strategy might become infeasible. This subsection presents an alternative approach, which incorporates the policy formulation process in the previously presented mathematical programming formulation.

Let $p_{i,a,j}$ represent the probability that the agent reaches $phase_j$ if it executes action a in $phase_i$, let α_i represent the probability that the agent is initially in $phase_i$, and let $x_{i,a}$ represent the expected number of times that action a is executed in $phase_i$. We can then formulate deliberation scheduling problems with controllable phase transitions into:

$$\max \sum_i x_i \times (v_i - c_i) \tag{4.11}$$

subject to:

probability conservation constraints:

$$\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a} \qquad : \forall j$$

$$x_i = \sum_a x_{i,a} \qquad : \forall i$$

$$x_{i,a} \geq 0 \qquad : \forall i, \forall a$$

time allocation constraints:

$$t_i = \sum_{k \in \mathcal{A}_i} \psi_{k,i} \qquad : \forall i$$

$$\tau_k \geq \sum_{i \in \mathcal{D}_k} \psi_{k,i} \qquad : \forall k$$

$$v_i = V_i(t_i) \qquad : \forall i$$

$$c_i = C_i(\tau_i) \qquad : \forall i$$

(additional constraints on the next page)

$$\tau_i \geq 0 \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

$$\psi_{k,i} \geq 0 \qquad : \forall k, \forall i$$

where $x_i = \sum_a x_{i,a}$ is the total expected number of times $phase_i$ is visited. In this work, it is assumed that the structure of phase transitions can be represented as a tree and so a phase cannot be visited more than once during the mission (the implications of relaxing this assumption will be discussed in the future work section at the end of this dissertation). That is to say, x_i can be used to represent the probability of visiting $phase_i$.

As introduced in Section 2.2, the constraint $\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a}$ indicates that the expected number of times $phase_j$ is visited must equal the initial probability distribution at $phase_j$ plus the expected number of times $phase_j$ is entered via all possible transitions.

The objective function $\sum_i x_i \times (v_i - c_i)$ in Eq. 4.11, which represents the total expected utility, is a quadratic function (since x_i, v_i and c_i are all variables). Quadratic optimization problems are generally computationally challenging. To deal with this, a way to reformulate Eq. 4.11 into a MILP using discretization approximation techniques is presented below.

Performance-profile-related parameters $T_{i,j}$, $V_{i,j}$ and $\delta_{i,j}$ were defined in Section 4.4. We now define deliberation-cost-function-related parameters $\Gamma_{i,j}$, $C_{i,j}$ and $\sigma_{i,j}$ in a similar way, i.e., $\Gamma_{i,j}$ and $C_{i,j}$ represent the j^{th} time point and its corresponding value in the discretized function of $C_i(\tau)$ respectively, and binary variable $\sigma_{i,j}$ represents whether time point $\Gamma_{i,j}$ is selected, and so Eq. 4.11 can be approximated as the following mixed integer linear program.

$$\max \sum_i \sum_j (\chi_{i,j} \times V_{i,j} - \zeta_{i,j} \times C_{i,j}) \tag{4.12}$$

subject to:

probability conservation constraints:

$$\sum_a x_{j,a} = \alpha_j + \sum_i \sum_a p_{i,a,j} \times x_{i,a} \qquad : \forall i$$

$$x_i = \sum_a x_{i,a} \qquad : \forall i$$

$$x_{i,a} \geq 0 \qquad : \forall i, \forall a$$

time allocation constraints:

$$t_i = \sum_j T_{i,j} \times \delta_{i,j} \qquad : \forall i$$

$$\tau_i = \sum_j \Gamma_{i,j} \times \sigma_{i,j} \qquad : \forall i$$

$$t_i = \sum_{k \in \mathcal{A}_i} \psi_{k,i} \qquad : \forall i$$

$$\tau_k \geq \sum_{i \in \mathcal{D}_k} \psi_{k,i} \qquad : \forall k$$

$$\tau_i \geq 0 \qquad : \forall i$$

$$t_i \geq 0 \qquad : \forall i$$

$$\psi_{k,i} \geq 0 \qquad : \forall k, \forall i$$

constraints for connecting x_i, $\chi_{i,j}$, and $\delta_{i,j}$:

$$\sum_j \delta_{i,j} = 1 \qquad : \forall i$$

$$\sum_j \chi_{i,j} = x_i \qquad : \forall i$$

(additional constraints on the next page)

$$\chi_{i,j} \le \delta_{i,j} \qquad : \forall i, \forall j$$

$$\chi_{i,j} \ge 0 \qquad : \forall i, \forall j$$

$$\delta_{i,j} \in \{0,1\} \qquad : \forall i, \forall j$$

constraints for connecting x_i, $\zeta_{i,j}$, and $\sigma_{i,j}$:

$$\sum_j \sigma_{i,j} = 1 \qquad : \forall i$$

$$\sum_j \zeta_{i,j} = x_i \qquad : \forall i$$

$$\zeta_{i,j} \le \sigma_{i,j} \qquad : \forall i, \forall j$$

$$\zeta_{i,j} \ge 0 \qquad : \forall i, \forall j$$

$$\sigma_{i,j} \in \{0,1\} \qquad : \forall i, \forall j$$

The constraints $\sum_j \delta_{i,j} = 1$ (where $\delta_{i,j}$ are binary variables) and $\chi_{i,j} \le \delta_{i,j}$ ($\chi_{i,j} \ge 0$) indicate that, for each $phase_i$, there exists at most one nonzero variable $\chi_{i,j}$, and the constraint $\sum_j \chi_{i,j} = x_i$ says that this nonzero variable must equal x_i. All these constraints work together to guarantee $\chi_{i,j} = x_i \times \delta_{i,j}$. In a similar manner, we can reason that $\zeta_{i,j} = x_i \times \sigma_{i,j}$.

Now, we can linearize the quadratic objective function in Eq. 4.11. That is,

$$\begin{aligned} &\sum_i x_i \times (v_i - c_i) \\ \simeq &\sum_i \sum_j (x_i \times V_{i,j} \times \delta_{i,j} - x_i \times C_{i,j} \times \sigma_{i,j}) \\ = &\sum_i \sum_j (\chi_{i,j} \times V_{i,j} - \zeta_{i,j} \times C_{i,j}) \end{aligned}$$

which is the linear objective function (where $V_{i,j}$ and $C_{i,j}$ are constants) used in Eq. 4.12.

$\psi_{k,i}$ in the solution to Eq. 4.12 represents the deliberation schedule, and the policy that maps each phase to its action choice can be derived from $x_{i,a}$: in $phase_i$, action a is executed with probability $\pi_{i,a} = \frac{x_{i,a}}{\sum_a x_{i,a}}$.

4.6.3 Policy-Oriented Phase Transitions

In the most difficult deliberation-scheduling problems, the probabilities of reaching subsequent phases may depend on the policy formulated in the current phase. That is to say, transition probabilities among phases may change as an agent spends more time computing (better) phase policies.

A preliminary algorithm to solve such challenging policy-oriented-phase-transition problems is to interleave the process of scheduling deliberations (using the algorithms presented earlier in this chapter) and the process of formulating policies (using anytime policy formulation algorithms, some of which will be discussed in the next chapter). That is, the agent starts by scheduling phase decision procedures with some estimates of phase transition probabilities (e.g., assuming the same transition probability to all possible subsequent phases), and then formulates phase policies according to the resulting schedule. During the policy formulation process, if it turns out that actual phase transition probabilities deviate a lot from the previously estimated probabilities, the agent will stop building policies. It will run the deliberation scheduling algorithm again with the updated phase transition information, and then continue to formulate policies according to the updated deliberation schedule.

It is clear that this is a myopic algorithm, and it may be possible to do better by exploiting domain-specific knowledge that can used to predict the changes of phase transitions over time. However, a thorough investigation of this topic is beyond the scope of this dissertation.

4.7 Experimental Evaluation

It is important to remember that the running time of the deliberation scheduling algorithm itself consumes computation time that could otherwise be used for deliberation. In other words, if an agent spends too much time scheduling deliberations, then it might have too little time to actually deliberate. As shown by Goldman et al. (2001), a MDP-based algorithm can also solve general deliberation scheduling problems (like ours), but they also pointed out that the computational complexity of that MDP-based algorithm is exponential in the number of phases and thus usually not applicable in time-limited domains. This section gives an empirical evaluation of our algorithms, particularly in the aspect of computational efficiency.

The results presented in this section are based upon test problems where the objective is to maximize the cumulative reward across all phases. However, the problems with the nonlinear objective functions discussed in Section 4.5 would have similar results (and our experiments confirmed this argument) because our mathematical reformulation of those objective functions does not considerably affect computational complexity of the test problems.

We here evaluate our deliberation scheduling techniques using some randomly generated performance profiles and deliberation cost functions (the detailed procedure is presented below). We will evaluate the deliberation scheduling techniques again in a realistic application domain after presenting our anytime policy formulation algorithm and constructing its performance profiles in the next chapter.

The rest of this section is organized as follows. We start by discussing our experimental setup. We then evaluate our algorithms for deterministic phase transitions, uncertain phase transitions, and controllable phase transitions, respectively.

4.7.1 Experimental Setup

The choices of performance profiles and deliberation cost functions as well as their exact parameters are not critical for the trends seen in the results presented in this section, but for the sake of reproducibility the details are described here.

We use continuous concave functions $V_i(t) = M \times (1 - e^{-K \times t})$ to evaluate the LP-based algorithm, and use general nonlinear functions $V_i(t) = \frac{Q}{1+e^{-J \times (t-D)}}$ to evaluate the MILP-based algorithm. The examples of those functions have been shown in Figure 4.3 and Figure 4.4. In both cases, continuous convex functions $C_i(\tau) = C \times \tau^N$ are used as our deliberation cost functions.

The parameters are randomly set: $M \sim [0.5, 5.0]$, $K \sim [0.05, 0.5]$, $Q \sim [0.5, 5.0]$, $J \sim [1.0, 2.0]$, $D \sim [1.0, 3.0]$, $C \sim [0.05, 0.5]$, and $N \sim [1.3, 1.6]$, where $x \sim [L, U]$ represents that x is uniformly distributed in the range $[L, U]$. The only rule we used to choose parameter ranges is to avoid a "simple zone" of the test problems where the deliberation costs are so high that it is obvious that none of the deliberations should be done.

In our experimental results shown in this section, each data point is the average value from 100 runs, and curves in the following figures are smoothed to improve readability.

4.7.2 Deterministic Phase Transitions

Figure 4.9 shows the computational efficiency of our algorithms in solving deliberation scheduling problems with deterministic phase transitions, where m in the figure denotes the number of pieces for each function if using piecewise linear approximation, and denotes the number of points for each function if using discretization. The y-axis of the figure specifies the total amount of runtime, including the time for

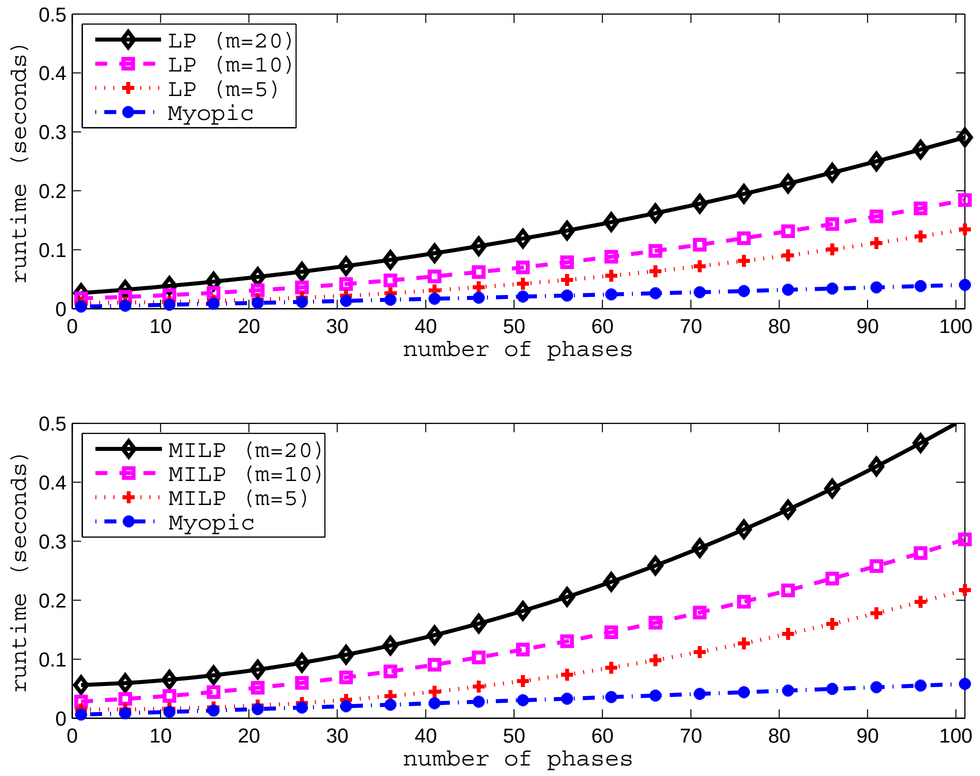


Figure 4.9: Runtime of the LP-based algorithm (top) and the MILP-based algorithm (bottom) for deterministic phase transitions.

making the piecewise linear approximation or discretization, and the time for constructing and solving a LP/MILP. We can see that, although slower than the myopic algorithm that attempts to maximize $V_i(t) - C_i(t)$ at each individual phase without worrying about future phases, our algorithms compute near-optimal solutions reasonably fast, especially when m is small, and their solution utilities significantly outperform that of the myopic algorithm as shown in Figure 4.10.

Not surprisingly, using a small m will reduce the approximation accuracy, and thus impair optimality. In Figure 4.10, we evaluate the optimality of our algorithms with various m, the myopic algorithm, and a naive algorithm that does not take into account additional available time in the midst of execution (but it can optimize the use of the available time prior to execution). Since, to the best of our knowledge,

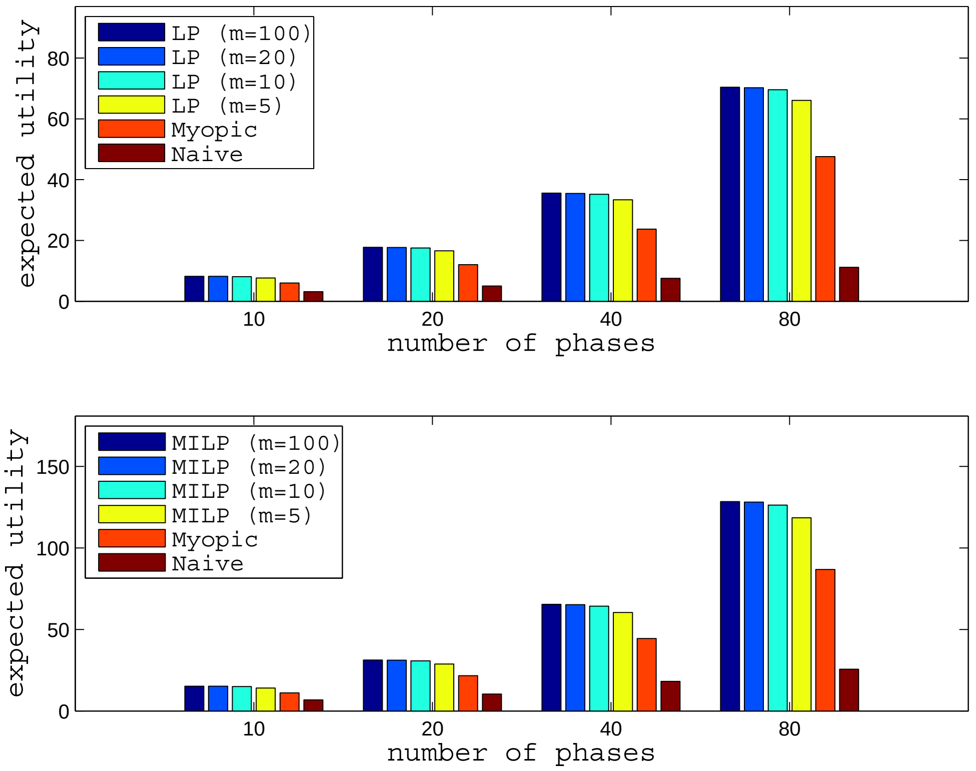
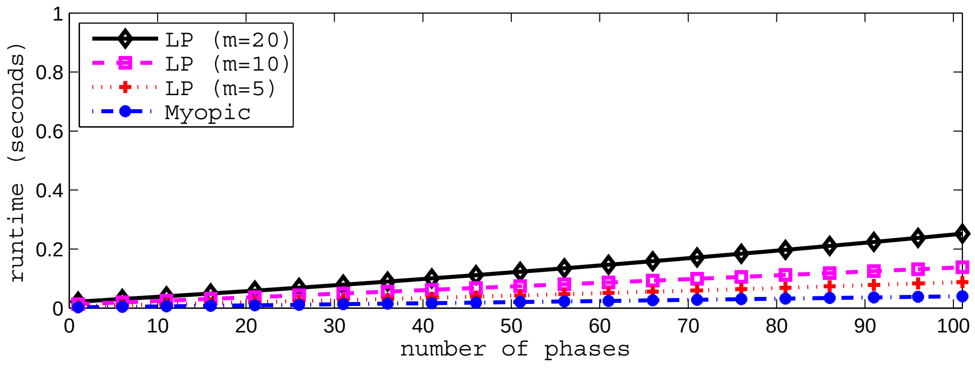


Figure 4.10: Optimality of the LP-based algorithm (top) and the MILP-based algorithm (bottom) for deterministic phase transitions.

there are no existing algorithms that are able to compute optimal deliberation schedules in those test problems (because of their nature of being nonlinear optimization problems), we use the solution of our algorithms with a large m (i.e., $m = 100$, which can usually make the approximation function very close to the input function) as the baseline. These empirical results show that, with $m = 20$, our algorithms are close to optimal. More importantly, we can see that our algorithms, even with a small m (such as m=5), can result in a much higher utility than the myopic algorithm and the naive algorithm.

4.7.3 Uncertain Phase Transitions

When testing our algorithms in problems with uncertain phase transitions, we assume that the phase transitions can be represented as a complete binary tree. When



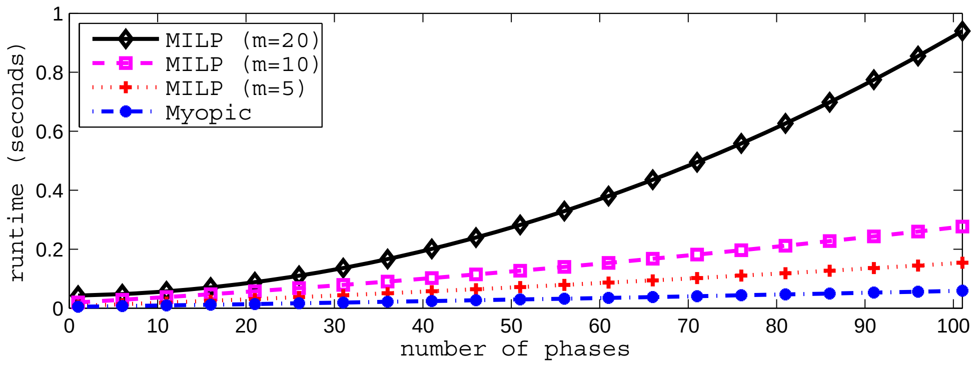
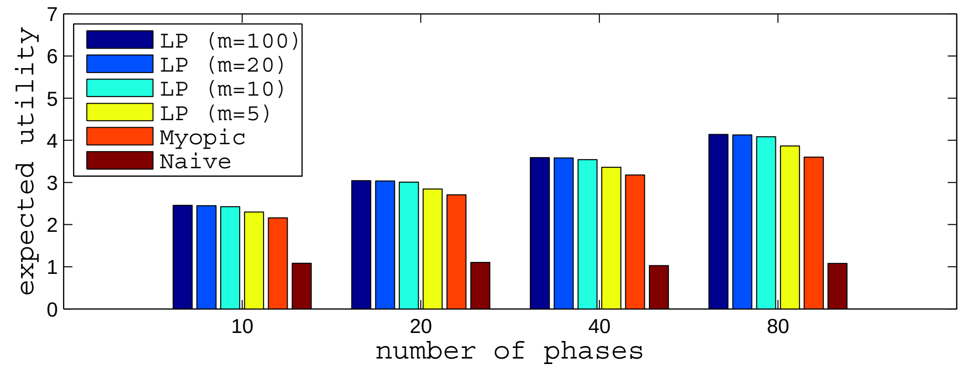


Figure 4.11: Runtime of the LP-based algorithm (top) and the MILP-based algorithm (bottom) for uncertain phase transitions.

the agent leaves a phase with two children, it will reach its left child and its right child with probability ρ and $1 - \rho$, respectively, where ρ is uniformly distributed in the range [0.0, 1.0]. Figure 4.11 shows the computational efficiency of our algorithms. Similarly as before, our algorithms are able to compute a near-optimal deliberation schedule within one second for a complex mission with 100 phases.

It is also interesting to note that the myopic algorithm also performs pretty well on average in Figure 4.12. This is because, for uncertain phase transitions, the time spent on a future phase is less valuable since it is possible that the agent will eventually not reach that future phase. In other words, in such cases, the agent is more prone to act myopically. However, unlike the myopic algorithm that always focuses on the upcoming phase, our mathematical-programming-based approach can



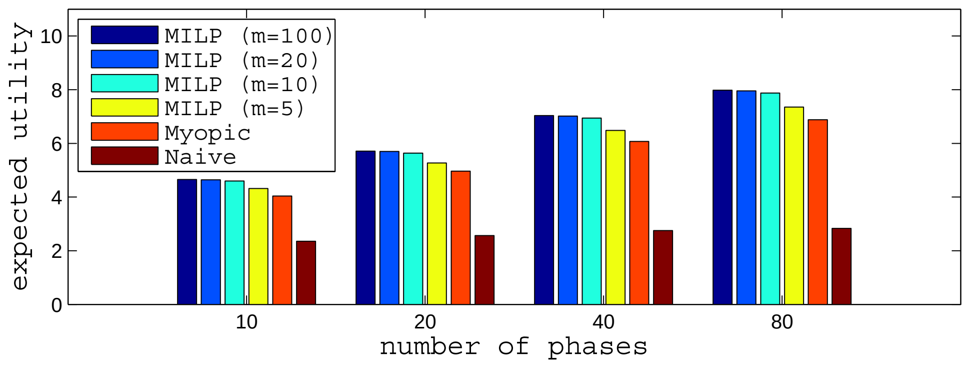


Figure 4.12: Optimality of the LP-based algorithm (top) and the MILP-based algorithm (bottom) for uncertain phase transitions.

look ahead and decide by itself which phases it should focus on. That is to say, our algorithms will not miss good opportunities of “preheating” future high-value phases. As shown in Figure 4.13, though our algorithm uses a very coarse approximation with $m = 5$ (and so its computation time is close to the myopic algorithm as was shown in Figure 4.11), it outperforms the myopic algorithms in most cases (97%).[8] It achieves a utility 85% higher than the myopic algorithm in the best case, and achieves a utility only 2% lower than the myopic algorithm in the worst case.

4.7.4 Controllable Phase Transitions

Finally, we evaluate our algorithm in solving problems with controllable phase transitions. It is again assumed that the phase transitions can be represented as a

[8]Indeed, if we use a sufficiently large m, the solution utility of our algorithm will never be lower than the myopic algorithm.

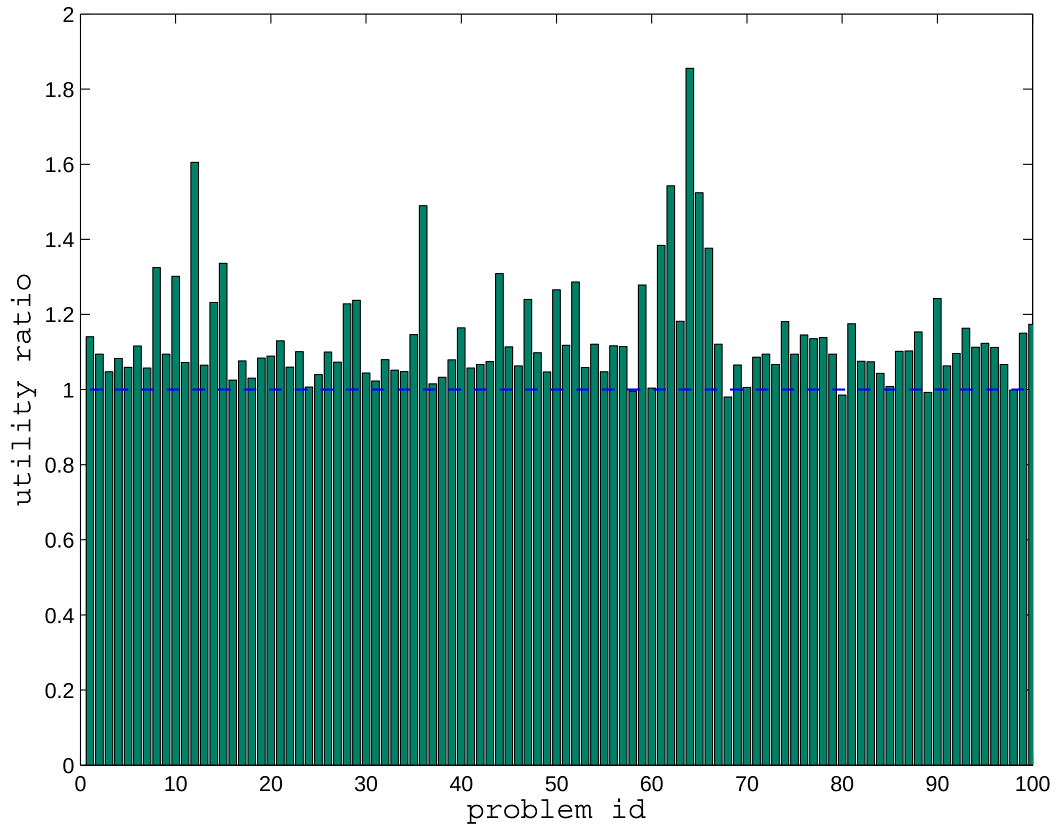


Figure 4.13: The utility ratio of the LP-based algorithm ($m = 5$) to the myopic algorithm on 100 test problems.

complete binary tree. At each phase with two children, there are two possible actions a_1 and a_2. a_1 moves the agent to the left child with probability ρ and to the right child with probability $1 - \rho$, and a_2 achieves the opposite effect. As is easy to see, the problems with controllable phase transitions will be reduced to problems with deterministic or uncertain phase transitions once action choices are made. That is, the optimality comparison results in the problems with controllable phase transitions would be similar to the previous results. Our experiments have confirmed this: the results are similar to Figure 4.10 when ρ is close to 1.0 (i.e., more deterministic), and similar to Figure 4.12 when ρ is close to 0.5 (i.e., more random).

On the other hand, the problems with controllable phase transitions are in general more computationally challenging than those with deterministic or uncertain phase transitions. When the number of phases is large in a complex problem, the MILP-based algorithm might need a relatively long time to find an exact deliberation schedule. Note that state-of-the-art MILP solvers (such as cplex) are usually able to return a good solution using much less time. Thus, for an online application problem, we can adopt a two-step algorithm: it first derives a policy (a mapping from phases to actions) by solving Eq. 4.12 with a limited time; with that policy, the problem is reduced to an easier one with deterministic/uncertain phase transitions, and then it can solve the reduced problem again and return a deliberation schedule.

As shown in Figure 4.14 where the solution utility without computational time limitation is normalized to one and error bars show standard deviation, this two-step algorithm is usually able to compute an approximately optimal deliberation schedule within a short time, which makes it applicable in time-limited domains.

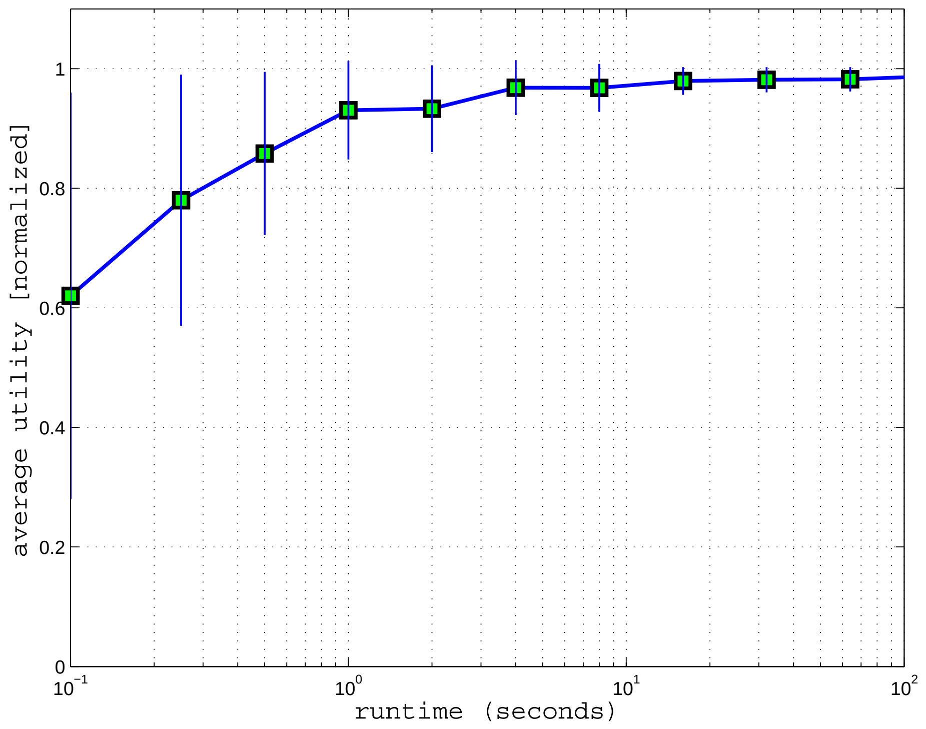


Figure 4.14: Average anytime performance of the two-step algorithm for controllable phase transitions. Parameters are set as follows: 30 phases, $m = 10$, and $\rho = 0.9$.

4.8 Summary

Deliberation scheduling is the process of scheduling decision procedures to maximize the overall system performance, and it is the core component of our computation-driven mission-phasing techniques. This chapter has presented a mathematical-programming-based approach for scheduling phase decision procedures, and illustrated it through several increasingly complex classes of deliberation scheduling problems. The presented algorithms can simultaneously and efficiently solve the coupled problems of deciding both when to deliberate given its cost and which decision procedures to execute during deliberation intervals. In comparison with prior work, this work can cope with a richer set of performance profiles and deliberation cost functions (through piecewise linear approximation and discretization techniques), and is applicable in complex stochastic domains where phase transitions may be uncertain.

The contributions of the work presented in this chapter are listed as follows:

- This work explicitly takes into account computation time that may be used in the midst of execution, and models the cost of using such computation time with deliberation cost functions. A new deliberation scheduling approach has been designed, which can help a time-limited agent judiciously determine the amount of computation time it should use, accounting for the cost of using that time, in several complex settings. As shown in the experimental results, this approach can improve agent performance in both deterministic environments (Figure 4.10) and stochastic environments (Figure 4.12).

- This work extends prior work that focused on continuous concave performance profiles. By formulating a deliberation scheduling problem into a mathematical program and using approximation techniques, this work can find a near-optimal deliberation schedule for decision procedures with any types of performance profiles.

- Furthermore, the mathematical programming formulation provides a domain-independent framework on which we can easily make simplifying transformations or impose additional constraints. The extension for coping with nonlinear objective functions has been introduced in Section 4.5, and the extension for handling non-deterministic phase transitions has been discussed in Section 4.6.

- Most importantly, the deliberation scheduling approach presented in this chapter is computationally efficient. The empirical results (shown in Figure 4.9 and Figure 4.11) have highlighted its ability to find a near-optimal schedule within a short time, e.g., finding a near-optimal schedule for a complicated problem with 100 phases within one second.

CHAPTER V

Effective Inner-Phase Heuristic Search

In Chapter V, we presented, analyzed and empirically evaluated a mathematical-programming-based deliberation scheduling approach, which we showed to be efficient and effective in the management of limited computation time in a wide variety of environments with different degrees of complexities and uncertainties. In general, that deliberation scheduling approach can judiciously schedule policy formulation procedures performed by any anytime policy formulation algorithm. Nevertheless, to make the most effective use of scarce time in time-critical systems, the deliberation scheduling approach should, not surprisingly, collaborate with a policy formulation algorithm with the best possible anytime performance.

To address this issue, this chapter investigates the problem where an autonomous agent has a finite amount of "think time" (assigned by our deliberation scheduling algorithm) for each phase, during which the agent builds and solves a Markov decision process for the corresponding phase decision procedure, after which the agent executes the policy of the phase MDP it has solved. Not surprisingly, time limitations could mean that the agent is unable to model and reason over the full state space of the phase (even though each phase decision procedure is often much simpler than the overall decision procedure), in which case the agent would only be able to find a policy for the portion of the state space it does generate. In executing this policy, if the agent reaches a state that is at the edge of its generated state space, the policy does not provide an action choice for this state or any subsequent state.

The objective in this chapter is to design an anytime policy formulation approach for finding a high-quality (partial) solution for each phase within its time limit. To this end, this chapter develops a heuristic search approach, highlighting the following two features. First, to speed up the process of finding a high-quality solution, the heuristic search approach selectively explores and expands the state space, i.e., focusing on a subset of states that are believed to lie along trajectories of an approximately optimal policy, balanced by spending less computational efforts on other states. Second, besides the process of formulating a complete policy in that selectively explored state space, the approach also adopts a fast planning algorithm to generate "coarse" solutions for states outside the explored space, which help the agent handle, though maybe not optimally, additional eventualities that are not captured in its formulated policy.

The rest of this chapter is organized as follows. Section 5.1 reviews related work in the field of anytime policy formulation. Section 5.2 describes the ideas behind our heuristic search approach. As an illustration (and also for the sake of evaluation), Section 5.3 explicitly implements these general and fundamental ideas in a class of challenging time-constrained problems represented in TÆMS models, and Section 5.4 empirically compares our solution approach with several prior heuristic search methods. In Section 5.5, we give a preliminary evaluation of the overall computation-driven mission-phasing approach, in which heuristic search is incorporated with problem decomposition and deliberation scheduling to further improve the effectiveness of utilizing limited time. Finally, Section 5.6 summarizes the work presented in this chapter.[1]

5.1 Background: Heuristic Search

In numerous application domains, only a small portion of the state space of the complex stochastic system can be reached by following an optimal policy from some predefined initial state. This fact has inspired the development of a number of efficient policy formulation algorithms. Typically, these algorithms adopt heuristic search techniques to selectively expand and explore a large state space, and they are often able to generate an optimal policy while avoiding exhaustive enumeration of all possible states. This section briefly introduces three popular heuristic search algorithms.

5.1.1 RTDP

The *real-time dynamic programming* (RTDP) algorithm is one of the most well-known heuristic search approaches (Barto et al., 1995). The algorithm performs successive trials on the environment. Each trial starts at the initial state of the world and ends at a goal state. In each trial, value updates are only performed on the states actually visited in that trial. The fundamental advantage of this algorithm is that it can quickly avoid paths that lead to low rewards. Thus, the exploration looks mainly at a promising subset of the state space.

The procedure of the RTDP algorithm is outlined as follows:

▷ Start with an admissible value function V.

▷ Repeat trials until the time limit is reached or an optimal policy is found.

For each trial, start by setting the current state i to the initial state and then repeat the following steps until reaching a goal state:

[1]This chapter is partially based on work that was originally reported in (Wu and Durfee, 2007b).

I. Improve the value function by performing Bellman backup on the current state i

$$V(i) \leftarrow \max_{a\in A_i}[\, r_{i,a} + \sum_{j\in S} p_{i,a,j} \times V(j)\,]$$

II. Pick up the best greedy action based upon

$$\pi(i) \leftarrow \operatorname*{argmax}_{a\in A_i}[\, r_{i,a} + \sum_{j\in S} p_{i,a,j} \times V(j)\,]$$

and change the current state i to the next state that results from a sample stochastic transition by performing that action.

To improve convergence speed and/or anytime performance of the RTDP algorithm, several variations of the algorithm have recently been developed. The *labeled RTDP* (LRTDP) algorithm speeds up the convergence process by keeping track of the states over which the value functions have already converged, and thus it can avoid visiting those states again (Bonet and Geffner, 2003).

The *bounded RTDP* (BRTDP) algorithm and the *focused RTDP* (FRTDP) algorithm maintain both upper and lower bounds on the value function, and so they can focus on states that are both relevant (likely reached under the current policy) and poorly understood (large gap between upper and lower bounds), which has been shown to be able to improve agent performance in (at least) stochastic shortest path problems (McMahan et al., 2005; Smith and Simmons, 2006).

5.1.2 Envelope

The *envelope* algorithm (Dean et al., 1995) is an alternative heuristic search approach. It starts with a restricted state space (or *envelope*) that only contains a path from the initial state to the goal state, and then gradually extends that envelope to include more states and computes new policies. Given more computation time, the algorithm will compute a more complete partial policy.

The procedure of the algorithm is summarized below:

- ▹ Start with an envelope including only a nominal path from the initial state to the goal state.
- ▹ Repeat the following steps until the time limit reaches or an optimal policy is found.

  I. Extend the envelope by including states that are outside the envelope of the current policy but that may be reached upon executing the policy. There are several possible strategies for choosing which states to add. For a detailed discussion of these strategies, we refer to (Dean et al., 1995).

  II. Compute a new policy in the extended envelope using the policy iteration algorithm. The policy generated in the previous step can be used as the starting point for policy iteration, which can usually result in fewer iterations for finding an optimal policy within the envelope.

The *envelope* algorithm is particularly good at solving "goal-oriented" MDPs, but one potential drawback is that it is not directly applicable to general MDPs in which there might be no explicit goal states.

5.1.3 AO* and LAO*

The AO* (Martelli and Montanari, 1978) algorithm and its recent extension (the LAO* algorithm (Hansen and Zilberstein, 2001)) are analogous to the well-known A* search algorithm. They start search at the initial state, and use an admissible heuristic function to direct the search. They repeatedly expand the "best" partial solution graph until a complete optimal policy is found. The procedure is outlined below, in which forward expansion of the best partial solution graph is interleaved with a state value revision step:

- ▷ Start with an explicit graph including only the initial state.
- ▷ Repeat the following steps until the best solution graph has no non-terminal tip states.

  I. Expand some non-terminal tip state n of the best partial solution graph.

  II. Update state values $V(i)$ based upon some admissible heuristic evaluation function $h(i)$, i.e.,

$$V(i) = \begin{cases} h(i) & \text{if } i \text{ is a non-terminal tip state} \\ \max_{a \in A_i} [\, r_{i,a} + \sum_{j \in S} p_{i,a,j} \times V(j) \,] & \text{otherwise} \end{cases}$$

  II. Mark best actions according to

$$\pi(i) \leftarrow \operatorname*{argmax}_{a \in A_i} [\, r_{i,a} + \sum_{j \in S} p_{i,a,j} \times V(j) \,]$$

  and update the best partial solution graph.

The AO* and LAO* algorithms are designed as efficient off-line policy formulation methods. They can often find an optimal solution without searching all reachable states.

5.2 Coping with a Very Large State Space

Unlike a classical dynamic programming algorithm (e.g., value iteration and policy iteration) that evaluates the full state space and finds an optimal policy for every state, the heuristic search algorithms introduced in Section 5.1 can often significantly reduce efforts spent on states that will never be reached by following an optimal policy from the initial state. In problems with large state spaces, this has an obvious advantage over dynamic programming since the heuristic search algorithms might find optimal solution policies from the initial state by considering many fewer states.

This section outlines a new heuristic-search policy formulation approach, which adopts a similar idea of being selective in expanding and exploring a large state space. However, unlike much prior work (e.g., the AO* algorithm) emphasizing the reduction of computation time for finding an optimal policy, our heuristic search approach puts emphasis on finding a high-quality solution within a pre-specified computation bound, and highlights the ability to handle challenging situations where computation time is so limited that even an agent that can "perfectly" expand its state space cannot (within the time limits) search every state reachable by the optimal policy.

Figure 5.1 captures the essential ideas of our heuristic search approach. The limitation of computation time restricts the number of states that an agent can expand and explore. The decision about which subset of the MDP state space to be expanded ("unrolled") will affect the quality of the derived policy. Our heuristic search algorithm, named *informed unroller* (IU), biases expansion towards states that are believed to lie along trajectories of high-quality policies. Specifically, the IU algorithm prioritizes the queue of states waiting to be expanded based on an estimate of the likelihood that the state would be encountered when executing an (approximately) optimal policy from the initial state. In other words, the IU algorithm emphasizes the exploration of the state space that is likely to be reached by following approximately optimal policies, while ignoring other states, to yield a better policy when the time limit is reached.

In order to correctly estimate the probabilities of states being reached by the optimal policy, a decision-making agent should take into account probable actions in the future when it evaluates states at the edge of its partially unrolled state space since these evaluations will affect the policy that is used to estimate the reaching probabilities. Clearly, in time-limited environments, considering all possible future

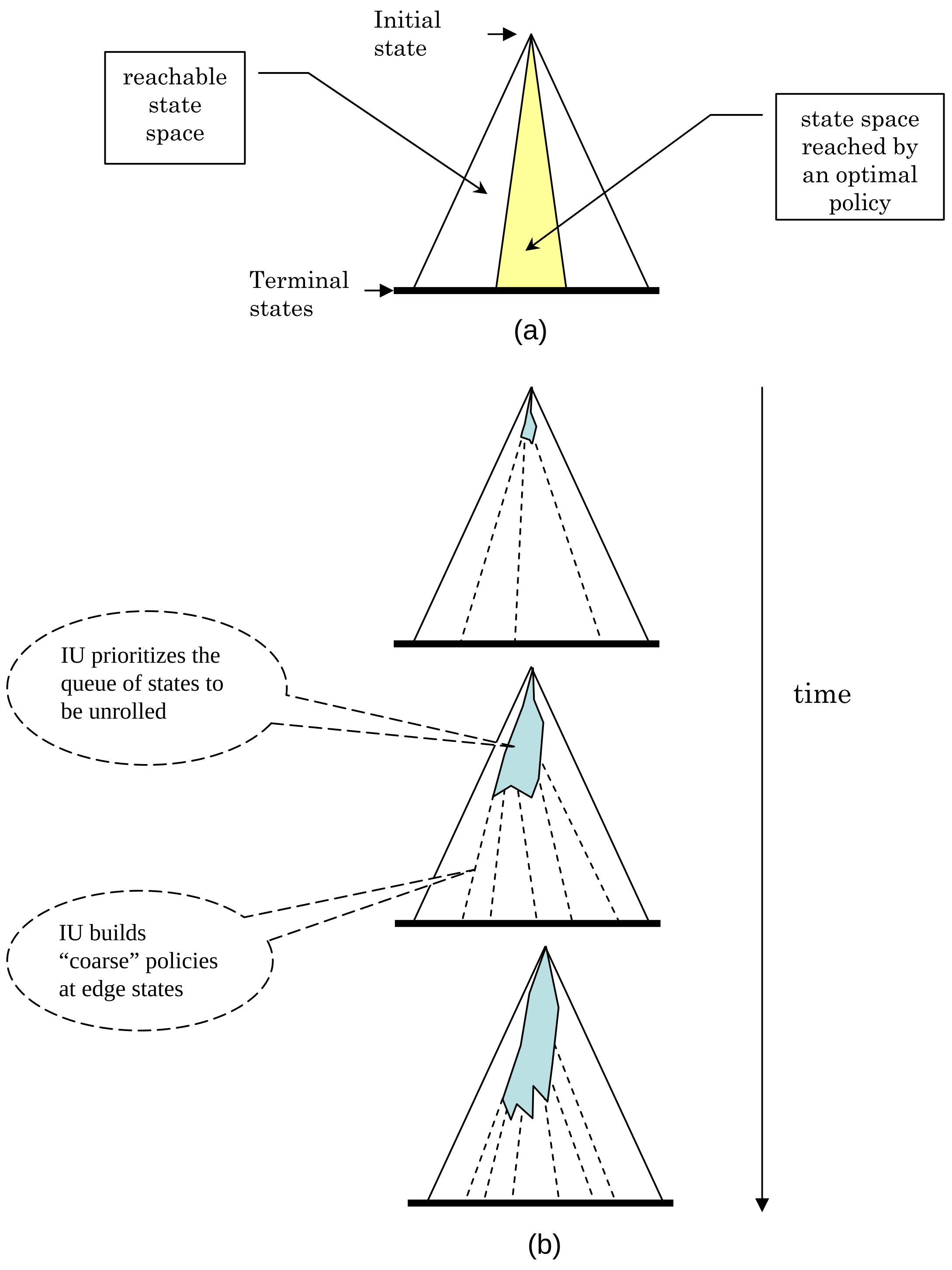

Initial state
reachable state space
state space reached by an optimal policy
Terminal states
(a)
IU prioritizes the queue of states to be unrolled
time
IU builds "coarse" policies at edge states
(b)

Figure 5.1: Illustration for the informed unroller algorithm. (a) An illustrative example about the complete reachable state space and a subset of the states that may be reached by following an optimal policy. (b) A typical sequence of state space expansions using the IU algorithm.

eventualities is computationally infeasible since it requires a full look-ahead (which is indeed equivalent to optimally solving the problem). One alternative practicable way is to generate fast but "coarse" solutions, which start at the edge states and end at some terminal states, using a fast planning algorithm. For the sake of reducing computational costs, the IU algorithm builds such coarse solutions based upon simplified MDP models in which stochastic state transitions are reduced to be deterministic. Without uncertainties, the complicated stochastic-planning problem modeled in the MDP becomes a considerably simpler classical planning and search problem. The decision-making agent can then choose to use a fast search method (such as the breadth-first search and the best-first search) to find an optimal deterministic plan, or to use an even faster greedy planning method to find a myopically optimal solution.

These plausible solutions are used to estimate the values of the edge states, e.g., through predicting and evaluating execution trajectories of the world. The partial policy formulated based upon such estimates is believed to be able to match an optimal policy reasonably well because that formulated policy can partially take into account future eventualities that are not modeled in the explored state space but that may be encountered when following the complete optimal policy.

Moreover, it is worth pointing out an additional advantage from building those coarse solutions: in situations where an agent has no (or very limited) computational capability in the midst of execution, the coarse solutions may, in a timely manner, tell the agent how to act when execution runs past the edge states. This is clearly an advantage over the naive approach of letting the agent randomly pick up an applicable method when out of the partial policy.

Recall that in the problems of interest in this chapter the computation time limit

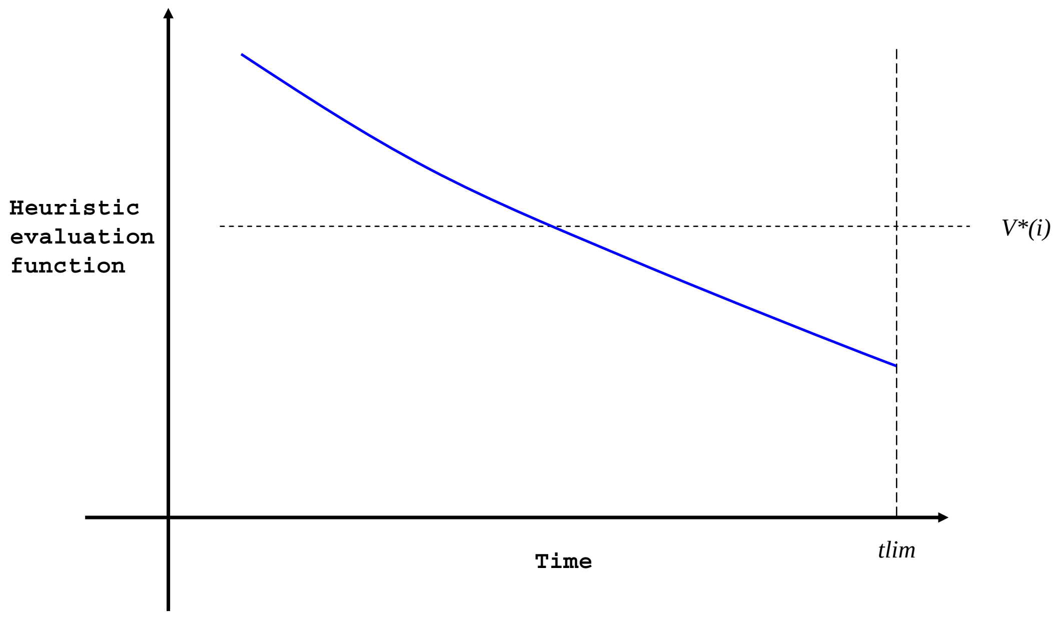


Figure 5.2: An example non-stationary heuristic evaluation function. *tlim* represents computation time limit and $V^*(i)$ represents the optimal state value of state i.

is fixed and known *a priori* (since computation time is assigned by the deliberation scheduling algorithm prior to the policy formulation stage). To make better usage of the computation time, the IU algorithm adopts a non-stationary heuristic evaluation function, which is illustrated in Figure 5.2. When computation time is sufficient, the IU algorithm adopts an admissible (or approximately admissible) heuristic evaluation function that not only evaluates the coarse solution built on an edge state but also considers potential improvements that can be made on that coarse solution by further state space expansion and exploration efforts. In a similar manner to many prior heuristic search algorithms (e.g., the LAO* and RTDP algorithms), such an admissible heuristic evaluation function may help the agent find alternative policies that are better than the previously formulated ones. On the other hand, the heuristic evaluation function decreases as the amount of computation time used increases. When the time approaches its limit, the IU algorithm adopts a non-admissible heuristic evaluation function that is only dependent on the edge state and the coarse solution built on it (but not taking into account possible improvement on that coarse solution since no computation time is available for it). Typically, expanding the best-so-far policy using a non-admissible heuristic function that underestimates state values would improve the estimate of the utility of that policy, making it further outperform other partial policies. That is to say, in situations with scarce computation time, the IU algorithm would rather focus on making its current policy more complete than exploring and examining alternative policies that may be better than the current policy, but only if given sufficient time for state space expansion.

So far, we have described the fundamental and general concepts of the informed unroller algorithm. To better illustrate these ideas and to empirically evaluate and compare this algorithm with prior methods, in the next section the IU algorithm is implemented to solve a particular class of time-constrained problems that are represented in TÆMS models.

5.3 Heuristic Search for Large TÆMS Problems

TÆMS is a hierarchical modeling language capable of representing complex task networks with intra-task uncertainties and inter-task dependencies (Lesser et al., 2004). It has been widely used to model complex realistic applications, such as the Information Gathering problem (Wagner et al., 2006), and the Coordinator problem (Musliner et al., 2006).

One way to find an optimal solution to a single-agent TÆMS problem is to expand the states implicitly defined in the TÆMS model into a Markov decision process (Wagner et al., 2006), and then build a policy using a MDP policy formulation algorithm, such as the backward induction algorithm and the value iteration algorithm. However, this solution approach quickly becomes infeasible as the size and the complexity of the TÆMS model increases. For example, in a moderately complex TÆMS model with $m = 20$ applicable methods where each method has $o = 9$ different possible outcomes and $d = 8$ methods can be performed over one execution, there are about $(o \times m)^d \approx 1.1 \times 10^{18}$ possible states for the agent to generate and reason over.

To address such large state spaces, our IU algorithm constructs a policy for its selectively-generated state space, and generates greedy plans starting at the edges of the expanded state space. As will be seen, the combination of the sequential decision making (for a subset of states that are likely to be reached by high-quality policies) and the fast planning (for other relevant states) can often yield a good solution to a large TÆMS problem within a short time in both the situations where the agent has limited computational capability during execution and the situations where the agent has no computational capability during execution.

The following discussion begins by introducing the TÆMS model in Section 5.3.1, followed by a recap of the prior approach for fully unrolling a TÆMS model into a MDP in Section 5.3.2. The detailed implementation of our heuristic-search-based IU algorithm is explained in Section 5.3.3.

5.3.1 Introduction: TÆMS Models

In a TÆMS task model[2] (Lesser et al., 2004), leaf nodes represent *methods* (primitive actions). A method might have multiple possible outcomes with different durations and qualities. An internal node in the model is a *task*, which is associated with a *quality accumulation function* (QAF) (such as *sum*, *min*, and *max*) that describes how the qualities of the subtasks of a task can be used to calculate the quality of

[2]Since we focus on single-agent problems, we omit the introduction of multi-agent interactions that can be represented in TÆMS models.

the task itself. A node in the TÆMS model might be constrained by *release time* (earliest possible start time) and *deadline*, and the temporal constraint applies recursively downward: a subtask inherits the most restrictive of its own constraints and those of its parent. A method violating its temporal constraint will yield zero quality. The TÆMS model also supports representing dependencies among tasks, such as enablement, disablement, facilitation, and hindrance, with *non-local effect* (NLE) links (Lesser et al., 2004). A NLE link indicates a task interrelationship where the execution of some task will have a positive or negative effect on the quality or duration of another task. Given a problem represented in a TÆMS model, the objective is to find a policy maximizing the expected quality at the root node of the model.

Figure 5.3 shows an example TÆMS model with 21 nodes, which we will use to illustrate our heuristic search techniques. The temporal constraint of a node is represented in the format $[t_a, t_b]$, where t_a and t_b represent the release time and deadline of the node, respectively. The description of the temporal constraint (if there is one) is placed under the node's name. In this example, a single method might have multiple possible outcomes. The distribution on the quality of a method outcome is depicted in the format $Q : a \pm b$, which implies that the method has probability 0.5, 0.25, and 0.25 of yielding the expected quality a, minimum quality $a - b$, and maximum quality $a + b$, respectively. The duration distribution of the method outcome is depicted in a similar format $D : a' \pm b'$, which says that the method will complete at its expected duration a', minimum duration $a' - b'$, and maximum duration $a' + b'$ with probability 0.5, 0.25, and 0.25, respectively. There is an *enable* NLE in this example, which indicates that task *Ctask3b* can yield a positive quality only when task *Ctask2b* has achieved a positive quality first.

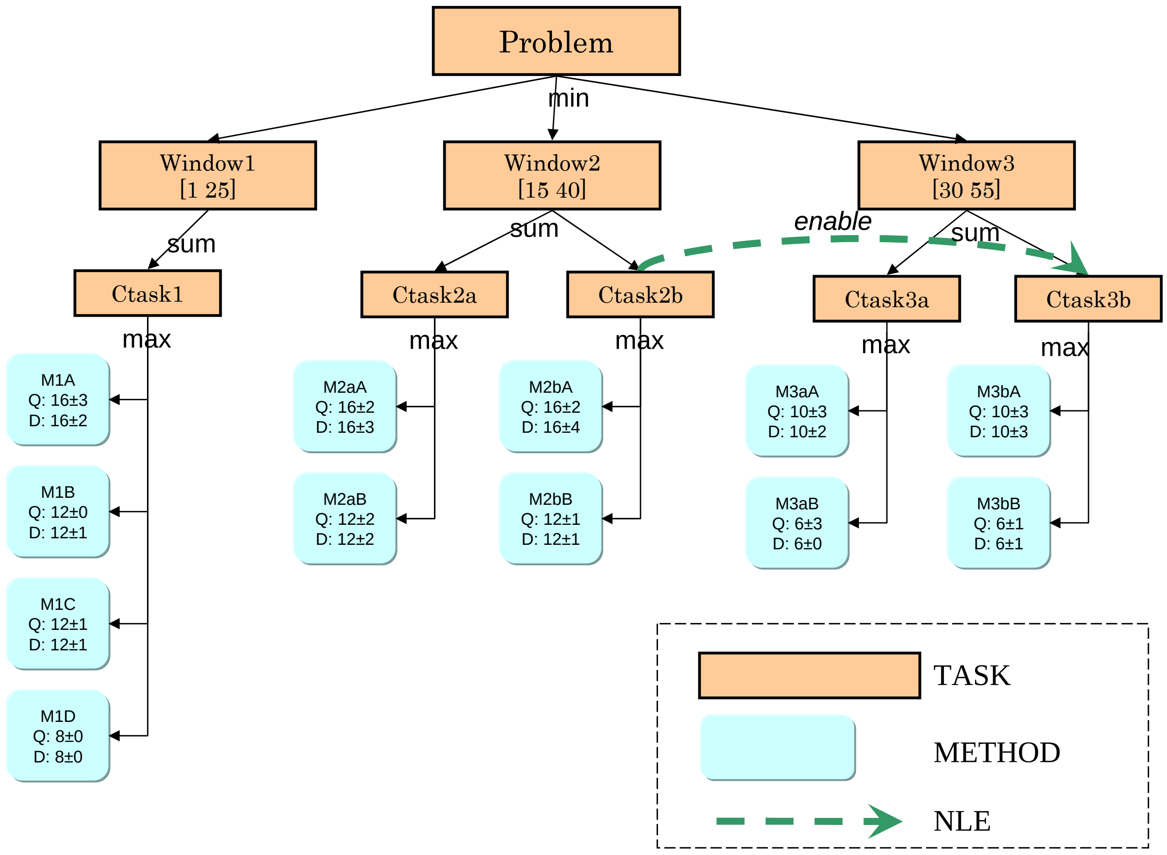


Figure 5.3: A simple example TÆMS model with 9 tasks, 12 primitive methods, and 1 enablement NLE.

5.3.2 Unrolling TÆMS Models into MDPs

Markov decision processes provide a good framework to compute optimal policies in uncertain environments, and thus are well suited for TÆMS problems having uncertainties in qualities and durations of method outcomes. This subsection recaps the prior approach for fully unrolling a TÆMS model into a MDP (Wagner et al., 2006; Musliner et al., 2006), whose solution policy will maximize the expected quality at the root of the TÆMS model, because our techniques extend these foundations.

In the TÆMS MDP, a state is defined as $\langle t, \mathbb{M} \rangle$, where t is the current time, and $\mathbb{M}$ is a set of method outcomes $\{o\}$. Each outcome $o = \langle m, \tau, d, q \rangle$ stores the information of its execution method m, start time τ, duration d, and quality q. Such a state representation assures that the conditional probability distribution of future states, given the present state and all past states, depends only upon the current state and not on any past state, i.e., the Markov property holds. When an agent executes a method m_i, which has probability p_i of taking d_i time steps and yielding quality q_i, in state $\langle t, \mathbb{M} \rangle$, the agent will reach the successor state $\langle t + d_i, \mathbb{M} \bigcup \{\langle m_i, t, d_i, q_i \rangle\} \rangle$ with probability p_i.

The unrolling procedure is summarized in Procedure 1. It is similar to an (uninformed) breadth-first search. At each loop, it pops the top state from the queue *openList* (line 4), and finds the set of applicable methods that can be executed in that state by examining their temporal and NLE constraints in the model (line 5). It then generates successor states for each applicable method according to state transition functions described above (line 6), updates the MDP (line 8), and puts newly generated, non-terminal successor states (those for times prior to the problem horizon) at the end of *openList* (line 10).

The unrolling procedure stops when *openList* is empty. At that point, the TÆMS

Procedure 1 mdp = **unroll**($model$)

```
 1: Initialize empty $mdp$, $initState$
 2: $openList \leftarrow \{initState\}$
 3: repeat
 4:   $state \leftarrow$ dequeue($openList$)
 5:   for all $m \in$ applicable-methods($state$, $model$) do
 6:     $succs \leftarrow$ successor-states($state$, $m$, $model$)
 7:     for all $succ \in succs$ do
 8:       $mdp \leftarrow$ update($state$, $m$, $succ$, $mdp$)
 9:       if $succ$ is not a terminal state then
10:         enqueue($succ$, $openList$)
11:       end if
12:     end for
13:   end for
14: until $openList$ is empty
15: return $mdp$
```

model is fully unrolled into a finite horizon MDP. Each MDP terminal state captures method outcomes of a possible execution trajectory of the problem, and the reward of the state is the quality of the root node of the TÆMS model given that execution trajectory. On the other hand, the internal states in the MDP state space always have zero reward, since all activities will be evaluated at the end of the execution (i.e., in terminal states). Given state transition probabilities and rewards, the backward induction algorithm can be used to generate an optimal policy in time linear in the number of states (Puterman, 1994).

5.3.3 Informed Unroller

Many application domains, such as the Coordinator domain (Musliner et al., 2006), are complicated, and often lead to very large state spaces. The above full (uninformed) unrolling procedure may not be practical for these complex problems, especially when computation time is limited. A straightforward solution to this time-limitation challenge is to stop the unrolling procedure at a pre-specified time point. Thereafter, rewards are computed for edge states of the partially unrolled state space, based upon the (partial) execution trajectories represented in those states, and the backward induction algorithm is used to derive a policy for the partially unrolled state space.

However, this straightforward approach suffers from two drawbacks. First, the limited computation time only allows a subset of states to be expanded and explored. The unrolling procedure described in Section 5.3.2 implements an uninformed breadth-first style expansion, which will expand all paths to equal (partial) depth regardless of the chance of the agent traversing the path. Second, the approach does not consider and specify actions to take after the agent executing a policy goes beyond the partially unrolled state space. Intuitively, randomly choosing an action (when out of the policy) is unlikely to be an effective way in accruing quality.

To address the first drawback mentioned above (the second will be discussed later), our informed-unroller (IU) algorithm prioritizes the queue of states waiting to be unrolled based on an estimate of the likelihood that the state would be encountered when executing an optimal policy from the initial state. Because the probability of reaching a state is dependent on the policy, the IU algorithm intersperses policy formulation (using the backward induction algorithm) with unrolling. It should be noted that, although the backward induction algorithm is fast (i.e., its runtime is linear to the number of states), formulating a policy at each state expansion step is generally too costly. To balance the benefits of unrolling more states and of being more directed in the unrolling direction, the IU algorithm recomputes a policy and reorders the states waiting to be expanded less frequently. The empirical results presented in this work are based upon a heuristic to sort the queue of states waiting to be expanded when the size of the unrolled state space has doubled since the last sorting.

The details of the IU algorithm are shown in Procedure 2, where lines $6 - 15$ are similar to the aforementioned uninformed unroller algorithm except that line 12 inserts the newly generated successor state in a new way described below (instead of always placing it at the end of *openList*).

Let state i represent the state being expanded, and let state j represent one of its successor states. If the reaching probability of state i is P_i, then the reaching probability of the new successor state j is estimated as:

$$P_j = \frac{P_i \times \max_a p_{i,a,j}}{|A_i|}$$

where $p_{i,a,j}$ represents the state transition probability function, and $|A_i|$ represents the number of applicable actions at state i.

That is, the estimated reaching probability of the new successor state is the reaching probability of its ancestor state (that leads to the new successor state) multiplied by the maximum transition probability from the ancestor state to the new successor state, followed by a discount factor $1/|A_i|$. According to P_j, the new successor state is inserted into *openList* while keeping *openList* sorted in descending order of estimated reaching probabilities. Since the IU algorithm does not sort *openList* at each iteration, this insertion process is a helpful, supplementary mechanism to help the agent emphasize the exploration of the promising subset of the state space (that is believed to be reached with high likelihood by high-quality policies) between *openList* sorting procedures.

Line 17 in Procedure 2 evaluates edge states of the partially unrolled state space by building and evaluating greedy plans starting at the edge states; the details will be discussed later (in Procedure 3). Line 18 solves the MDP by using the backward

Procedure 2 mdp = **informed-unroll**($model$, $tlim$)

```
Initialize empty $mdp$, $initState$
$preSortSize \leftarrow 10$
$K \leftarrow 2$
$openList \leftarrow \{initState\}$
repeat
  $state \leftarrow$ dequeue($openList$)
  for all $m \in$ applicable-methods($state$, $model$) do
    $succs \leftarrow$ successor-states($state$, $m$, $model$)
    for all $succ \in succs$ do
      $mdp \leftarrow$ update($state$, $m$, $succ$, $mdp$)
      if $succ$ is not a terminal state then
        insert($succ$, $openList$, $mdp$)
      end if
    end for
  end for
  if size($mdp$) $\geq K \times preSortSize$ then
    $mdp \leftarrow$ eval-edge-states($mdp$, $model$)
    $policy \leftarrow$ solve($mdp$)
    $prob \leftarrow$ compute-reaching-prob($mdp$, $policy$)
    $openList \leftarrow$ sort($openList$, $prob$)
    $preSortSize \leftarrow K \times preSortSize$
  end if
until $openList$ is empty or runtime reaches $tlim$
return $mdp$
```

induction algorithm to derive a policy, and line 19 computes the probability of the agent reaching each edge state when executing the derived policy. Edge states waiting to be expanded are sorted in line 20 according to their probabilities of being reached, where the state with the highest reaching probability is placed at the top of the queue, i.e., in a highest-probability-first manner.

To build an intermediate policy that can properly guide the IU exploration direction, potential activities in the future should be considered when evaluating edge states. An exact evaluation of an edge state requires a full look-ahead, and is obviously impractical. Instead, the IU algorithm computes the heuristic value of an edge state by adopting a fast greedy algorithm to build a plan starting at that edge state.[3] Unlike a policy considering all possible eventualities, a plan only represents a particular execution trajectory. Specifically, a plan is composed of deterministic copies of TÆMS methods. Each deterministic method (deterministic because it has exactly one outcome) corresponds to one actual TÆMS method; its duration is the maximum duration of the TÆMS method and its quality is the expected quality of the TÆMS method.

As stated, these greedy plans can serve two purposes. First, given an edge state i and a greedy plan p starting at it, the IU algorithm can predict a unique terminal state i' at the end of the execution of the plan p (i.e., by adding deterministic outcomes of the methods represented in the plan p into the state i). The quality of this terminal state i' can then be fed back to estimate the state value of the edge state i. Considering possible improvement in the further state exploration, the IU

[3]In some domains, problems come with some (suboptimal but good) initial solutions, and we can use these solutions to evaluate edge states instead.

Procedure 3 $[plan, state]$ = **greedy-plan**$(state, model)$

```
1: Initialize an empty $plan$
2: while $state$ is not a terminal state do
3:   $\overline{method} \leftarrow$ best-method$(state, model)$
4:   $plan \leftarrow$ add$(\overline{method}, plan)$
5:   $state \leftarrow$ successor-state$(state, \overline{method})$
6: end while
7: return $plan, state$
```

algorithm sets the heuristic value of an edge state i to:

$$f(i) = qual(i) + K(t) \times (qual(i') - qual(i)) \tag{5.1}$$

where $qual(i)$ and $qual(i')$ indicate the qualities accumulated by the completed methods modeled in state i and i', respectively, and $K(t)$ is a decreasing function. In empirical results shown in this chapter, a linearly decreasing function is adopted, which is initially set to the ratio of the average maximum duration of all applicable TÆMS methods to the average expected duration of those methods (because deterministic methods in the greedy plan take their maximum durations instead of expected durations), and then gradually decreases to 1.0 as computation time approaches its limit.

Second, these plans can tell the agent what to execute (though maybe suboptimally) after the agent reaches the edge of the partially unrolled space during execution. Recall that a deterministic method uses the maximum duration of its corresponding TÆMS method, and so the plan composed of deterministic methods is always executable, i.e., the agent can simply wait if a TÆMS method completes before its maximum duration.

The details of generating a greedy plan are presented in Procedure 3. Given an edge state, the procedure starts with an empty plan (line 1), and then gradually

Procedure 4 $\overline{method}$ = **best-method**(*state*, *model*)

```
1: $\hat{m} \leftarrow$ applicable-det-methods($state$, $model$)
2: if $\hat{m}$ is empty then
3:   return $wait$
4: end if
5: $\hat{m} \leftarrow \text{argmax}_{m \in \hat{m}}$ quality(successor-state($state$, $m$), $model$)
6: $\hat{m} \leftarrow \text{argmax}_{m \in \hat{m}}$ is-in-unexplored-branch($state$, $m$, $model$)
7: $\hat{m} \leftarrow \text{argmax}_{m \in \hat{m}}$ $-1 \times$deadline($m$)
8: $\hat{m} \leftarrow \text{argmax}_{m \in \hat{m}}$ method-success-prob($m$)
9: $\hat{m} \leftarrow \text{argmax}_{m \in \hat{m}}$ qual-density($m$)
10: return one-of($\hat{m}$)
```

augments the plan by myopically adding deterministic methods until reaching the problem horizon (lines 2 – 6).

The heuristic used to decide which deterministic method to be inserted into the plan is described in the *best-method* procedure shown in Procedure 4, where the function "argmax" stands for the arguments of the maximum, i.e., the set of arguments for which the value of the given expression attains its maximum value. The best-method procedure begins by checking applicable deterministic methods (line 1). If none of the methods can be executed, it returns a *wait* method that will let the agent idle one time step (lines 2 – 4). Otherwise, given a set of applicable methods, the procedure chooses the method(s) that can lead to the successor state(s) with the highest quality (line 5). If multiple deterministic methods tie, the heuristic picks the method(s) located in an unexplored branch where an unexplored branch refers to a sub-model rooted at a TÆMS node that is directly under a *min* QAF and has zero quality so far (line 6). If tied again, the methods are, in turn, filtered by their deadlines (line 7), success probabilities (i.e., the probability of successfully completing a method while ignoring its temporal constraints) (line 8), and quality densities (i.e.,

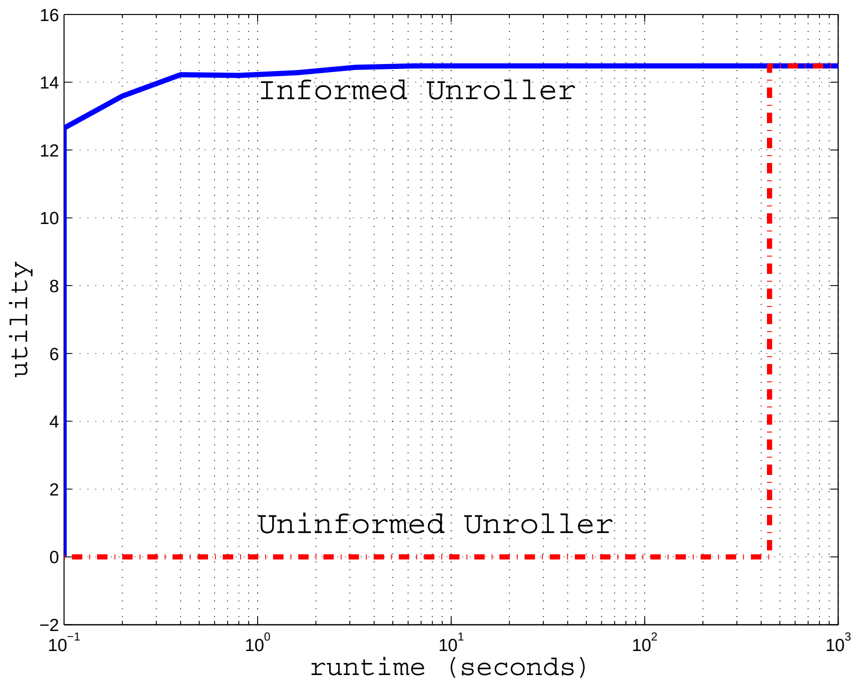


Figure 5.4: Comparison of the runtime between the uninformed unroller and the informed unroller on the example. The informed unroller finds an approximately optimal solution within 1 second and finds an optimal solution within 10 seconds, while the uninformed unroller takes 442 seconds to find a complete, optimal solution.

the quality divided by the duration) (line 9).

This section concludes by comparing the uninformed unroller (UU) algorithm (described in Section 5.3.2) and the informed unroller (IU) algorithm (described in this section) on the example problem depicted in Figure 5.3. As shown in Figure 5.4, the informed unroller finds an (approximately) optimal policy considerably faster than the uninformed unroller. Specifically, the informed unroller finds an approximately optimal solution within 1 second and finds an optimal solution within 10 seconds, while the uninformed unroller needs 442 seconds to find an optimal solution.

To provide a better understanding of state expansion and exploration behaviors, Figure 5.5 summarizes the states unrolled by these two solvers when the unrolling time is limited to 10 seconds for either of them. As illustrated, the uninformed unroller unrolls a larger state space than the informed unroller, but its exploration

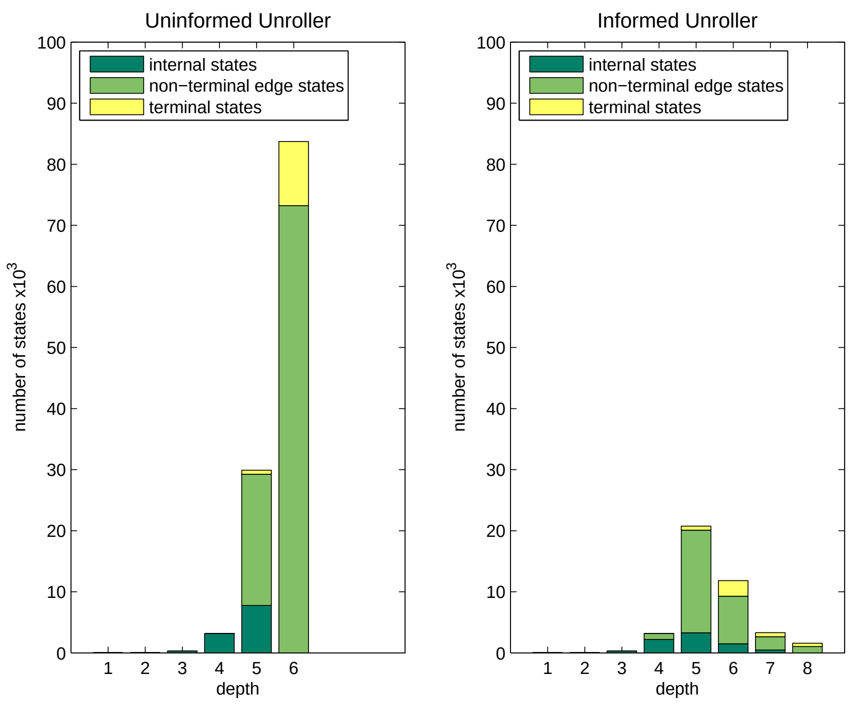


Figure 5.5: Comparison of the state space expansions between the uninformed unroller and the informed unroller on the example. The informed unroller explores deeper than the uninformed unroller although the informed unroller expands fewer states (where time limit is 10 seconds).

depth toward the finite horizon is shallower. This is as expected. The breadth-first style search of the uninformed unroller results in exploring all paths to equal depth regardless of the probability that the agent will traverse the path (which is unlikely to be an effective way when the number of states that can be generated is restricted). In contrast, although the informed unroller explores a smaller number of states (because it spends much of its computation time building greedy plans at edge states and computing intermediate policies to guide further exploration), it is able to explore deeper due to its selective search strategy of focusing on states with higher probability of being reached in high-quality solutions and ignoring many other states with low or zero reaching probabilities.

5.4 Experimental Evaluation

As discussed above, the IU algorithm is capable of finding a high-quality solution within a short time, which makes it a potentially promising approach for time-critical applications. This section helps test this claim through evaluating and comparing the IU algorithm with prior effective heuristic search algorithms.

Our tests are based upon the Coordinator project (Musliner et al., 2006). The Coordinator project researches real-world multi-agent coordination problems, and has gained much attention in recent years (Musliner et al., 2006; Raja et al., 2006; Emami et al., 2006; Zhang and Xu, 2006). The empirical results shown in this section are collected from 16 *3-agent* and 22 *4-agent* problems from the Coordinator project[4], which are divided into 136 single-agent TÆMS problems. The computational complexity of these problems varies. The simplest test problem has only 26 nodes in its TÆMS model (that leads to several thousands of MDP states), but the most complex one has 155 nodes (that leads to tens of millions of MDP states), which provides a diverse test set to evaluate TÆMS solvers under a wide variety of situations.

We compare our IU algorithm with the AO* algorithm and the RTDP algorithm, which were introduced in Section 5.1.[5] To improve computational efficiency of the AO* algorithm, instead of revising cost at each iteration, our implementation of the AO* algorithm, in a similar way to (Hansen and Zilberstein, 2001), only revises costs and updates action choices once all of the edge states at the last cost-revision point have been expanded. We have also implemented and evaluated some variations of the RTDP algorithm, including the bounded RTDP and focused RTDP al-

[4]These problems are devised and used by a large research team (of which we are part) centered at Honeywell Labs.

[5]The envelope algorithm is not implemented and evaluated because it is not directly applicable to problems without explicit goal states.

gorithms which have been shown to have better anytime performance in *stochastic-shortest-path* problems. However, in our problems (that typically have very large state spaces), none of these variations outperforms the RTDP algorithm on average, which we believe is mainly because the large size of the state space makes the upper and lower bounds of states difficult to converge.

Like many heuristic algorithms, the heuristic evaluation functions used are critical to the performance of heuristic-search-based policy formulation methods. The work is this dissertation compares the performance of the IU, AO*, and RTDP solvers under three different heuristic functions described below.

***greedy-plan* heuristic:** a non-admissible heuristic that estimates state values through building and evaluating greedy plans at edge states. The detail was described in Eq. 5.1.

***max-qual* heuristic:** an admissible heuristic that estimates the value of an edge state based upon the assumption that all current and future applicable methods (denoted as $\mathbb{M}$) can be successfully completed by the agent starting at that edge state:

$$f(i) = qual(state(i, \mathbb{M}))$$

***constant-value* heuristic:** a fast heuristic that returns the sum of a constant value *42* and the quality accumulated so far at the edge state:[6]

$$f(i) = qual(i) + 42$$

Empirical results are shown in Figure 5.6. The first column indicates the names of the heuristic search solvers. The second column specifies the heuristic evaluation

[6]The value of 42 was chosen as an arbitrary constant, because this was the answer given by the computer in The Hitchhiker's Guide to the Galaxy in response to being asked the meaning of life, the universe, and everything.

Algorithm	Heuristic Evaluation Function	Quality		#States (thousands)
		Case I: No Computation during Execution	Case II: Limited Computation during Execution	
IU	*greedy-plan*	136.7	138.4	106.3
AO*		124.1	127.2	140.4
RTDP		118.8	120.5	138.9
IU	*max-qual*	73.1	86.1	177.7
AO*		67.7	85.1	210.4
RTDP		74.5	86.5	224.2
IU	*constant-value*	74.3	91.4	224.1
AO*		69.2	87.1	270.1
RTDP		73.8	90.6	272.5

Figure 5.6: Optimality comparison among IU, AO*, RTDP, and their variations.

functions used to evaluate edge states. The third column denotes the average solution qualities over the aforementioned 136 test problems when computation time is limited to 100 seconds, based upon the assumption that the agent has no computation power in the midst of execution and thus it simply follows the greedy plan (if feasible) or randomly picks up applicable methods when running out of the partial policy. The fourth column is similar to the third one except that it is assumed that the agent now has limited computation power and so it can invoke the *best-method* procedure (Procedure 4) to find and execute a myopically optimal method. The last column indicates the numbers of states expanded within the 100-second unrolling time.

Adopting an admissible heuristic function is a preliminary condition for the AO* and RTDP algorithms to find optimal solution policies. However, as shown in Figure 5.6, in situations where computation time is limited, the admissible *max-qual* heuristic does not perform well. The underlying reason is that the state space of a TÆMS MDP is often very large, such that even with 100 seconds the heuristic search algorithms can only search a small fraction of the reachable state space. The max-qual heuristic optimistically assumes that all future applicable methods can be accomplished; this overestimation of the state values encourages the agent to schedule high-quality methods in its partial policy (regardless of the method completion times). High-quality methods often have long durations, which means that scheduling high-quality methods for the early stage may squeeze time windows for executing future methods. In other words, the max-qual heuristic may make the agent achieve high quality in the tasks that are modeled in the expanded state space, but may result in low quality in the future tasks (that have not been explored yet). Note that many Coordinator problems have complex reward structures in which the quality of a task may be the minimum quality of its subtasks. In such situations, the max-qual heuristic might severely impair the performance of the agent because of the low quality of the future tasks.

The results also show that the performance of the constant-value heuristic is poor. This is because, although the constant-value heuristic can evaluate edge states very quickly (and thus it results in the largest explored spaces among the three heuristic evaluation functions), the algorithms using this heuristic are often not correctly guided to promising areas in the midst of search since this heuristic overlooks the differentiation of potential activities that the agent can perform when starting at various edge states.

Among all three heuristic evaluation functions, the greedy-plan heuristic is the best, particularly in situations where the agent has no computation power in the midst of execution. The greedy-plan heuristic requires the agent to spend effort building greedy plans during the heuristic search procedure, which, without surprise, reduces the number of states that the agent can expand. However, as shown in the results, it is worthwhile to do this. These plans provide a good knowledge about potential promising exploration areas. In other words, when the agent needs to determine policy actions in its partially unrolled state space (for AO* and IU) or in the visited trials (for RTDP), it can have a better understanding about potential activities in the future, which in turn means that the agent can be smarter in selecting which subset of states to explore. As a result, within the limited time, the agent can formulate a better partial policy despite searching fewer states. Moreover, typically, these plans are better than randomly selecting methods, which explains why the greedy-plan heuristic considerably outperforms the others when the agent has no computational power during execution, e.g., the IU algorithm gets a quality 136.7 using the greedy-plan heuristic, which is 87% higher than the quality 73.1 using the max-qual heuristic, and 84% higher than the quality 74.3 using the constant-value heuristic.

We conclude this section with a detailed look at the anytime performance of the aforementioned three heuristic search algorithms using the greedy-plan heuristic (that is the best). Figure 5.7 displays the average results collected from the 136 test problems. Its x-axis indicates the runtime, and its y-axis represents the average solution quality. Clearly, the IU algorithm outperforms the RTDP algorithm and the AO* algorithm. We believe that the reason for IU being better than RTDP is that RTDP runs trials to the problem horizon, and so it may spend much computational effort in building greedy plans for states that are far away from the initial state. Because of the large branching factor, these states have only a small prob-

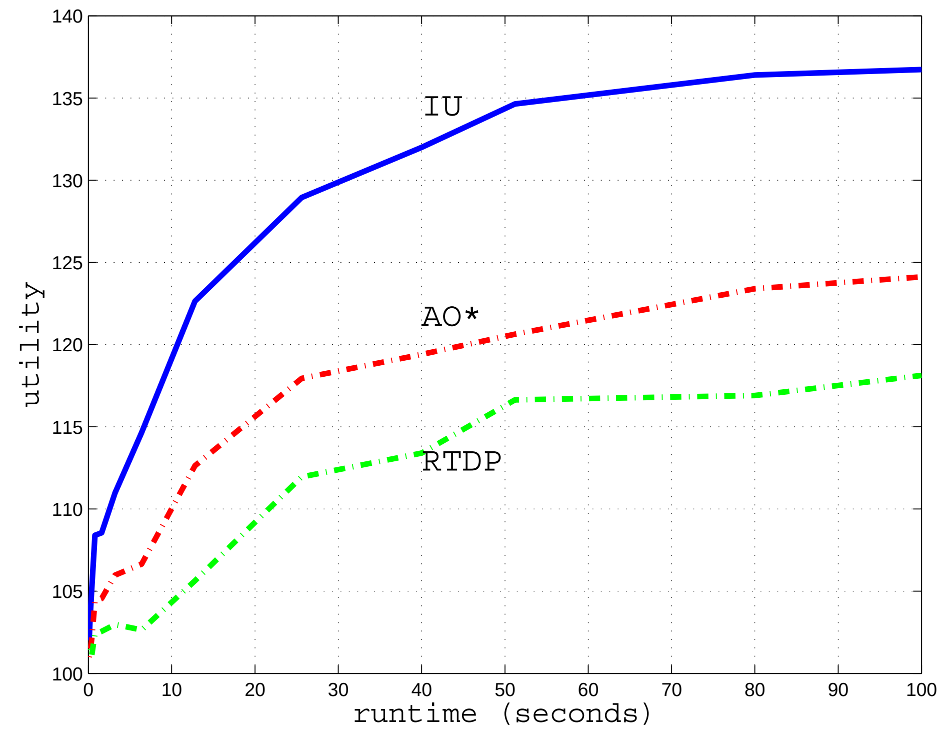


Figure 5.7: Anytime performance comparison among the IU, AO*, and RTDP algorithms.

ability of being reconsidered and reused during trial runs and a small probability of being reached during execution, which makes the effort spent in those states less contributive in deriving a good partial policy. On the other hand, the reason for IU being better than AO* is that although AO* can focus its computational effort on its partial solution graph (instead of considering all reachable states like the uninformed unroller algorithm), the AO* algorithm does not explicitly and intentionally differentiate edge states in its partial solution graph like the IU algorithm does (i.e., probability-first search), mainly because the AO* algorithm was originally designed as an off-line policy formulation algorithm for deriving optimal solutions.

In sum, unlike the RTDP algorithm that runs trials to the problem horizon, which may lead to a long but narrow exploration area, and unlike the AO* algorithm that treats edge states equally, which may lead to a wide but short exploration area, we believe that the IU algorithm strikes a good balance between the width and the depth of its explored state space, and thus it is able to find a high-quality solution faster.

5.5 Computation-Driven Mission-Phasing for Large TÆMS Problems

As shown in the previous section, for the problems with large state spaces but with limited computation time, the IU algorithm can often find a better solution than the prior heuristic search algorithms, which makes the IU algorithm a promising anytime policy formulation algorithm. However, it should be noted that the IU algorithm still suffers from several drawbacks. First, it does not consider the problem structure that may be exploited to reduce the total number of MDP states. Second, it is unable to effectively exploit computation time available in the midst of execution to improve its solution. Third, it cannot determine how much time it should use for its policy formulation procedure in environments where an agent can choose to pay for more computation time. Fortunately, all the above drawbacks can be overcome by our computation-driven mission-phasing approach where the IU algorithm works with a problem decomposition method (that will be presented in the next subsection) and the deliberation scheduling algorithm (that was presented in the last chapter) to further improve the performance of time-limited agents.

This section is organized as follows. It starts by presenting a heuristic decomposition method in Section 5.5.1, and then describes a way of constructing performance profiles of the IU algorithm in Section 5.5.2. The reader who is not interested in these details may go directly to the experimental results in Section 5.5.3 where we empirically evaluate the overall computation-driven mission-phasing approach.

5.5.1 Heuristic Decomposition

Much research work has been devoted to speeding up planning by breaking problems into sequences of sub-problems, such as the "landmarks" approach (Porteous et al., 2001) and the "doorway" decomposition heuristic (Parr, 1998). These decomposition techniques exploit "weak connections" points between parts of a large problem, which can often result in a significant reduction in the computational cost for finding an optimal or an approximately optimal solution. In this subsection, we describe a TÆMS model heuristic decomposition method, based upon a similar idea.

The decomposition method partitions the time horizon of a large TÆMS problem into several disjoint time windows, each of which corresponds to a smaller TÆMS MDP problem. The implementation is outlined in Procedure 5. It begins by examining the given model to determine nodes that will be used as the root nodes of phase models (line 2). The heuristic adopted in this work is to choose nodes at the deepest level of the model where some nodes have specified release times and/or deadlines (because the decomposition method will adjust these temporal constraints to make phase problems independent). In cases where the resulting phases are too large (i.e., with more than 10^5 states), nodes at a deeper level will be selected. After that, for each of the selected nodes, the algorithm constructs a phase problem corresponding to the sub-model rooted at that node (line 5). Phases are then sorted according to deadlines of root-nodes of their models to get ready for merging (line 8).

In order to keep interactions and dependencies in the TÆMS task network, for each NLE, the decomposition method merges the phase containing the NLE's source node, and the phase containing the NLE's destination node, as well as all phases between them (line 9).[7] Of course, in a highly (NLE) connected problem, this might

[7]An alternative way is to use internal commitments, which is one of our future research directions.

Procedure 5 $phases$ = **heuristic-decompose**($model$, $tlim$)

```
1: Initialize empty $phases$
2: $nodes$ = **find-phase-root-nodes**($model$)
3: **for all** $node \in nodes$ **do**
4:   Initialize $phase$
5:   $phase.model \leftarrow$ **sub-model**($node$, $model$)
6:   $phases \leftarrow$ **add**($phase$, $phases$)
7: **end for**
8: $phases \leftarrow$ **sort-phases**($phases$)
9: $phases \leftarrow$ **merge-nle-linked-phases**($phases$)
10: $phases \leftarrow$ **merge-small-phases**($phases$)
11: $phases \leftarrow$ **split-overlap-by-hill-climbing**($phases$)
12: **return** $phases$
```

lead to phase problems with large TÆMS models. However, it can still rely on the IU algorithm to find a good solution to a large phase MDP problem within a short time. The decomposition method also attempts to merge several small neighboring phases into a moderate-size phase (where the resulting phase would not exceed 10^4 states) (line 10). This is because a moderate-size phase can still be solved efficiently by the IU algorithm while it can maintain (most of) the temporal constraints of the tasks within the merged phases.

The step after merging phases is to determine temporal boundaries of the resulting phases. In the work presented in this dissertation, the heuristic decomposition method starts with a straightforward way of evenly splitting time-window overlaps between relevant phases, and then implements a hill-climbing procedure to improve the time-window splits (line 11). The detail of the hill-climbing procedure is described in Procedure 6.

At each hill-climbing iteration, the method spends a predefined amount of time $tlim'$ (that is 1% of the amount of time available prior to execution or a constant 0.1

Procedure 6 $phases$ = **split-overlap-by-hill-climbing**($phases$)

```
1: $phases \leftarrow$ **evenly-split-overlap**($phases$)
2: **repeat**
3:   **for all** $phase \in phases$ **do**
4:     **if time-window-changed**($phase$) **then**
5:       $phase.mdp \leftarrow$ **informed-unroll**($phase.model$, $tlim'$)
6:     **end if**
7:   **end for**
8:   $phase \leftarrow$ **find-phase**($phases$)
9:   $phase \leftarrow$ **enlarge-time-window**($phase$)
10: **until** a local maximum is found or the number of iterations exceeds $K$
11: **return** $phases$
```

second, whichever is smaller, in our implementation) running the IU algorithm for each phase whose time window has been changed in the last iteration (lines $3-7$). Thereafter, the decomposition method chooses a phase that appears to be able to improve the overall solution quality most by enlarging the time window of that phase (line 8). In detail, this *find-phase* procedure starts at the root-node of the original overall TÆMS model. It selects a subtask node with the highest quality density if the root-node's QAF is *sum* or *max*, and selects a subtask node with the lowest quality if the root-node's QAF is *min*. If the selected node corresponds to a phase, the procedure returns that phase. Otherwise, it repeats the above procedure from the selected subtask node until a phase is found. After a phase is selected, the decomposition method enlarges the time window of that phase by tightening time windows of its neighboring phases (line 9). The hill-climbing procedure repeats the above time-window revision iteration until a local maximum is found or the number of iterations exceeds a predefined constant K ($K = 50$ in our implementation). At this point, the problem represented in a large TÆMS model has been decomposed into a sequence of smaller phase problems corresponding to independent sub-models with disjoint time windows.

It should be pointed out that, using the heuristic decomposition method, the agent may fail to derive an optimal overall policy to the original problem model even given unlimited computation time. This is due to two reasons. First, splitting time-window overlaps may result in stricter temporal constraints on the participant tasks, which restrict the state and action space of the problem. Second, due to the nonlinearities in some QAFs (such as *min*), the combination of optimal phase policies may not be an optimal solution to the overall problem. The reason is that maximizing a nonlinear function of expected values does not necessarily maximize the expected value of the nonlinear function, e.g., $\max\min(E(x), E(y)) \neq \max E(\min(x, y))$ where $E()$ is the expectation function, and x and y are random variables.

However, in time-critical environments, the decomposition method is of value. Problem decomposition results in smaller state spaces, which could mean that the agent can find a high-quality solution faster. In the empirical results shown at the end of this chapter, we will see that the combination of the decomposition method and the IU algorithm can help the agent build a better solution within limited time than the IU algorithm by itself.

5.5.2 Constructing Performance Profiles

A follow-on problem of decomposing a large problem model into multiple phase models is that the decision-making agent would need to cope with more than one decision procedure (one for each phase). Despite the reduced size of the state space (due to decomposition), in time-critical situations formulating a complete and optimal policy for each phase might still be impractical (because typically an effective decomposition method breaks a problem only at weakly-connected points and so the resulting sub-problems might still be moderately large). Using an anytime policy formulation solver (such as the IU algorithm) is only part of the answer. There is also the need for the agent to judiciously distribute its limited computation time among multiple phases.

In general, the utility of time allocation is dependent on two factors: the accuracy of performance profiles used to predict the solution quality of decision procedures, and the optimality of the deliberation schedule built upon those performance profiles. Chapter IV has presented a mathematical-programming-based deliberation scheduling algorithm that can quickly make an optimal or near-optimal allocation of the limited computation time, based upon the given performance profiles. We here discuss the remaining challenge: how to construct accurate performance profiles. We illustrate our approach on the TÆMS MDP problems introduced before, but the ideas can also apply to other similar problem domains.

As described in Section 4.2 and Section 4.4, performance profiles are used to characterize the expected performance of decision procedures for varying computation-time allotments. They give the agent some prior knowledge about the problems it is to solve. Therefore, in situations with limited computation time and with more than one decision procedure, the agent can predict the expected performance of each decision procedure, and then, based upon these predictions, judiciously distribute its time over those decision procedures.

The fundamental insight of using performance profiles is that similar problem instances would have similar runtime performance, which means that the definition of the "similarity" of problem instances affects the accuracy of performance profiles. A simple construct of performance profiles that directly maps runtime into the expected quality of the solution derived within that amount of time (i.e., assuming all problem instances in the running domain are similar) is often insufficient to give a good prediction of how the decision procedure performs because some problem instances may be considerably more challenging than others even in the same application domain. That is to say, a good construct of performance profiles must take into account problem features that can greatly affect computational complexity.

For the TÆMS test problems used in this chapter, several features of the TÆMS models are considered when constructing performance profiles. The details are discussed below. The empirical results show that our construct of performance profiles performs reasonably well (although we do not argue that it is the best way for all general problems).

In detail, the performance profile of a phase decision procedure is conditional on the following inputs besides the runtime t:

- ▷ An integer l, which corresponds to the size of the phase state space. We say that a phase problem has complexity level l if the estimated size of its state space (that is computed as $(b)^d$ where b is the average number of method outcomes, and d is the duration of the time window of the phase divided by the average expected duration of methods in that phase) falls in the range $[1000 \times l, \ \ 1000 \times (l+1)]$.

- ▷ An integer o, which characterizes time-window overlaps (i.e., temporal constraint overlaps) of tasks in the phase problem. Specifically, a phase problem has overlap level o if the ratio of the average overlap "width" to the average time-window "width" is in the range $[0.1 \times o, \ \ 0.1 \times (o+1)]$.

- ▷ An integer f, which indicates how many alternative methods the agent has for a task on average.

It should be pointed out that the quality of TÆMS methods may be arbitrarily different from problem to problem. To meet this challenge, our construct of performance profiles is based upon the *normalized* solution quality (instead of the actual quality). That is, the expected quality of the solution generated with a predefined small amount of runtime τ (τ is set to *0.1* seconds in our implementation) is normalized to a constant *1.0* (and the solution quality at other time points is scaled in the same proportion). According to our experience, this normalized performance profile, together with a preliminary online examination, can better predict the performance of the phase decision procedure than a standard performance profile.

The detailed procedure of generating and using such normalized performance profiles in our test domain is described below.

- ▶ ***In the off-line stage:***
  - ◇ 36 problems (out of the 136 test problems introduced in Section 5.4) are randomly selected, and are decomposed into $7,782$ phase problems using a decomposition method similar to that presented in Section 5.5.1 (but NLEs may be arbitrarily added or removed to generate more test problems).
  - ◇ These phase problems are categorized into groups according to their features $\mathbb{F}$ (including l, o and f introduced above). All phase problem instances in the same group have the same features, and their normalized solution qualities over time t are averaged and stored in a function $v = V_{\mathbb{F}}(t)$.
- ▶ ***In the online stage:***
  - ◇ Run the decomposition method presented in Section 5.5.1 to decompose the input problem into multiple phases.

- ◇ For each phase i, run the IU algorithm for a short period of time τ (defined above) to derive a preliminary solution, whose solution quality (denoted as Q_i) will be combined with the normalized performance profile to generate a standard performance profile in the next step.
- ◇ For each phase i, find the corresponding normalized performance profile (built in the off-line stage) according to the phase's features $\mathbb{F}_i$, and compute its standard performance profile as $V_i(t) = Q_i \times V_{\mathbb{F}_i}(t)$.
- ◇ Compute a deliberation schedule using the deliberation scheduling approach presented in Chapter IV, based upon the phases' performance profiles $V_i(t)$ (and deliberation cost functions $C_i(\tau)$ if applicable).
- ◇ Follow the derived deliberation schedule to build phase policies with the IU algorithm. Note that performance profiles provide predictions, which means that they may sometimes deviate from the actual results. To cope with this issue, in the procedure for formulating policies (or even in the procedure for executing policies if the agent can reconsider its solution in the midst of execution), the agent can choose to re-run the deliberation scheduling algorithm with the updated information when the actual performance of the solution deviates a lot from the prediction, and then follow the new deliberation schedule for policy formulation.[8]

The above procedure is indeed the computation-driven mission-phasing procedure, which incorporates informed unrolling, deliberation scheduling, and problem decomposition. As will be shown in the evaluation section, this CMP procedure can make more effective usage of the limited computation time than the pure informed

[8]In this work, this repair step is not implemented in the experiments since the performance profiles we constructed and used predict solution qualities reasonably well.

unrolling procedure and other prior approaches.

5.5.3 Experimental Evaluation

This subsection empirically evaluates the efficacy of our computation-driven mission phasing approach in the same test domain as used to evaluate the IU algorithm in Section 5.4. As was mentioned in Section 5.5.2, 36 problems were randomly selected for the training set for building performance profiles, and so the results shown in this subsection are based upon the remaining 100 test problems.

In order to give a comprehensive evaluation, for each test problem, this work not only considers the case where the agent has a finite computation time prior to execution, but also considers the case where the agent has some additional computation time in the midst of execution as well as the case where the agent can have as much computation time as it likes by paying additional costs. The parameters of these three test cases are defined as follows:

Case I : The agent has a limited computation time (i.e., t seconds) prior to execution to construct a solution, and it has no additional computation time during execution to improve that solution. Both $t = 50$ and $t = 100$ scenarios are tested.

Case II : The agent has a constant amount of computation time (i.e., 5 seconds) prior to execution to construct a solution, but it has some additional computation time (i.e., τ seconds per step) to reconsider the problem features and improve its solution in the midst of execution. Both $\tau = 0.5$ and $\tau = 1$ scenarios are tested.

Case III : The agent can use as much computation time as it desires prior to execution to formulate its policy, but spending more time in formulating the

Algorithm	Case I		Case II		Case III	
	t = 50	t = 100	τ = 0.5	τ = 1.0	C = 1	C = 2
CMP	**144.2**	**149.3**	**137.4**	**143.2**	**128.4**	**120.5**
IU	133.9	136.7	124.8	130.7	114.5	106.8
RTDP	113.1	118.8	117.3	125.2	98.3	90.7

Figure 5.8: The optimality comparison between phasing and non-phasing. CMP outperforms IU and RTDP in all test cases, based upon the average results over 100 test problems.

policy will delay the mission more and so result in a higher penalty, which is defined as $C(t) = c \times t$, where c denotes the cost per second and t represents the amount of the computation time used. In a similar manner to Case I, it is assumed that no further deliberation is available in the midst of execution. In this case, both $c = 1$ and $c = 2$ scenarios are tested.

The rest of this subsection is organized as follows. It starts by comparing the CMP approach with the techniques without using the phasing strategy, and then evaluates and compares each component technique of the CMP approach with prior techniques while keeping the other two components constant.

Phasing vs. Non-Phasing

Figure 5.8 illustrates and compares the results, averaged over 100 test problems, among the CMP approach and the other two approaches — RTDP and IU. These two approaches do not depend on phasing but can also exploit computation time both before and during execution. The RTDP algorithm exploits additional computation time during execution by starting trials at the current state (instead of the initial state) and updating values of the states that may be reached in the future. The

IU algorithm exploits additional time during execution by trimming its MDP and *openList* and continuing the unrolling process to unroll more states when moving from state to state.[9]

From the empirical results, we can see that the CMP approach performs better than the RTDP algorithm and the IU algorithm. This is mainly due to two reasons. First, decomposing a large problem into phases often reduces the number of MDP states, which in turn results in a smaller search place and so the agent can find a high-quality solution more quickly. Second, the CMP approach is more sophisticated in allocating computation time. It is worth pointing out that the RTDP, IU, and CMP approaches adopt considerably different ways of utilizing computation time. The RTDP algorithm repeatedly performs trials each of which starts at the current state and ends at the problem horizon, which could mean that computation time is roughly evenly distributed from the current state to the problem horizon. The IU algorithm implements a probability-first style search. Since states near the agent's current state usually have higher reaching probabilities than states far away from the current state (because of branching futures), the IU algorithm will focus more on the states near the current state (but, of course, not so much as the uninformed unroller). In contrast, the CMP approach adopts the mathematical-programming-based deliberation scheduling algorithm (as will be shown, whose runtime is negligible in comparison to policy formulation time) to decide which phases are worth more computation time. Such a selective way of allocating and using computation time makes the CMP approach find better solutions than the RTDP and IU approaches within time limits.

Heuristic Search Algorithm (with DEC and MP)	Case I		Case II		Case III	
	t = 50	t = 100	τ = 0.5	τ = 1.0	C = 1	C = 2
IU	144.2	149.3	137.4	143.2	128.4	120.5
AO*	130.5	141.9	128.8	138.7	109.1	100.1
RTDP	122.5	134.7	116.4	129.5	107.9	98.2

Figure 5.9: Evaluation of the CMP heuristic search component. The IU algorithm outperforms the AO* algorithm and the RTDP algorithm in all test cases, based upon the average results over 100 test problems. These heuristic search algorithms work with the same problem decomposition and deliberation scheduling algorithms.

Evaluation of the Policy Formulation Component

Figure 5.9 shows the results for three inner-phase heuristic search algorithms, including the IU, AO*, and RTDP algorithms, together with the same problem decomposition method (described in Section 5.5.1) and the same mathematical-programming-based deliberation scheduling algorithm (described in Chapter IV). The results are similar to those shown in Section 5.4, and support the previous conclusion that the IU algorithm can be a better policy formulation algorithm than the AO* algorithm and the RTDP algorithm in situations with limited computation time and with very large state spaces (at least for these types of test problems).

Evaluation of the Deliberation Scheduling Component

Figure 5.10 evaluates the deliberation scheduling component of the CMP approach. The mathematical-programming-based (MP-based) deliberation scheduling algorithm presented in Chapter IV is compared with a naive deliberation scheduling

[9]The trimming procedure, removing states that are no longer reachable, is for improving computational efficiency.

Deliberation Scheduling Algorithm (with DEC and IU)	Case I		Case II		Case III	
	t = 50	t = 100	τ = 0.5	τ = 1.0	C = 1	C = 2
MP	144.2	149.3	137.4	143.2	128.4	120.5
Naive	131.7	135.6	118.4	126.7	114.2	108.9
MDP	76.2	94.5	64.8	67.2	14.2	-73.1

Figure 5.10: Evaluation of the CMP deliberation scheduling component. The MP-based deliberation scheduling algorithm outperforms the naive deliberation scheduling algorithm and the MDP-based deliberation scheduling algorithm in all test cases, based upon the average results over 100 test problems. These deliberation scheduling algorithms work with the same problem decomposition and informed unroller algorithms.

algorithm, which attempts to evenly distribute computation time among phases, and a MDP-based deliberation scheduling algorithm (Goldman et al., 2001).

The MP-based algorithm can quickly find a near-optimal deliberation schedule with respect to the given performance profiles, i.e., 0.12 seconds on average, and a maximum 0.54 seconds for the largest test problem with 21 phases. The cost is low — this amount of time is usually negligible in comparison with the time used for policy formulation that is typically between 10 and 100 seconds, but the gain can be high — the deliberation schedule derived using this small amount of time can help the agent judiciously spend its remaining time on policy formulation procedures. In contrast, the MDP-based algorithm, though also able to find a near-optimal deliberation schedule, takes a long time (i.e., up to several hundred seconds) to derive the schedule. As a result, the agent might not have time for actual deliberations and thus its performance is poor, particularly in case III where the time used will incur a cost. On the other hand, although the naive algorithm can find a deliberation

Decomposition Algorithm (with IU and MP)	Case I		Case II		Case III	
	t = 50	t = 100	τ = 0.5	τ = 1.0	C = 1	C = 2
DEC	144.2	149.3	137.4	143.2	128.4	120.5
DEC-Large	138.4	141.4	131.2	138.3	118.5	112.4
DEC-Small	100.6	102.1	99.7	101.4	96.5	97.2

Figure 5.11: Evaluation of the CMP problem decomposition component. The heuristic decomposition method using a moderate merging size outperforms the method using a large merging size or a small merging size in all test cases, based upon the average results over 100 test problems. These problem decomposition methods work with the same deliberation scheduling and informed unroller algorithms.

schedule very quickly (i.e., 0.03 seconds on average), its solution is often suboptimal since it does not consider the differences of the values of phases.

Evaluation of the Problem Decomposition Component

This evaluation section concludes by evaluating the last CMP component — problem decomposition. The heuristic decomposition method presented in Section 5.5.1 is compared against two of its variations (since no prior decomposition methods are directly applicable to TÆMS models), including a DEC-Large method that merges small neighboring phases until the estimated number of states in the resulting phase is greater than a large predefined merging size 10^5 (vs. 10^4 in the CMP decomposition method), and a DEC-Small method that uses a small merging size 10^3 and so it often does not merge phases.

From the empirical results shown in Figure 5.11, we can see that the heuristic decomposition method with the moderate merging size (10^4) performs best. The DEC-Large method, though better than the approaches without phasing (whose results were shown in Figure 5.8), often yields phases with large state spaces and thus slows down the procedure for finding high-quality solutions because larger phases typically take longer time. On the other hand, the DEC-Small method results in small phases, in each of which the agent might be able to formulate a complete and optimal policy. However, recall that phasing partitions the problem time horizon into disjoint time windows, which means that having more independent phases may need to place stricter temporal constraints on the tasks within these phases and thus may impair the optimality. This explains why the solution quality of the DEC-Small method (that oversimplifies the problem) is far below the solution quality of the DEC method.

5.6 Summary

In this chapter, we have presented and empirically evaluated the informed unroller algorithm that can quickly find a high-quality solution to a complex problem with a very large state space, and have integrated this algorithm with problem decomposition and deliberation scheduling techniques to further improve the performance of a time-limited agent.

The contributions of the work presented in this chapter can be summarized as follows:

- To cope with computationally challenging situations where an agent only has a finite amount of "think time" to build and solve a large MDP for its phase decision procedure, this chapter has designed the informed unroller algorithm, which emphasizes the expansion of a subset of states that are believed to have a high probability of lying along trajectories of high-quality policies, while ignoring other states, to yield a better policy within the time limit. The empirical results (shown in Figure 5.6 and Figure 5.7) from the Coordinator domain have demonstrated the ability of this algorithm to find a high-quality solution faster than the prior heuristic search techniques, and thus the IU algorithm represents a promising new way for anytime policy formulation.

- The overall computation-driven mission-phasing approach, in which the informed unroller algorithm works with the problem decomposition and deliberation scheduling techniques, has been presented and evaluated. The problem decomposition method exploits problem structure to create phases and reduce the size of the state spaces, the across-phase deliberation scheduling techniques automatically distribute computation time among phase decision procedures according to their predicted values to the overall solution quality, and the inner-phase informed unroller algorithm biases state space expansion efforts towards states that are likely to be reached by high-quality policies. Together, these techniques help the agent better focus on "critical" regions of a large state space, and thus can often improve the performance of the time-limited agent. For example, as shown in Figure 5.8, the CMP approach achieved, on average, about 10% higher quality than the IU algorithm in the Coordinator test problems (and the IU algorithm has been shown to have better performance than prior approaches). This means that the computation-driven mission-phasing approach is of value in time-critical application domains.

CHAPTER VI

Conclusion

The work in this dissertation designed, analyzed, and evaluated a suite of computationally efficient algorithms that can automatically identify and utilize resource reconfiguration opportunities in resource-constrained environments and problem reconsideration opportunities in time-critical environments. The analytical and experimental results illustrated and emphasized that the mission phasing approach, incorporating problem decomposition, resource/time allocation, and policy formulation, can help a constrained agent judiciously and effectively exploit those opportunities to improve its performance.

This chapter concludes the dissertation with a summary of the main contributions of this work and a discussion of several promising future research directions.

6.1 Summary of Contributions

This work consists of two parts, corresponding to two popular constraints: the resource constraint and the computation time constraint.

- **Resource-Driven Mission-Phasing**

  The first part of this dissertation coped with non-consumable execution resource constraints. Chapter II presented a MILP-based approach that automates the process of finding and using phases for a capacity-limited agent. Chapter III extended this phasing idea to multi-agent environments where a group of agents share scarce resources. The contributions of this part of the work can be summarized as follows.

- ▷ This work explicitly took into account potential opportunities in the midst of execution to reconfigure resources and switch policies, and designed computationally efficient algorithms (including an abstract MDP algorithm for single-agent resource reconfiguration problems and a MILP-based algorithm for multi-agent resource reallocation problems) to optimize the use of these fixed opportunities in complex stochastic systems. The empirical results (Figure 2.6 and Figure 3.8) confirmed that exploiting such phase-switching opportunities can considerably improve the agent performance, particularly in tightly constrained systems (the reward doubles in some test cases).
- ▷ As an extension to utilizing fixed phase-switching opportunities, Section 2.5 (for single-agent systems) and Section 3.5 (for multi-agent systems) presented MILP-based algorithms that are able to automate the process of finding and using mission phases in stochastic, constrained systems, which not only eliminates the need for having phases predefined in the description of a mission, but also avoids potential sub-optimality caused by phases being improperly defined by a user.
- ▷ The automated resource-driven mission-phasing algorithms presented in this work are computationally efficient. Through formulating a whole mission-phasing problem into a compact mathematical formulation and then simultaneously solving the coupled problems of mission decomposition, resource allocation, and policy formulation, the presented algorithms could effec-

tively exploit problem structure, which results in a significant reduction in computational cost in comparison with the approach that considers mission decomposition, resource allocation, and policy formulation in isolation (e.g., a reduction from hours to seconds as was shown in Figure 3.12).

▷ Unlike much prior work where agents reactively reconfigure resources when exogenous events occur, this work, based upon Markov decision processes and sequential decision-making theory, can proactively determine and optimally utilize resource reconfiguration opportunities. It provides a new computationally efficient resource-reconfiguration mechanism for resource-constrained environments.

■ Computation-Driven Mission-Phasing

The computation-driven mission-phasing approach, the focus of the second part of this dissertation, aimed to further prior problem decomposition techniques that can properly decompose a problem into sub-problems but cannot solve all sub-problems completely within time limits. To this end, this work developed a mathematical-programming-based deliberation-scheduling approach that can selectively and effectively distribute limited computation time among multiple sub-problems, and developed a heuristic-search-based informed-unroller algorithm to make effective use of the assigned computation time within each sub-problem. Together they could help a time-limited agent focus its computational effort on high-value subsets of states within high-value phases. The contributions of this part of the work are summarized below.

▷ **Scheduling Phase Decision Procedures**

Chapter IV presented a new deliberation scheduling approach for computation time allocation. Compared to prior computationally efficient techniques, the techniques presented in this work are capable of finding near-optimal solutions in a wider variety of environments due to the following factors.

- This work explicitly considered opportunities (and modeled the costs) of improving policies of the future phases in the midst of execution, and developed a computationally efficient approach to solve the coupled problems of deciding both when to deliberate given its cost, and which decision procedures to execute during deliberation intervals. The experimental results showed and emphasized that this approach could improve agent performance in both deterministic environments (Figure 4.10) and stochastic environments (Figure 4.12).
- Using piecewise linear approximation and discretization techniques (introduced in Section 4.4.2), the deliberation scheduling approach presented in this work can cope with any class of performance profiles, which extended prior work focusing on concave performance profiles.
- The mathematical programming formulation provides a general and fundamental framework, which can be reformulated or extended to model other system aspects. As an illustration, Section 4.5 discussed how to transform deliberation scheduling problems with nonlinear objective functions into more computationally tractable problems with linear objective functions.
- The deliberation scheduling approach presented in this work is applicable in complex environments with uncertainties. Problems with non-deterministic phase transitions have been analyzed prior to this work

(e.g., Horvitz, 2001), but this work is more general since it can automatically schedule deliberations based upon deliberation costs and can tackle more general performance profiles.

▷ **Efficient Inner-Phase Heuristic Search**

To cope with challenging problems where an agent only has a finite amount of "think time" to build and solve a large MDP for its phase decision procedure, the work in Chapter V designed the informed unroller algorithm, which emphasizes the expansion of a subset of states that are believed to have a high probability of lying along trajectories of high-quality policies. The empirical evaluation (shown in Section 5.4) demonstrated the ability of this heuristic search algorithm to effectively and efficiently explore a large state space within limited computation time, and thus it represented a promising new strategy for anytime policy formulation.

6.2 Future Work

Although this dissertation presented a suite of algorithms to improve agent performance in constrained stochastic systems, there is still much interesting work left in the research areas of this dissertation. Several promising future research directions are outlined below.

▷ **Resource Constraints and Time Limitations**

Resource-driven mission-phasing problems are NP-complete. Although the solution approaches designed in this work can exploit problem structure to reduce computational cost, finding an exact solution to a complex RMP problem might still be difficult, particularly in time-limited environments.

This work performed some preliminary investigation on problems with both resource constraints and time limitations (in Section 2.6.4 and Section 3.6.4) through directly exploiting anytime performance of state-of-art MILP solvers. In the future, I would like to examine other possibilities, including heuristic search methods that can selectively search large MDP state spaces, and factored MDP solvers that can exploit problem structure and sparsity within the MDPs.

▷ **Resource Reallocation and Decentralized MDPs**

A simplifying assumption made in this work is that, once a resource reallocation is scheduled, participant agents will always be able to successfully redistribute resources among themselves at that scheduled time, regardless of which states they are in. I plan to relax this assumption in the future to consider sequential resource allocation problems with additional constraints on when and where the agents are able to exchange resources. For example, physical agents might only be able to exchange resources when they are at the same location at the same time. Or, as another example, a task might not be aborted once it starts, which means that it may be impossible to reassign the resources used by that task until the task is completed.

Decentralized MDPs are one possible way to solve such problems. Our preliminary work in (Wu and Durfee, 2006a) (not included in this dissertation) has developed a MILP-based algorithm for solving transition independent Dec-MDPs. That work linked the Dec-MDP formulation with the MILP formulation, and pointed out one way to characterize resource constraints in the MILP formulation. In the future, I will dig deeper in this direction.

▷ **Extensions in Deliberation Scheduling**

This work presented a fundamental mathematical-programming-based deliberation scheduling algorithm, and illustrated its extensions for nonlinear objective functions and non-deterministic phase transitions. One of my future research directions is to investigate some other potential extensions including:

- For an online application problem with a large number of phases, a complete mathematical formulation might be too complex to be solvable within time limits. Considering the fact that probabilities of reaching phases in the far future are often low due to uncertainties, one possible solution is to restrict the number of phases being put into the mathematical formulation. As time passes, deliberation scheduling can be implemented again in the midst of execution.

- This work made a simplifying assumption that the utility of a phase is determined only by the computation time assigned for it. This is not always true. For example, $phase_i$ might also gain utility when allocating time for another (relevant) $phase_j$ because $phase_i$ and $phase_j$ might have some similar sub-problems. Thanks to the mathematical formulation presented in this work, it is often easy to model such relations, e.g., $v_i = V_i(t_i, t_j)$. I will systematically investigate the implications of relaxing this assumption in the future.

- Finally, with the above techniques of limiting look-ahead steps and modeling inter-influence between phases, an autonomous agent might be able to solve deliberation scheduling problems with loops in phase transitions by *unrolling* the transition graph to make it acyclic. This is another promising future research topic.

6.3 Closing Remarks

System constraints affect the performance of autonomous agents. This work designed, analyzed, and evaluated mission phasing approaches, in which the problem decomposition, resource/time allocation, and policy formulation techniques are integrated, to help constrained agents perform better in both resource-limited environments and time-limited environments.

While there are still many open questions (some of which were discussed in the previous section), this dissertation showed that the phasing strategy is an effective way for improving resource/time allocation in stochastic systems, and helped pave the way for future work on constrained systems.

Bibliography

E. Altman. Constrained Markov decision processes with total cost criteria: Lagrange approach and dual LP. *Methods and Models in Operations Research*, 48:387–417, 1998.

A. C. Antoulas, D. C. Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. *Contemporary Mathematics*, 280:193–220, 2001.

A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138, 1995.

R. Becker, S. Zilberstein, V. R. Lesser, and C. V. Goldman. Solving transition independent decentralized Markov decision processes. *Journal of Artificial Intelligence Research*, 22:423–455, 2004.

R. Bellman. A Markov decision process. *Journal of Mathematical Mechanics*, 6: 679–684, 1957.

C. A. Bererton, G. J. Gordon, and S. Thrun. Auction mechanism design for multi-robot coordination. In *Advances in Neural Information Processing Systems*, 2003.

D. S. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of Markov decision processes. In *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence*, pages 32–37, 2000.

M. S. Boddy and T. Dean. Solving time-dependent planning problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 979–984, 1989.

M. S. Boddy and T. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, 1994.

H. Bojinov, A. Casal, and T. Hogg. Multiagent control of self-reconfigurable robots. *Artificial Intelligence*, 142:99, 2002.

B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of 13th International Conference on Automated Planning and Scheduling*, pages 12–21, 2003.

C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons, New York, 1998.

S. de Vries and R. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003.

T. Dean and S. H. Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 304–309, 1995.

T. Dean, L. P. Kaelbling, J. Kirman, and A. E. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

D. A. Dolgov. *Integrated Resource Allocation and Planning in Stochastic Multiagent Environments*. PhD thesis, Computer Science Department, University of Michigan, February 2006.

D. A. Dolgov and E. H. Durfee. Constructing optimal policies for agents with constrained architectures. Technical Report CSE-TR-476-03, University of Michigan, 2003.

D. A. Dolgov and E. H. Durfee. Computationally-efficient combinatorial auctions for resource allocation in weakly-coupled MDPs. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, July 2005.

D. A. Dolgov and E. H. Durfee. Resource allocation among agents with MDP-induced preferences. *Journal of Artificial Intelligence Research*, 27:505–549, December 2006.

G. R. Drozeski. *A Fault-Tolerant Control Architecture for Unmanned Aerial Vehicles*. PhD thesis, Georgia Institute of Technology, 2005.

M. Earl and R. D'Andrea. Iterative MILP methods for vehicle control problems. *IEEE Transactions on Robotics*, 21(6):1158–1167, December 2005.

G. Emami, J. Cheng, D. Cornwell, M. Feldhousen, C. Long, V. Malhotra, I. Starnes, L. Kerschberg, A. Brodsky, and X. Zhang. Active: agile coordinator testbed integrated virtual environment. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1580–1587, 2006.

E. Feinberg. Constrained discounted Markov decision processes and Hamiltonian cycles. *Mathematics of Operations Research*, 25:130–140, 2000.

B. Fox and D. M. Landi. An algorithm for identifying the ergodic subchains and transient states of a stochastic matrix. *Communications of the ACM*, 2:619–621, 1968.

C. V. Goldman and S. Zilberstein. Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research*, 22:143–174, 2004.

R. P. Goldman, D. J. Musliner, and K. D. Krebsbach. Managing online self-adaptation in real-time environments. In *Proceedings of the 2nd International Workshop on Self Adaptive Software*, pages 6–23, 2001.

E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

E. Horvitz. Principles and applications of continual computation. *Artificial Intelligence*, 126(1-2):159–196, 2001.

L. Kallenberg. Linear programming and finite Markovian control problems. *Mathematisch Centrum, Amsterdam*, 1983.

D. Karuppiah, R. Grupen, A. Hanson, and E. Riseman. Smart resource reconfiguration by exploiting dynamics in perceptual tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005.

H. Kautz and J. Walser. Integer optimization models of AI planning problems. *Knowledge Engineering Review*, 15(1):101–117, 2000.

T. Lane and L. P. Kaelbling. Toward hierarchical decomposition for planning in uncertain environments. In *Proceedings of the 2001 IJCAI Workshop on Planning under Uncertainty and Incomplete Information*, pages 1–7, 2001.

T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, 1986.

V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X. Zhang. Evolution of the GPGP/TAEMS domain-independent coordination framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, 2004.

A. Martelli and U. Montanari. Optimizing decision trees through heuristically guided search. *Communications of the ACM*, 21(12):1025–1039, 1978.

H. B. McMahan, M. Likhachev, and G. J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd international conference on Machine Learning*, pages 569–576, 2005.

N. Meuleau, M. Hauskrecht, K.-E. Kim, L. Peshkin, L. P. Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 165–172, 1998.

D. Musliner, R. Goldman, and K. Krebsbach. Deliberation scheduling strategies for adaptive mission planning in real-time environments. *Metacognition in Computation*, 5:04, 2005.

D. J. Musliner, E. H. Durfee, and K. G. Shin. CIRCA: A cooperative intelligent real time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1561–1574, 1993.

D. J. Musliner, E. H. Durfee, and K. G. Shin. World modeling for the dynamic construction of real-time control plans. *Artificial Intelligence*, 74(1):83–127, 1995.

D. J. Musliner, E. H. Durfee, J. Wu, D. A. Dolgov, R. P. Goldman, and M. S. Boddy. Coordinated plan management using multiagent MDPs. In *Working Notes of the AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management*, March 2006.

D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *IEEE Journal on Communications*, 24(8):1439 – 1451, 2006.

R. Parr. Flexible decomposition algorithms for weakly coupled Markov decision problems. In *Proceedings of the 14th Conference in Uncertainty in Artificial Intelligence*, pages 422–430, 1998.

A. Pekec and M. Rothkopf. Combinatorial auction design. *Management Science*, 49 (11):1485–1503, 2003.

A. Phillips. Functional decomposition in a vehicle control system. In *Proceedings of the 2002 American Control Conference*, 2002.

J. Porteous, L. Sebastia, and J. Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of the 6th European Conference on Planning*, pages 37–48, 2001.

M. J. D. Powell. *Approximation Theory and Methods.* Cambridge University Press, Cambridge UK, 1981.

D. Precup and R. Sutton. Multi-time models for temporally abstract planning. *Advances in Neural Information Processing Systems*, 10:1050–1056, 1998.

M. L. Puterman. *Markov Decision Processes.* John Wiley & Sons, New York, 1994.

A. Raja, G. Alexander, and V. Mappillai. Leveraging problem classification in online meta-cognition. *Proceedings of AAAI 2006 Spring Symposium on Distributed Plan and Schedule Management, Stanford*, pages 97–104, 2006.

S. A. Reveliotis. *Real-Time Management of Resource Allocation Systems: A Discrete Event Systems Approach.* Springer-Verlag New York, 2005.

D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, 10(1):107–124, 2001.

S. Russel. Rationality and intelligence. *Common Sense, Reasoning, and Rationality*, 2002.

G. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. *Proceedings of the 8th International Conference on Very Large Data Bases*, pages 257–262, 1982.

D. Schrage and G. Vachtsevanos. Software-enabled control for intelligent UAVs. In *Proceedings of 1999 International Conference on Control Applications*, 1999.

S. P. Sethi, H. Yan, H. Zhang, and Q. Zhang. Optimal and hierarchical controls in dynamic stochastic manufacturing systems: A survey. *Manufacturing & Service Operations Management*, 4(2):133–170, 2002.

T. Smith and R. Simmons. Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic. In *Proceedings of the National Conference on Artificial Intelligence*, 2006.

R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

D. Teneketzis, S. H. Javid, and B. Sridhar. Control of weakly-coupled Markov chains. In *Proceedings of the 1980 19th IEEE Conference on Decision and Control including the Symposium on Adaptive Processes*, 1980.

P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pages 585–590, 1999.

T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in AI planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 304–309, 1999.

T. Wagner, A. Raja, and V. Lesser. Modeling uncertainty and its implications to sophisticated control in TAEMS agents. *Autonomous Agents and Multi-Agent Systems*, 13(3):235–292, 2006.

L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, J. Prasad, D. Schrage, and G. Vachtsevanos. An open platform for reconfigurable control. *Control Systems Magazine, IEEE*, 21(3):49–64, 2001.

L. A. Wolsey. *Integer Programming.* John Wiley & Sons, New York, 1998.

C. Wu and D. Castanon. Decomposition techniques for temporal resource allocation. In *IEEE Conference on Decision and Control*, 2004.

J. Wu and E. H. Durfee. Automated resource-driven mission phasing techniques for constrained agents. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 331–338, 2005.

J. Wu and E. H. Durfee. Mixed-integer linear programming for transition-independent decentralized MDPs. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2006a.

J. Wu and E. H. Durfee. Mathematical programming for deliberation scheduling in time-limited domains. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2006b.

J. Wu and E. H. Durfee. Sequential resource allocation in multi-agent systems with uncertainties. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2007a.

J. Wu and E. H. Durfee. Solving large taems problems efficiently by selective exploration and decomposition. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2007b.

L. Xiao, M. Johansson, and S. P. Boyd. Simultaneous routing and resource allocation via dual decomposition. *IEEE Transactions on Communications*, 52(7):1136–1144, 2004.

X. Zhang and H. Xu. Towards automated development of multi-agent systems using RADE. *Proceedings of the 2006 International Conference on Artificial Intelligence*, pages 44–50, 2006.