COMPLEXITY ANALYSIS AND OPTIMAL ALGORITHMS FOR DECENTRALIZED DECISION MAKING

A Dissertation Presented

by

DANIEL S. BERNSTEIN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2005

Computer Science

COMPLEXITY ANALYSIS AND OPTIMAL ALGORITHMS FOR DECENTRALIZED DECISION MAKING

A Dissertation Presented

by

DANIEL S. BERNSTEIN

Approved as to style and content by:

Shlomo Zilberstein, Chair

Andrew G. Barto, Member

Neil Immerman, Member

Christopher Raphael, Member

Tuomas W. Sandholm, Member

W. Bruce Croft, Department Chair
Computer Science

This dissertation is dedicated to my parents.

ACKNOWLEDGMENTS

I am deeply grateful to my advisor Shlomo Zilberstein, who taught me so much and was so supportive over the past seven years. I am not alone in saying that Shlomo is a truly great advisor. I feel fortunate to have been able to work with him, and I look forward to all that we will do together in the future.

I would also like to thank the other members of my thesis committee. Andy Barto has done groundbreaking work in the area of reinforcement learning. Talking with him and attending his lab meetings has given me insight into the subject and has shaped the way I think about automated decision making. I have had many illuminating conversations with Neil Immerman. I first learned of Neil's extraordinary work in complexity theory when I was an undergraduate, and I was thrilled to be able to publish papers with him as a graduate student. Chris Raphael and Tuomas Sandholm provided extensive comments on drafts of the dissertation. At the defense, they asked insightful questions and brought up some very interesting issues.

I benefitted greatly from interactions with other graduate students in the department. First, I would like to thank the members of the RBR lab: Martin Allen, Chris Amato, Andy Arnt, Raphen Becker, Zhengzhu Feng, and Mark Gruman. They provided plenty of useful feedback on my work and attended many practice talks. Special thanks also go to Ted Perkins and Balaraman Ravindran for engaging in fun conversations and silly games with me early on in graduate school.

Over the past few years, I was fortunate to be able to collaborate with some excellent researchers outside of the department. Eric Hansen's insight into POMDP theory helped immensely in the development of the algorithms in this thesis. Bob

Givan contributed some very clever ideas to the complexity analysis of the DEC-POMDP problem.

I'd like to thank Pauline Hollister, Sharon Mallory, and Michele Roberts for being extremely helpful when administrative issues arose.

I met my wife, Ann Guo, soon after I came to UMass. It is hard to imagine getting through graduate school without her. I feel lucky that we found each other, and I know that I have her love and support in anything I decide to pursue.

Finally, I would like to thank my parents. I couldn't ask for better parents than the ones I have. My mom and dad have just been amazing from the very beginning. They allowed me to become a free thinker, and this has benefitted me in more ways than they know. Mom and dad, thanks for always cheering me on and I love you!

ABSTRACT

COMPLEXITY ANALYSIS AND OPTIMAL ALGORITHMS FOR DECENTRALIZED DECISION MAKING

SEPTEMBER 2005

DANIEL S. BERNSTEIN, B.Sc., CORNELL UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Shlomo Zilberstein

Coordination of distributed entities is required for problems arising in many areas, including multi-robot systems, networking applications, e-commerce applications, and the control of autonomous space vehicles. Because decisions must often be made without a global view of the system, coordination can be difficult. This dissertation focuses on the development of principled tools for solving problems of distributed decision making.

As a formal framework for such problems, we use the decentralized partially observable Markov decision process (DEC-POMDP). This framework is very general, incorporating stochastic action effects, uncertainty about the system state, and limited communication. It has been adopted for use in the fields of control theory, operations research, and artificial intelligence. Despite this fact, a number of fundamental questions about the computational aspects of the model have gone unanswered.

One contribution of this thesis is an analysis of the worst-case complexity of solving DEC-POMDPs. It was previously established that for a single agent, the finite-horizon version of the problem is PSPACE-complete. We show that the general problem is NEXP-complete, even if there are only two agents whose observations together determine the system state. This complexity result illustrates a fundamental difference between single agent and multiagent decision-making problems. In contrast to the single agent problem, the multiagent problem provably does not admit a polynomial-time algorithm. Furthermore, assuming that EXP and NEXP are distinct, the problem requires super-exponential time to solve in the worst case.

A second contribution is an optimal policy iteration algorithm for solving DEC-POMDPs. Stochastic finite-state controllers are used to represent policies. A controller can include a correlation device, which allows agents to correlate their actions without communicating during execution. The algorithm alternates between expanding the controller and performing value-preserving transformations, which modify a controller without sacrificing value. We present two efficient value-preserving transformations, one which can reduce the size of the controller and another which can improve its value while keeping the size fixed. Our policy iteration algorithm serves as the first nontrivial exact algorithm for DEC-POMDPs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Planning for Distributed Agents

A central problem in artificial intelligence is that of deciding on a course of action, given knowledge about the world. Often, problems of this nature possess simple solutions. If a robot stands next to a battery charger, and its battery power is very low, it should charge the battery. Other scenarios present more challenges, however. Consider, for example, a situation in which a team of robots must navigate through the site of a disaster to find and rescue victims. Each robot has many decisions to make, and in this case the decisions are more difficult because of time pressure, conflicting goals, noisy sensors, and limited communication with the other robots. The appropriate action to take is often far from obvious, and sophisticated reasoning capabilities may be required.

Automated planning is the use of algorithms that take as input a formal problem description and output a policy for acting. The intuitive example given above shows that the addition of different types of uncertainty can increase the complexity of planning problems. Uncertainty can result from stochastic action effects, limited information about the state of the world, and the presence of other decision makers.

Much work on automated planning under uncertainty has decision theory at its foundation. The basic decision-theoretic problem setup is very general. A decision maker, or agent, has a set of actions. Each action leads to a distribution over outcomes, and each outcome produces some reward for the agent. The aim is simply to choose the action with the highest expected reward.

Basic decision theory does not capture the fact that many decision-making problems are sequential in nature. The decision an agent makes at one point in time will affect its situation at a future point in time. The standard decision-theoretic model for expressing such problems is the Markov decision process (MDP). In an MDP, an agent interacts with its environment at some discrete time scale. At each time step, the agent perceives the state of the system and chooses an action. Based on the action, a state transition occurs, and the environment produces a real-valued reward. The aim is to act so as to maximize expected long-term reward. MDPs have been used in many different areas, and have been applied to a wide range of problems.

The assumption of full knowledge of the state of the system is sometimes unrealistic. To address this, researchers often employ a generalization of MDPs called partially observable Markov decision processes (POMDPs). In a POMDP, the agent must base its decisions on noisy observations of the state. Though the model is more expressive, it also leads to more difficult problems. For example, the agent must sometimes trade off between acting to gain immediate reward and acting to gain information about the state that will lead to later reward.

Even the POMDP model is not general enough to express problems in which a team of distributed agents must act together in pursuit of a common goal. Here, we take distributed to mean that each agent receives its own local observations. The agents are not able to share observations during execution time. A naive approach would be to consider the other agents as part of the hidden state, and solve the problem as a POMDP. However, the best policies for the other agents depend on the best policy for the original agent, and this leads to circularity. To deal with this problem, we use the more general decentralized partially observable Markov decision process (DEC-POMDP).

The computational problem that we address in this work is that of finding a set of local policies for the agents that maximizes expected long-term reward. We assume

that a complete model of the system is available to the solver. Thus, our problem can be thought of as centralized planning for distributed agents.

Planning for distributed agents is a problem that many researchers have studied in the past. However, prior to our work, little was known about optimal algorithms for the problem. We provide the first nontrivial optimal algorithm, along with an analysis of the inherent complexity of the problem. Our insights constitute a significant addition to the body of knowledge on automated planning under uncertainty, and the tools we have developed expand the range of problems that can be solved using automated planning techniques.

1.2 Summary of Contributions

This thesis contains answers to fundamental questions about the computational aspects of DEC-POMDPs. The algorithms and analyses herein provide a rigorous foundation for future work on these problems.

The first question regards the inherent complexity of DEC-POMDPs. To date, the finite-horizon version of the problem has not been shown to be complete for any complexity class. It is known that the finite-horizon POMDP problem is PSPACE-complete when represented in standard tabular form [51]. It is believed that P $\neq$ PSPACE, but this has not been proven. Thus, finite-horizon POMDPs are likely to be intractable. Obviously, finite-horizon DEC-POMDPs are at least as hard. However, it was previously unknown whether or not they are *harder* than POMDPs.

We show that finite-horizon DEC-POMDPs are NEXP-complete, even with just two agents whose observations together determine the state. This reveals a fundamental difference between centralized and decentralized control of Markov processes. Since P $\neq$ NEXP, the two-agent problem is *provably* intractable. Furthermore, assuming that EXP $\neq$ NEXP, these problems require superexponential time to solve in the worst case.

Besides answering an open question about complexity theory and decentralized control, we expect our complexity analysis to be useful in the design of algorithms for DEC-POMDPs. In particular, it will help in determining which algorithmic ideas from POMDP theory will easily transfer to the DEC-POMDP case, and which will not. An algorithm that terminates in exponential time in the worst case, such as forward search from a start state, is expected to be difficult (though not necessarily impossible) to adapt to the multiagent case.

Dynamic programming algorithms are the simplest and most widely used algorithms for solving MDPs. These algorithms have been extended to POMDPs, and have proved useful in solving many problems, despite the negative complexity results for the model. Our second contribution is a generalization of the standard dynamic programming approach to POMDPs that yields optimal policies for DEC-POMDPs. This serves as the first nontrivial optimal algorithm for DEC-POMDPs.

Our dynamic programming framework represents policies using stochastic finite-state controllers. A simple way to implement this is to give each agent its own local controller. In this case, the agents' policies are all independent. A more general class of policies includes those which allow agents to share a common source of randomness without sharing observations. We define this class formally, using a shared source of randomness called a *correlation device*.

In our dynamic programming framework, the two phases of an iteration are *exhaustive backups*, which add nodes to the controller, and *value-preserving transformations*, which change the controller without sacrificing value. We prove that repeated application of these two operations yields convergence to optimality. The analysis is done for the infinite-horizon discounted case, though many of the central ideas carry over to the finite-horizon case.

We describe two types of value-preserving transformations, both of which can be performed efficiently using linear programming. The first type allows us to remove

nodes from the controller, and the second allows us to improve the value of the controller while keeping its size fixed. Empirical results provide insight into the dynamic programming algorithm and guidance for implementing it.

1.3 Previous Work

Decentralized control of Markov processes is a topic that has been studied in several different fields. In this section, we provide historical background for our work on DEC-POMDPs, and cite existing results.

1.3.1 Control Theory

Early work on distributed decision making was done by Radner [61], who introduced the *static team decision problem*. In this problem, the payoff to the agents is based on a single decision, and there is no dynamics. This problem was later shown to be NP-complete by Tsitsiklis and Papadimitriou [50].

The problem including dynamics was studied extensively in the control theory community in the 1970s (see [31] for a survey). It is an example of a problem with a *nonclassical information structure*. In this general class of problems, decisions must be made without knowledge of the full history of observations. DEC-POMDPs fall into this category because the agents do not have access to each others' observations.

Much of the work at this time focused on delayed information sharing. For problems in which all observations are shared at each step with a one-step delay, reduction to the single-agent case is possible. However, it was shown that delays of more than one step give rise to problems [69, 33]. During this period, no algorithms were proposed for the general problem, beyond brute force search through the space of all policies.

More recently, an approach based on approximate dynamic programming has been introduced for the case in which the observations of all the agents together determine

the state [14]. With this approach, the aim is to find a distributed value function that approximates the value function of the centralized version of the problem, and to extract a policy from the distributed value function.

1.3.2 Operations Research

A relevant body of work was produced in the operations research community in the 1970s. At this time, algorithms had not even been developed for POMDPs. Breakthroughs were made when Astrom [1] discovered that a POMDP can be converted into an equivalent MDP with a continuous state space, and Smallwood and Sondik [64] subsequently developed a method for implementing value iteration for this MDP using finite memory and time.

Many different versions of value iteration were developed (these are surveyed in [10]), along with a policy iteration algorithm [65]. In addition, Platzman explored the use of finite-state controllers as a policy representation for infinite-horizon POMDPs [57]. Despite all the work on dynamic programming for POMDPs, no attempts were made to extend these algorithms to the multiagent case.

1.3.3 Game Theory

Game theory deals with interactions among agents with potentially different objectives. A game consists of a set of *players*, a set of *strategies* for each player, and a *payoff function* for each player. The payoff function assigns a value to each *strategy profile*, or tuple of strategies for the agents. The most well-known solution concept for games is the *Nash equilibrium*, which is a strategy profile from which no player has incentive to deviate.

In a *normal form* representation of a game, the payoffs for all possible strategy profiles are explicitly enumerated. Much of the research in game theory has focused on more structured representations of games. In an *extensive form game*, payoffs are based on sequences of actions chosen by the players. A game of this type can be

viewed as a tree, called a *game tree*. Extensive form games of *imperfect information* allow for the players to receive local observations. Examples of these types of games include the card games poker and bridge.

Though these types of games have been discussed for decades, there has been relatively little work on general algorithms for them. Perhaps the most important result in this area was independently invented in [62] and [37]. This work presents a way of representing extensive form games that allows for efficient computation of a Nash equilibrium in the two-player zero-sum case. Though this is a powerful result, it should be noted that *efficient* in this context means polynomial in the size of the game tree. For many games of interest, the game tree is extremely large. More recent work has focused on representing these types of games using extensions of the influence diagram framework [38]. This representation is often more natural and space efficient.

A *stochastic game* is an extensive form game in which the players jointly control a Markov process. By introducing local observations, we get a *partially observable stochastic game (POSG)*. A DEC-POMDP can be viewed as a POSG in which all agents share the same payoff function. Extending algorithms for DEC-POMDPs to POSGs is an interesting topic, and will be discussed in the final chapter.

1.3.4 Artificial Intelligence

Early work on planning in artificial intelligence assumed a deterministic world. The aim was to find a sequence of actions leading from a start state to a goal state [20]. More recently, research on planning has focused on problems involving uncertainty. This has led researchers to employ decision-theoretic approaches. In particular, the Markov decision process has been adopted as the central framework for sequential decision making under uncertainty [7].

The MDP framework has allowed AI researchers to draw connections to work in other disciplines such as operations research and control theory, and to view existing systems in a new light. Over the past few decades, many algorithms for planning and learning have been analyzed using the MDP framework [7, 66], and this work has given rise to many new applications.

The POMDP framework has gained popularity in AI for handling problems in which the decision maker has limited access to the system state. Improvements on the original value iteration algorithm have been developed [10, 34, 18], along with new policy iteration algorithms [28, 58]. This work will be described in more detail in a later chapter. In addition, several approximation algorithms have been developed. Many of these are surveyed in [45].

Early work in distributed artificial intelligence focused on heuristic approaches to agent coordination (see [71] for a survey). Many subproblems of distributed control have been addressed, including task allocation, communication, and merging locally-generated plans. One advantage a formal approach has over previous approaches is that it provides a precise definition of optimality. If an optimal policy can be found, then all of the subproblems mentioned above are solved implicitly.

Recently, there has been extensive work on planning for multiple agents that uses the MDP framework as a formal foundation. To model multiple interacting agents, each with complete access to the state, one can use an MDP model in which each action is actually a vector of local actions, one for each agent. Early work on coordination in multiagent MDPs was done by Boutilier [8], and since then the framework has been used for multiagent planning [24, 25] and reinforcement learning [13, 35, 70, 11].

The DEC-POMDP framework has been used to model multiagent decision making problems with limited observability. In [46], the authors present a heuristic algorithm which holds the policies of all agents but one fixed, and solves the remaining agent's

problem as a POMDP. Though effective on some problems, this algorithm is not guaranteed to converge to a globally optimal solution. Another heuristic algorithm is presented in [17]. With this approach, the problem is decomposed into a sequence of one-step problems, from which an approximate solution can be obtained. Finally, model-free reinforcement learning algorithms have been developed for DEC-POMDPs [54, 15].

Some authors have studied versions of the model in which additional assumptions are made. In [3], it is assumed that each agent has a local state, and that the state transitions for all the agents are independent. An algorithm is presented that takes advantage of this special structure. Other types of independence assumptions are explored in [23]. The I-POMDP model, described in [21], can be thought of as a DEC-POMDP in which agents have beliefs about the other agents' policies and beliefs.

Another body of work deals with communication in DEC-POMDPs. Implicit communication is possible within the general framework, but for some problems it may be useful to model communication explicitly. Extensions to the DEC-POMDP framework to include communication are presented in [72, 60, 22].

1.4 Examples

The DEC-POMDP framework can be used to model problems in which the decision making is inherently distributed, and it can also be used for testing out distributed solutions to problems for which information sharing is possible. Below are some examples of real-world problems which can be formalized as DEC-POMDPs.

Multi-robot navigation: Navigation using a team of robots can be viewed as a DEC-POMDP. The agents are the robots, and the partial observability may come from noisy sensors and limited inter-robot communication. At each time step, each

robot must decide where to move next. In [17, 16], problems of this nature were formalized as DEC-POMDPs. In these problems, the goal of the team of robots is to capture an opponent that moves around in a random fashion.

Load balancing among queues: The problem of load balancing among a collection of queues fits naturally within the DEC-POMDP framework. In this problem, queues receive jobs and try to redistribute them to balance the load on the system. In this case, the agents are the individual queues. Each queue may only observe its own backlog and the backlogs of its immediate neighbors. Using only this information, a queue must decide on each time step whether or not to pass a job to another queue, and if so, which queue. A small problem of this type was modeled as a DEC-POMDP and solved approximately in [14].

Packet routing: Distributed control problems often arise in the area of computer networking. The problem of packet routing, in particular, is a good fit for the DEC-POMDP framework. In this problem, the aim is to route packets through a network in such a way that the average transfer time is minimized. The agents are the routers. Each router is directly connected to a set of other routers, and must decide on each time step which of its neighbors it should send each packet to. The routers are unaware of the topology of the network, so decisions must be made based on local information. Costs are incurred when packets are sent along a link, and when packets wait in a queue at a router. In [53], a network routing problem was formulated as a DEC-POMDP and solved via a multiagent reinforcement learning algorithm.

Multi-rover exploration: Intelligent coordination of distributed entities is important for many problems in the realm of space exploration. One particular problem that may benefit from work on DEC-POMDPs is that of exploring the surface of Mars

with a team of rovers. In this case, each rover is an agent. The reward function for the rovers may be based on successfully collecting scientific data and transmitting it to Earth. Each rover's decisions may involve selecting sites to explore and choosing experiments to perform. The effects of actions are stochastic, and limitations on communication force the rovers to act in a distributed manner. Initial work on formalizing rover decision making as a DEC-POMDP can be found in [3].

1.5 Outline

The remainder of the thesis is organized as follows.

Chapter 2 introduces the formal models used in this thesis. The MDP, POMDP, and DEC-POMDP frameworks are defined. In addition, the notions of value functions and optimality are introduced.

Chapter 3 contains a complexity analysis. We first present a review of computational complexity theory, and discuss previous complexity results for MDPs and POMDPs. Following this, we provide a complexity analysis of the finite-horizon DEC-POMDP problem, showing that it is NEXP-complete even with only two agents whose observations together determine the state. These results were published in [5].

Chapter 4 is a review of the main dynamic programming algorithms for centralized control. The classical value iteration and policy iteration algorithms are described for MDPs. For POMDPs, Smallwood and Sondik's value iteration algorithm is described. In addition, a policy iteration algorithm is presented that uses stochastic finite state controllers to represent policies.

In chapter 5, we present an extension of policy iteration for POMDPs to the DEC-POMDP case. In this extension, the local policy for each agent is represented using a stochastic finite-state controller, and a correlation device is used to induce dependencies among agents. A convergence proof is given, along with different ways of implementing the algorithm. Experiments using policy iteration are described in

chapter 6. We previously developed closely related dynamic programming algorithms for DEC-POMDPs, and these are described in [29, 6].

Finally, chapter 7 contains a summary of the contributions of the thesis and a discussion of possible future work.

CHAPTER 2

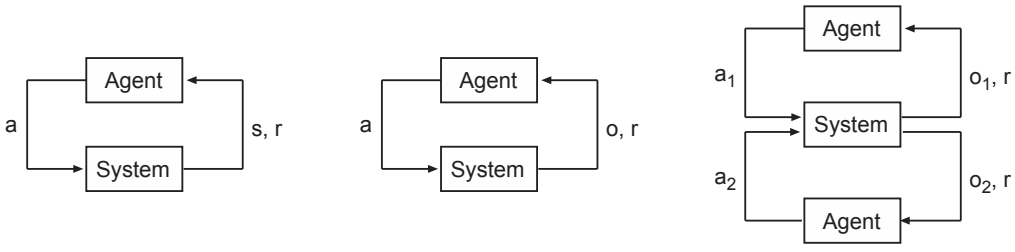
MODELS OF SEQUENTIAL DECISION MAKING

In this chapter, we describe the formal framework upon which our work is based. We begin by defining Markov decision processes, which model decision making by a single agent that has complete access to the state of the system. To deal with problems in which the agent has limited observations, we consider the more general partially observable Markov decision process.

We must turn to an even more general model to handle problems involving teams of distributed agents. This model is called the decentralized partially observable Markov decision process. An interesting special case of this model is the case in which the observations of all the agents, taken together, determine the state. Because this model involves distributed decision making but does not allow for hidden state, we call it the decentralized Markov decision process. We provide a formal definition of each model, along with definitions of decision-making policies and optimality.

2.1 Markov Decision Process

The *Markov decision process (MDP)* framework provides a useful way to model a single agent acting in a stochastic environment. The agent interacts with the environment at some discrete time scale. At each time step, the agent observes the state of the system and consequently chooses an action. The action produces a numerical reward and causes a stochastic transition to a new state. The agent's objective is to maximize its expected long-term reward. The interaction between the agent and its environment is illustrated in Figure 2.1a.



(a) (b) (c)

Figure 2.1: (a) Markov decision process. (b) Partially observable Markov decision process. (c) Decentralized partially observable Markov decision process with two agents.

We consider MDPs that can be defined as a tuple $\langle S, A, T, R \rangle$, where

- S is a finite set of states, with distinguished initial state s_0.
- A is a finite set of actions.
- $T : S \times A \to \Delta S$ is a state transition function.
- $R : S \times A \to \Re$ is a reward function.

Let us define a *history* as a sequence of state-action pairs, starting with s_0. The agent may choose its action based upon the entire history up until the current time step. Furthermore, the agent may employ randomness in selecting an action. We define a *policy* to be a mapping from histories to distributions over actions, and denote a policy with the symbol δ. In the following subsection, we describe in detail how policies are to be compared.

Performance Criteria

A performance criterion provides a way to define the utility of a policy. Recall that the agent aims to maximize expected long-term reward. We will consider two different ways of formalizing this notion.

We first consider the *finite-horizon* performance criterion. In this case, we assume that the agent will act for a predetermined number of time steps. This number is called the *horizon*, and is denoted H. Using this criterion, the *value* of a policy δ for a state s is defined as

$$V^{\delta}(s) = E_{s,\delta}\left[\sum_{t=0}^{H-1} R(s_t, a_t)\right].$$

We further define the *optimal value* for state s as

$$V^{*}(s) = \max_{\delta} V^{\delta}(s).$$

Any policy that achieves value $V^{*}(s_0)$ from the start state s_0 is an *optimal policy*. We note that the algorithms we will consider actually produce policies which optimize the value of *all* states simultaneously.

For some problems, either the number of time steps is not known in advance, or the number is so large that it is best thought of as infinite. For these problems, we use the *infinite-horizon discounted* performance criterion. Under this criterion, the value of a policy δ for a state s is defined as

$$V^{\delta}(s) = E_{s,\delta}\left[\sum_{t=0}^{\infty} \beta^{t} R(s_t, a_t)\right],$$

where $\beta \in [0, 1)$ is a *discount factor*. The discount factor ensures that the sum of rewards is finite, and is useful in the development of provably convergent MDP solution techniques. Optimal values and policies are defined as in the previous case.

Although we defined policies to be stochastic mappings from histories to actions, only a small subclass of policies is necessary for optimal behavior. It can be shown that given the current state, an agent need not remember previous states. In addition, stochasticity is not needed for optimal behavior. In the finite-horizon case, there is always an optimal policy which maps the current state and time step into an action.

In the infinite-horizon case, there is always an optimal policy which maps only the current state to an action. These properties do *not* hold when the agent receives noisy observations of the state, as will be described in the next section.

2.2 Partially Observable MDP

In a *partially observable Markov decision process (POMDP)*, an agent controls a Markov process with the aim of maximizing its expected long-term reward. However, instead of observing the state at each time step, the agent receives a noisy observation based on the state and the previous action (see Figure 2.1b). This generalization allows for more flexibility in modeling decision-making problems, but leads to higher complexity, as we will see in the next chapter.

Formally, a POMDP is defined as a tuple $\langle S, A, T, R, \Omega, O \rangle$, where

- S is a finite set of states, with distinguished initial state s_0.
- A is a finite set of actions.
- $T : S \times A \to \Delta S$ is a state transition function.
- $R : S \times A \to \Re$ is a reward function.
- Ω is a finite set of observations.
- $O : A \times S \to \Delta\Omega$ is an observation function.

The most general definition of a policy is as a stochastic mapping from histories of action-observation pairs to actions. The performance criteria and corresponding optimality definitions are the same as in the MDP model. As in that case, the most general class of policies is not necessary for optimal behavior. It turns out that optimality can be achieved using only deterministic mappings from histories of observations to actions.

In later chapters, we will consider classes of policies in which agents have bounded memory. To achieve the best possible performance with limited memory, stochasticity may be required.

2.3 Decentralized POMDP

We now turn to models in which a Markov process is controlled by a team of distributed agents. On each time step, the agents each receive potentially different observations, and subsequently choose actions. The rewards and state transitions can depend on the vectors of actions of all the agents. This is illustrated in the Figure 2.1c.

A *decentralized partially observable Markov decision process (DEC-POMDP)* is defined formally as a tuple $\langle I, S, \vec{A}, T, R, \vec{\Omega}, O \rangle$, where

- I is a finite set of agents.
- S is a finite set of states, with distinguished initial state s_0.
- $\vec{A} = \times_{i \in I} A_i$ is a set of joint actions, where A_i is the set of actions for agent i.
- $T : S \times \vec{A} \to \Delta S$ is the state transition function.
- $R : S \times \vec{A} \to \Re$ is the reward function.
- $\vec{\Omega} = \times_{i \in I} \Omega_i$ is a set of joint observations, where Ω_i is the set of observations for agent i.
- $O : \vec{A} \times S \to \Delta \vec{\Omega}$ is an observation function.

We now define a *local history* to be a sequence of local observations and actions. Local actions are chosen based on local histories, and it turns out that optimal behavior can be achieved with a set of deterministic mappings from local histories to actions.

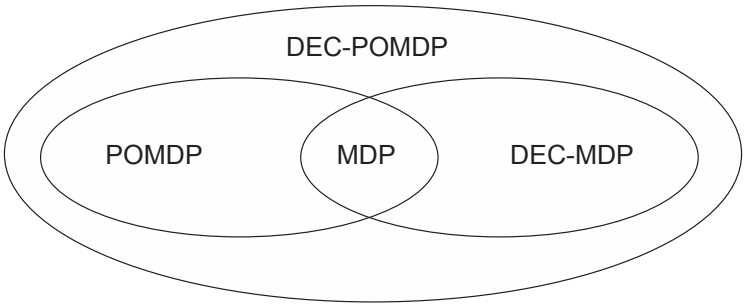


Figure 2.2: The relationships among the different models of decision making under uncertainty.

Of course, if the agents could share histories, greater reward could be possible, but this would no longer be distributed control.

As in the single agent case, we will consider bounded memory policies. Within these constraints, stochasticity and correlation among agents may be necessary for optimality. This will be shown formally in a later chapter.

2.4 Decentralized MDP

In a DEC-POMDP, there can be distributed policy execution and partial observability of the system state. It seems natural to consider the special case in which the observations of the agents together determine the state, even though no single agent necessarily observes the state.

Formally, we say that a DEC-POMDP is *jointly observable* if there exists a mapping $J : \vec{\Omega} \rightarrow S$ such that whenever $P(\vec{o}|\vec{a}, s')$ is nonzero, $J(\vec{o}) = s'$. We define a *decentralized Markov decision process (DEC-MDP)* to be a DEC-POMDP that is jointly observable.

The relationships among the four models presented are shown in Figure 2.2. We see that all of the models can be viewed as special cases of a DEC-POMDP.

CHAPTER 3

COMPLEXITY RESULTS

In this chapter, we present a complexity analysis of the finite-horizon DEC-POMDP problem. We first show that the problem can be solved nondeterministically in exponential time. Next, we show that the problem is hard for nondeterministic exponential time, even when it is restricted to be a DEC-MDP with two agents. This stands in contrast to the fact that POMDPs are complete for polynomial space, and illustrates a fundamental difference between centralized and decentralized control.

We begin the chapter with a brief introduction to the the theory of computational complexity. Following that, we review the complexity results for centralized control of Markov decision processes. With this background in hand, we proceed with our analysis of the decentralized case.

3.1 Review of Computational Complexity

A complexity class is a set of problems, where a problem is an infinite set of instances, each of which has a "yes" or "no" answer. In order to discuss the complexity of optimization problems, we must have a way of converting them to "yes/no" problems. The typical way this is done is to set a threshold and ask whether or not the optimal solution yields a reward that is no less than this threshold.

3.1.1 Some Complexity Classes

The first complexity class we consider is P, the set of problems that can be solved in polynomial time (in the size of the problem instance) on a sequential computer.

NP is the set of problems that can be solved *nondeterministically* in polynomial time. A nondeterministic machine automatically knows the correct path to take any time there is a choice as to how the computation should proceed. An example of a problem that can be solved nondeterministically in polynomial time is deciding whether a sentence of propositional logic is satisfiable. The machine can guess an assignment of truth values to variables and evaluate the resulting expression in polynomial time. Of course, nondeterministic machines do not really exist, and the most efficient known algorithms for simulating them take exponential time in the worst case. In fact, it is strongly believed by most complexity theorists that P $\neq$ NP (but this has *not* been proven formally).

Complexity can also be measured in terms of the amount of *space* a computation requires. One class, PSPACE, includes all problems that can be solved in polynomial space. Any problem that can be solved in polynomial time or nondeterministic polynomial time can be solved in polynomial space (i.e., P $\subseteq$ NP $\subseteq$ PSPACE) — that P $\subseteq$ PSPACE can be seen informally by observing that only polynomially much space can be accessed in polynomially many time steps.

Moving up the complexity hierarchy, we have exponential time (EXP) and nondeterministic exponential time (NEXP). By exponential time, we mean time bounded by 2^{n^k}, where n is the input size and $k > 0$ is a constant. It is known that PSPACE $\subseteq$ EXP $\subseteq$ NEXP, and it is believed that EXP $\neq$ NEXP (but again this has not been proven). It *has* been proven that the classes P and EXP are distinct, however.

Some classes of problems cannot be solved by a Turing machine in a finite number of steps. These problems are called *undecidable*. The problems which *can* be solved in finite time constitute the *recursive* set.

3.1.2 Reductions and Completeness

The notion of a *reduction* is important in complexity theory. We say that a problem A is *reducible* to a problem B if any instance x of A can be converted into an instance $f(x)$ of B such that the answer to x is "yes" if and only if the answer to $f(x)$ is "yes." A problem A is said to be *hard* for a complexity class C (or C-hard) if any problem in C is *efficiently* reducible to A. If the complexity class in question is P, efficient means that $f(x)$ can be computed using at most logarithmic space, while for the classes above P, efficient means that $f(x)$ can be computed using at most polynomial time. A problem A is said to be *complete* for a complexity class C (or C-complete) if (a) A is contained in C, and (b) A is hard for C. For instance, the satisfiability problem mentioned above is NP-complete and P-hard. However, unless P = NP, satisfiability is not P-complete.

3.2 Complexity Results for Centralized Control

For the purposes of complexity analysis, the transition and reward functions for an MDP are assumed to be represented as tables of rational numbers. For the finite-horizon case, the horizon H is provided as input, and for the infinite-horizon discounted case, a rational discount factor, β, is provided as input. Finally, a rational number K is provided, and the decision problem is whether or not there exists a policy which yields an expected total reward of at least K from the start state s_0.

It is assumed that in the finite-horizon MDP problem, $H < |S|$. Intuitively, the horizon must be on the order of the size of the problem. This actually strengthens any hardness result, because if the the problem is hard with the restriction, then it is at least as hard without the restriction. It is used in the upper bound proof, which gives an algorithm with running time proportional to the horizon. The following result, first proven in [51], implies that both versions of the problem can be solved efficiently.

Theorem 1 *The finite-horizon and infinite-horizon discounted MDP problems are both P-complete.*

For the POMDP results, the form of the input and the restrictions on the horizon are the same as in the fully observable case. For the finite-horizon case, noisy observations lead to a jump in complexity [51].

Theorem 2 *The finite-horizon POMDP problem is PSPACE-complete.*

For all practical purposes, this result says that finite-horizon POMDPs require exponential time to solve in the worst case. In contrast to the fully observable case, moving to the infinite-horizon increases the complexity even more. In fact, it has been shown that the problem becomes undecidable [44].

Theorem 3 *The infinite-horizon discounted POMDP problem is undecidable.*

3.3 Upper Bound for Decentralized Control

This section consists of a straightforward upper bound on the worst-case time complexity of solving finite-horizon DEC-POMDPs.

Theorem 4 *The finite-horizon DEC-POMDP problem can be solved nondeterministically in exponential time.*

Proof: We must show that a nondeterministic machine can solve any finite-horizon DEC-POMDP using at most exponential time. Consider an arbitrary DEC-POMDP with k agents. First, a joint policy δ can be "guessed" and written down in exponential time. This is because a joint policy consists of k mappings from local histories to actions, and since $H < |S|$, all histories have length less than $|S|$. A DEC-POMDP together with a joint policy can be viewed as a POMDP together with a policy, where the observations in the POMDP correspond to the observation tuples in the DEC-POMDP (one from each agent), and the POMDP actions correspond to tuples of

DEC-POMDP actions (again, one from each agent). In exponential time, each of the exponentially many possible sequences of observations can be converted into a belief state (i.e., a probability distribution over the state set giving the probability of being in each state after seeing the given observation sequence). The transition probabilities and expected rewards for the corresponding exponential-sized belief-state MDP can be computed in exponential time. Using dynamic programming, the joint policy can be evaluated in time polynomial in its size, which is exponential in the size of the original DEC-POMDP. □

3.4 Hardness Result for Decentralized Control

We now turn to the more challenging task of proving that the problem is NEXP-hard. We will in fact prove the stronger result that a DEC-MDP with two agents is NEXP-hard. This is done using a reduction from the TILING problem. We describe the TILING problem, give an overview of the reduction and correctness proof, and finally present the entire proof in detail.

3.4.1 The TILING Problem

We can show this lower bound by reducing any NEXP-complete problem to a two-agent DEC-MDP using a polynomial-time algorithm. For our reduction, we use an NEXP-complete problem called TILING [40, 49], which is described as follows. We are given a board size n (represented compactly in binary), a set of tile types $L = \{\mathsf{tile\text{-}0}, \ldots, \mathsf{tile\text{-}k}\}$, and a set of binary horizontal and vertical compatibility relations $H, V \subseteq L \times L$. A *tiling* is a mapping $f : \{0, \ldots, n-1\} \times \{0, \ldots, n-1\} \to L$. A tiling f is *consistent* if and only if (a) $f(0,0) = \mathsf{tile\text{-}0}$, and (b) for all x, y $\langle f(x,y), f(x+1,y)\rangle \in H$, and $\langle f(x,y), f(x,y+1)\rangle \in V$. The decision problem is to determine, given L, H, V, and n, whether a consistent tiling exists. An example of a tiling instance and a corresponding consistent tiling is shown in Figure 3.1.

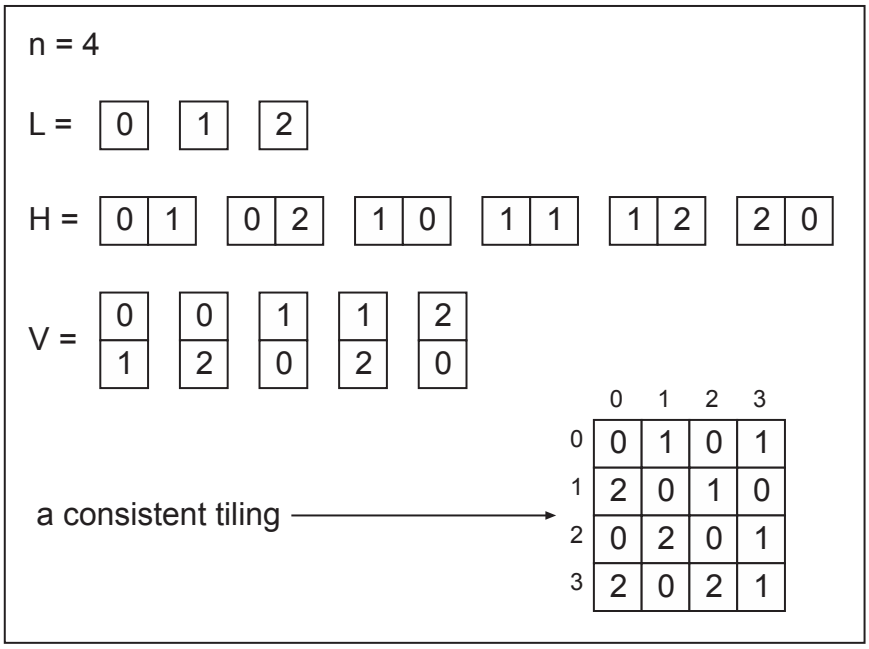


Figure 3.1: An example of a tiling instance.

In the remainder of this section, we assume that we have fixed an arbitrarily chosen instance of the tiling problem, so that L, H, V, and n are fixed. We then construct an instance of DEC-MDP that is solvable if and only if the selected tiling instance is solvable. We note that the DEC-MDP instance must be constructible in time polynomial in the size of the tiling instance (which in particular is logarithmic in the value of n), which will require the DEC-MDP instance to be at most polynomially larger than the tiling instance.

3.4.2 Overview of the Reduction

The basic idea of our reduction is to create a two-agent DEC-MDP that randomly selects two tiling locations bit by bit, informing one agent of the first location and the other agent of the second location. The agents' local policies are observation-history based, so the agents can base their future actions on the tiling locations given to them. After generating the locations, the agents are simultaneously "queried" (i.e., a

state is reached in which their actions are interpreted as answers to a query) for a tile type to place at the location given. We design the DEC-MDP problem so that the only way the agents can achieve nonnegative expected reward is to base their answers to the query on a single jointly-understood tiling that meets the constraints of the tiling problem.

This design is complicated because the DEC-MDP state set itself cannot remember which tiling locations were selected (this would cause exponential blowup in the size of the state set, but our reduction must expand the problem size at most polynomially — we note that the tiling grid itself is not part of the tiling problem size, only the compactly represented grid size n is in the problem specification); the state will only contain certain limited information about the relative locations of the two tile positions. The difficulty of the design is also increased by the fact that any information remembered about the specified tiling locations must be shared with at least one of the agents to satisfy the joint observability requirement. To deal with these two issues, we have designed the DEC-MDP to pass through the following phases (a formal description follows later):

Select Phase **Select two bit indices and values, each identifying a bit position and the value at that position in the location given to one of the agents**. These are the only bits that are remembered in the state set from the locations given to the agents in the next phase — the other location bits are generated and forgotten by the process.

The bit values remembered are called value-1 and value-2, and the indices to which these values correspond are called index-1 and index-2. Bit index-1 of the address given to agent 1 will have the value value-1, and likewise for index-2, value-2, and agent 2.

Generate Phase **Generate two tile locations at random, revealing one to each agent.** The bits selected in the above select phase are used, and the other location bits are generated at random and immediately "forgotten" by the DEC-MDP state set.

Query Phase **Query each agent for a tile type to place in the location that was specified to that agent.** These tile types are remembered in the state.

Echo Phase **Require the agents to echo the tile locations they received in the generate phase bit by bit.** In order to enforce the accuracy of these location echoes, the DEC-MDP is designed to yield a negative reward if the bit remembered from the original location generation is not correctly echoed (the DEC-MDP is designed to ensure that each agent cannot know which bit is being checked in its echoes). As the agents echo the bits, the process computes state information representing whether the locations are equal or adjacent horizontally or vertically, and whether the agents' locations are both $(0, 0)$ (again, we cannot just remember the location bits because it would force an exponential state set). The echo phase allows us to compute state information about adjacency/equality of the locations *after* the tile types have been chosen, so that the agents' tile choices cannot depend on this information. This is critical in making the reduction correct.

Test Phase **Check whether the tile types provided in the query phase come from a single consistent tiling.** In other words, check that if the agents were asked for the same location they gave the same tile types during query, if they were asked for adjacent locations they

gave types that satisfy the relevant adjacency constraints, and if the agents were both queried for location $(0, 0)$ they both gave tile type tile-0. The process gives a zero reward only if the tile types selected during the query phase meet any applicable constraints as determined by the echoed location bits. Otherwise, a negative reward is obtained.

Note that because we are designing a DEC-MDP, we are required to maintain joint observability: the observations given to the agents at each time step must be sufficient to reconstruct all aspects of the DEC-MDP state at that time step. In particular, the bit indices and values selected in the select phase must be known to the agents (jointly), as well as the information computed in the echo phase regarding the relative position of the two locations.

We achieve this joint observability by making all aspects of the DEC-MDP state observable to both agents, except for the indices and values selected in the select phase and the tile types that are given by the agents (and stored by the process) during the query phase. Each agent can observe which bit index and value are being remembered from the *other* agent's location, and each agent can observe the stored tile type it gave (but not the tile type given by the other agent). Because each agent can see what bit is saved from the other agent's location, we say that one location bit of each agent's location is *visible* to the other agent.

We call the five phases just described "select," "generate," "query," "echo," and "test" in the development below. A formal presentation of the DEC-MDP just sketched follows below, but first we outline the proof that this approach represents a correct reduction.

3.4.3 Overview of the Correctness Proof

Here we give an overview of our argument that the reduction sketched above is correct in the sense that there exists a policy that achieves expected total reward zero at the start state if and only if there is a solution to the tiling problem we started with.

It is straightforward to show that if there exists a consistent tiling there must exist a policy achieving zero reward. The agents need only "agree on" a consistent tiling ahead of time, and base their actions on the agreed upon tiling (waiting during selection and generation, giving the tile type present at the generated location during query, faithfully echoing the generated location during echo, and then waiting during test — at each point being guaranteed a zero reward by the structure of the problem). Note that it does not matter how expensive it might be to find and represent a consistent tiling or to carry out the policy just described because we are merely arguing for the existence of such a policy.

We now outline the proof of the harder direction, that if there is no consistent tiling then there is no policy achieving expected reward zero. Note that since all rewards are nonpositive, any chance of receiving any negative reward forces the expected total reward to be negative.

Consider an arbitrary policy that yields expected reward zero. Our argument rests on the following claims, which will be proved as lemmas in Section 3.4.5:

Claim 1. The policy must repeat the two locations correctly during the echo phase.

Claim 2. When executing the policy, the agents' selected actions during the query phase determine a single tiling, as follows. We define a query situation to be *dangerous* to an agent if and only if the observable bit value of the other agent's location (in the observation history) agrees with the bit value at the same index in the agent's own location (so that as far as the agent in danger knows, the other agent is being queried about the same location).

During dangerous queries, the tile type selected by the agent in danger must depend only on the location queried (and not on the index or value of the bit observed from the other agent, on any other observable information, or on which agent is selecting the tile type). The agents' selected actions for dangerous queries thus determine a single tiling.

Claim 3. The single tiling from Claim 2 is a consistent tiling.

Claim 3 directly implies that if there is no consistent tiling, then all policies have negative expected reward, as desired.

3.4.4 Formal Presentation of the Reduction

Now we give the two-agent DEC-MDP D that is constructed from the selected tiling instance $\langle L, H, V, n\rangle$. We assume throughout that n is a power of two. It is straightforward to modify the proof to deal with the more general case — one way to do so is summarized briefly in a later section.

The State Set

We describe the state set S of D below by giving a sequence of finite-domain "state variables," and then taking the state set to be the set of all possible assignments of values to the state variables. We list the variables in three groups: the first group is observable to both agents, the second group only to agent 1, and the third group only to agent 2.

The first variable that is observable to both agents is the current phase of the process, $\mathsf{phase} \in \{\mathsf{select}, \mathsf{gen}, \mathsf{query}, \mathsf{echo}, \mathsf{test}\}$. Next is the index of the next location bit to be generated/echoed, $\mathsf{index} \in \{0, \ldots, 2\log n\}$. There is also a variable that is eventually true if both tile location are (0,0). This is denoted $\mathsf{origin} \in \{\mathsf{yes}, \mathsf{no}\}$. Finally, we have a variable that keeps track of the relative tile positions during the echo phase, $\mathsf{rel\text{-}pos} \in Q$. This variable evolves according to the transition rules of a

special finite-state automaton (FSA) with state set Q. This FSA will be described in the following subsection.

The variables observed only by agent 1 include the index of the bit remembered for agent 2, the value of the bit remembered for agent 2, the bit for transmitting a tile position to agent 1, and the tile type selected by agent 1 in the query phase. These are denoted $\mathsf{index\text{-}2} \in \{0, \ldots, 2\log n - 1\}$, $\mathsf{value\text{-}2} \in \{0, 1\}$, $\mathsf{pos\text{-}bit\text{-}1} \in \{0, 1\}$, and $\mathsf{tile\text{-}sel\text{-}1} \in L$, respectively. Similarly, only agent 2 observes the variables $\mathsf{index\text{-}1} \in \{0, \ldots, 2\log n - 1\}$, $\mathsf{value\text{-}1} \in \{0, 1\}$, $\mathsf{pos\text{-}bit\text{-}2} \in \{0, 1\}$, and $\mathsf{tile\text{-}sel\text{-}2} \in L$.

We write a state by enumerating its variable values, e.g., as follows:

$$\langle \mathsf{gen}, 3, \mathsf{yes}, q_0; 4, 0, 1, \mathsf{tile\text{-}1}; 5, 1, 0, \mathsf{tile\text{-}3} \rangle \in S.$$

Semicolons are used to group together variables that have the same observability properties. We can represent sets of states by writing sets of values in some of the components of the tuple rather than just values. The "*" symbol is used to represent the set of all possible values for a component. We sometimes use a state variable as a function from states to domain values for that variable. For instance, if q matches $\langle \mathsf{gen}, *, *, *; *, *, *, *; *, *, *, * \rangle$, then we will say $\mathsf{phase}(q) = \mathsf{gen}$.

The initial state s_0 is as follows: $\langle \mathsf{select}, 0, \mathsf{yes}, q_0; 0, 0, 0, \mathsf{tile\text{-}0}; 0, 0, 0, \mathsf{tile\text{-}0} \rangle$.

The Action Sets and Table of Transition Probabilities

We must allow "wait" actions, "zero" and "one" actions for echoing location address bits, and tile type actions from the set of tile types L for answering during the query phase. We therefore take the action sets $A_1 = A_2$ to be $\{\mathsf{wait}, 0, 1\} \cup L$.

We give the transition distribution $P(s, a_1, a_2, s')$ for certain action pairs a_1, a_2 for certain source states s. For any source-state/action-pair combination not covered by the description below, the action pair is taken to cause a probability 1.0 self-transition back to the source state. The combinations not covered are not reachable

from the initial state under any joint policy. Also, we note that the FSA-controlled state component rel-pos does not change from its initial state q_0 until the echo phase.

Select Phase. This is the first step of the process. In this step, the process chooses, for each agent, which of that agent's bits it will be checking in the echo phase. The value of that bit is also determined in this step. Transition probabilities when phase = select are given as follows.

$P(s, a_1, a_2, s') = \frac{1}{(4 \log n)^2}$ in the following situations:

$s = s_0 = \langle \mathsf{select}, 0, \mathsf{yes}, q_0; 0, 0, 0, \mathsf{tile\text{-}0}; 0, 0, 0, \mathsf{tile\text{-}0} \rangle$,

$s' = \langle \mathsf{gen}, 0, \mathsf{yes}, q_0; i_2, v_2, 0, \mathsf{tile\text{-}0}; i_1, v_1, 0, \mathsf{tile\text{-}0} \rangle$,

$i_1, i_2 \in \{0, \ldots, 2 \log n - 1\}$, and

$v_1, v_2 \in \{0, 1\}$.

Generate Phase. During these steps, the two tile positions are chosen bit by bit. Note that we have to check for whether we are at one of the bits selected during the select phase, so that the value of the bit is the same as the value chosen during selection. Transition probabilities when phase = generate are given as follows. The second case describes the deterministic transition from the generate phase to the query phase.

$P(s, a_1, a_2, s') = \frac{1}{h}$ in the following situations:

$s = \langle \mathsf{gen}, k, \mathsf{yes}, q_0; i_2, v_2, *, \mathsf{tile\text{-}0}; i_1, v_1, *, \mathsf{tile\text{-}0} \rangle$ with $0 \leq k \leq 2 \log n - 1$,

$s' = \langle \mathsf{gen}, k+1, \mathsf{yes}, q_0; i_2, v_2, b_1, \mathsf{tile\text{-}0}; i_1, v_1, b_2, \mathsf{tile\text{-}0} \rangle$, where

$b_1 = v_1$ if $k = i_1$, else b_1 is either 0 or 1,

$b_2 = v_2$ if $k = i_2$, else b_2 is either 0 or 1, and

h is the number of allowed settings of b_1, b_2 from above.

$P(s, a_1, a_2, s') = 1$ in the following situations:

$s = \langle \mathsf{gen}, 2 \log n, \mathsf{yes}, q_0; i_2, v_2, *, \mathsf{tile\text{-}0}; i_1, v_1, *, \mathsf{tile\text{-}0} \rangle$, and

$s' = \langle \mathsf{query}, 0, \mathsf{yes}, q_0; i_2, v_2, 0, \mathsf{tile\text{-}0}; i_1, v_1, 0, \mathsf{tile\text{-}0} \rangle$.

Query Phase. The query phase consists of just one step, during which each agent chooses a tile type. Transition probabilities when $\mathsf{phase} = \mathsf{query}$ are given as follows.

$P(s, a_1, a_2, s') = 1$ in the following situations:

$$s = \langle \mathsf{query}, 0, \mathsf{yes}, q_0; i_2, v_2, 0, \mathsf{tile\text{-}0}; i_1, v_1, 0, \mathsf{tile\text{-}0} \rangle,$$

$$t_1 = \begin{cases} a_1 & \text{if } a_1 \in L \\ \mathsf{tile\text{-}0} & \text{otherwise} \end{cases}, \qquad t_2 = \begin{cases} a_2 & \text{if } a_2 \in L \\ \mathsf{tile\text{-}0} & \text{otherwise} \end{cases}, \text{ and}$$

$$s' = \langle \mathsf{echo}, 0, \mathsf{yes}, q_0; i_2, v_2, 0, t_1; i_1, v_1, 0, t_2 \rangle.$$

Echo Phase. During the echo phase the agents are asked to repeat back the addresses seen in the generate phase, and information about the relative position of the addresses is calculated by the FSA described below and recorded in the state. The FSA is accessed here using the function FSANEXT described below. Transition probabilities when $\mathsf{phase} = \mathsf{echo}$ are given as follows.

$P(s, a_1, a_2, s') = 1$ in the following situations:

$$s = \langle \mathsf{echo}, k, o, q; i_2, v_2, 0, t_1; i_1, v_1, 0, t_2 \rangle,$$

$$b_1 = \begin{cases} a_1 & \text{if } a_1 \in \{0, 1\} \\ \mathsf{0} & \text{otherwise} \end{cases}, \qquad b_2 = \begin{cases} a_2 & \text{if } a_2 \in \{0, 1\} \\ \mathsf{0} & \text{otherwise} \end{cases},$$

$$s' = \langle p, k', o', \mathrm{FSANEXT}(q, b_1, b_2); i_2, v_2, 0, t_1; i_1, v_1, 0, t_2 \rangle,$$

$$\text{where } p, k' = \begin{cases} \mathsf{echo}, k+1 & \text{for } 0 \le k < 2\log n - 1 \\ \mathsf{test}, 0 & \text{for } k = 2\log n - 1 \end{cases}, \text{ and}$$

$o' = \mathsf{yes}$ if and only if ($o = \mathsf{yes}$ and $a_1 = a_2 = 0$).

Test Phase. The test phase consists of just one step terminating the process in a zero-reward absorbing state.

$P(s, a_1, a_2, s') = 1$ in the following situations:

$s = \langle \mathsf{test}, 0, *, *; *, *, 0, *; *, *, 0, * \rangle$, and

$s' = \langle \mathsf{test}, 0, \mathsf{yes}, q_0; 0, 0, 0, \mathsf{tile\text{-}0}; 0, 0, 0, \mathsf{tile\text{-}0} \rangle$.

Finite-State Automaton. Here we describe the domain of the rel-pos state component and how the component's value evolves during the echo phase based on the bit pairs chosen by the agents (recall that it remains fixed during the other phases). The component is controlled by a deterministic finite state automaton (FSA) with a state set Q of size polylogarithmic in n. The state set is assumed to include four distinguished subsets of states: apart, equal, hor, and ver. Each subset corresponds to a possible relation between the two agents' echoed tile positions. The automaton takes as input the string of bit pairs (the alphabet for the automaton consists of the four symbols [00], [01], [10], [11], with the first component of each symbol representing the bit produced by agent 1 and the second component representing the bit produced by agent 2). This automaton is the cross product of three individual automata, each of which keeps track of a different piece of information about the two tile positions represented by the sequence of bit pairs. These automata are described as follows:

1. Equal Tile Positions

   This automaton computes whether the two tile positions produced are equal or not. Consider the following regular expression:

   $$([00] + [11])^*.$$

   There is a constant-sized FSA corresponding to the above expression that, on inputs of length $2 \log n$, ends in an accept state if and only if $(x_1, y_1) = (x_2, y_2)$, where (x_1, y_1) is the tile position represented by the sequence of bits given by agent 1, and (x_2, y_2) is the tile position represented by the sequence of bits given by agent 2.

2. Horizontally Adjacent Tile Positions

   This automaton computes whether the second tile position is horizontally adjacent to the first tile position by a single increment in the x coordinate. Its regular expression is as follows:

$$[10]^*[01]([00]+[11])^* \underbrace{([00]+[11])\cdots([00]+[11])}_{\log n}.$$

   There is an $O(\log n)$-sized FSA corresponding to the above expression that, on inputs of length $2\log n$, ends in an accept state if and only if $(x_1+1, y_1) = (x_2, y_2)$, where x_1, y_1, x_2, and y_2 are as in the description of the first automaton. (We note that it is not always the case that a regular expression has a corresponding FSA that is only polynomially bigger. However, for all the regular expressions we consider this property does hold.)

3. Vertically Adjacent Tile Positions

   This automaton computes whether the second tile position is vertically adjacent to the first tile position by a single increment in the y coordinate. Its regular expression is as follows:

$$\underbrace{([00]+[11])\cdots([00]+[11])}_{\log n}[10]^*[01]([00]+[11])^*.$$

   There is an $O(\log n)$-sized FSA corresponding to the above expression that, on inputs of length $2\log n$, ends in an accept state if and only if $(x_1, y_1{+}1) = (x_2, y_2)$ where x_1, y_1, x_2, and y_2 are as in the descriptions of the previous two automata.

We can take the cross product of these three automata to get a new automaton with size $O((\log n)^2)$. Let accept_1, accept_2, and accept_3 be the sets of accept states from the three component automata, respectively, and let reject_1, reject_2, and reject_3

be the corresponding sets of reject states. From these sets we construct distinguished sets apart, equal, hor, and ver of cross-product automaton states as follows.

$$\begin{aligned}
\mathsf{apart} &= \text{reject}_1 \times \text{reject}_2 \times \text{reject}_3. \\
\mathsf{equal} &= \text{accept}_1 \times \text{reject}_2 \times \text{reject}_3. \\
\mathsf{hor} &= \text{reject}_1 \times \text{accept}_2 \times \text{reject}_3. \\
\mathsf{ver} &= \text{reject}_1 \times \text{reject}_2 \times \text{accept}_3.
\end{aligned}$$

The rest of the automaton's states comprise the set Q'. Let $q_{1,0}$, $q_{2,0}$, and $q_{3,0}$ denote the start states of the three component automata. We define the start state of the cross-product automaton to be the state $q_0 = \langle q_{1,0}, q_{2,0}, q_{3,0} \rangle$.

We now define two functions based on this automaton. One function takes as input the state of the automaton and a bit pair, and returns the next state of the automaton. The second function takes as input a pair of bit strings of the same length and returns the state that the automaton will be in starting from its initial state and reading symbols formed by the corresponding bits in the two strings in sequence.

Definition: For $q \in Q$ and $a_1, a_2 \in \{0, 1\}$, $\text{FSANEXT}(q, a_1, a_2) = q'$, where $q' \in Q$ is the resulting state if the automaton starts in state q and reads the input symbol $[a_1 a_2]$.

Definition: The function FSA is defined inductively as follows:

$$\text{FSA}(\epsilon, \epsilon) = q_0.$$

$$\text{FSA}(b_0 \cdots b_{k+1}, c_0 \cdots c_{k+1}) = \text{FSANEXT}(\text{FSA}(b_0 \cdots b_k, c_0 \cdots c_k)), b_{k+1}, c_{k+1}).$$

Note that the range of FSA for inputs of length $2 \log n$ is $\mathsf{apart} \cup \mathsf{equal} \cup \mathsf{hor} \cup \mathsf{ver}$.

In this proof, we assume that the TILING grid size n was an exact power of two. We note that the proof can be adapted by adding two components to the cross-product FSA described here, where the two new components are both FSAs over the

same alphabet. The first new component accepts a string only when both x_1 and y_1 (as described above) are less than n (so that the tiling location represented by (x_1, y_1) is in the tiling grid). The second new component behaves similarly for (x_2, y_2). The DEC-MDP can then be constructed using the smallest power of two larger than n, but modified so that whenever either new component of the FSA rejects the (faithfully) echoed bit sequences, then the process gives a zero reward regardless of the tile types returned during query.

Each new component can be viewed as an FSA over a {0,1} alphabet, because each focuses either on just the agent 1 echoes or on just the agent 2 echoes. We describe the FSA for checking that x_1 is less than n — constructing the two components is then straightforward. Suppose that $k = \lceil \log n \rceil$ is the number of bits in the binary representation of n, and that the bits themselves are given from least to most significant as $b_1 \cdots b_k$. Suppose also that there are j different bits equal to 1 among $b_1 \cdots b_k$, and that these bits are at indices $i_1, \ldots, i_j$. We can then write a regular expression for detecting that its input of k bits from least to most significant represents a number in binary that is strictly less than n:

$$\left[(0+1)^{i_1-1}0\, b_{i_1+1} \cdots b_k\right] + \left[(0+1)^{i_2-1}0\, b_{i_2+1} \cdots b_k\right] + \cdots + \left[(0+1)^{i_j-1}0\, b_{i_j+1} \cdots b_k\right].$$

It can be shown that this regular expression has an equivalent FSA of size $O((\log n)^2)$.

The Reward Function

We now describe the reward function for D. The reward $R(s, a_1, a_2, s')$ given when transitioning from state s to state s' taking action pair a_1, a_2 is -1 in any situation except those situations matching one of the following patterns. Roughly, we give zero reward for waiting during select and generate, for answering with a tile type during query, for echoing a bit consistent with any remembered information during echo, and for having given tile types satisfying the relevant constraints during test. The relevant

constraints during test are determined by the rel-pos state component computed by the FSA during the echo phase.

$R(s, a_1, a_2, s') = 0$ if and only if one of the following holds:

Select phase: $s = \langle \mathsf{select}, *, *, *; *, *, *, *; *, *, *, * \rangle$
and $a_1 = a_2 = \mathsf{wait}$.

Generate phase: $s = \langle \mathsf{gen}, *, *, *; *, *, *, *; *, *, *, * \rangle$
and $a_1 = a_2 = \mathsf{wait}$.

Query phase: $s = \langle \mathsf{query}, *, *, *; *, *, *, *; *, *, *, * \rangle$
and both $a_1 \in L$ and $a_2 \in L$.

Echo phase: $s = \langle \mathsf{echo}, k, *, *; i_2, v_2, *, *; i_1, v_1, *, * \rangle$
and $a_1, a_2 \in \{0, 1\}$,
where $(a_1 = v_1$ or $k \neq i_1)$ and $(a_2 = v_2$ or $k \neq i_2)$.

Test Phase (i): $s = \langle \mathsf{test}, *, o, \mathsf{equal}; *, *, *, t_1; *, *, *, t_1 \rangle$
and $a_1 = a_2 = \mathsf{wait}$, where $(o = \mathsf{no}$ or $t_1 = \mathsf{tile\text{-}0})$.

Test Phase (ii): $s = \langle \mathsf{test}, *, *, \mathsf{hor}; *, *, *, t_1; *, *, *, t_2 \rangle$
and $a_1 = a_2 = \mathsf{wait}$, where $\langle t_1, t_2 \rangle \in H$.

Test Phase (iii): $s = \langle \mathsf{test}, *, *, \mathsf{ver}; *, *, *, t_1; *, *, *, t_2 \rangle$
and $a_1 = a_2 = \mathsf{wait}$, where $\langle t_1, t_2 \rangle \in V$.

Test Phase (iv): $s = \langle \mathsf{test}, *, *, \mathsf{apart}; *, *, *, *; *, *, *, * \rangle$
and $a_1 = a_2 = \mathsf{wait}$.

Observations, Threshold, and Horizon

The first four component fields of each state description are fully visible to both agents. The last eight state component fields are split into two groups of four, each

group visible only to one agent. We therefore take the agent 1 observations Ω_1 to be partial assignments to the following state variables: phase, index, origin, rel-pos, index-2, value-2, pos-bit-1, and tile-sel-1. Similarly, the observations Ω_2 are partial assignments to the following state variables: phase, index, origin, rel-pos, index-1, value-1, pos-bit-2, and tile-sel-2. The observation distribution $O(s, a_1, a_2, s', o_1, o_2)$ simply reveals the indicated portion of the just-reached state s' to each agent deterministically.

We say that an observation sequence is p-phase if the sequence matches the pattern $*\langle p, *, *, *; *, *, *, *\rangle$, where the first "*" stands for any observation sequence. Here, p can be any of gen, query, echo, or test.

We take the horizon H to be $4 \log n + 4$, because the process spends one step in each of the select, query, and test phases, $2 \log n + 1$ steps in the generate phase, and $2 \log n$ steps in the echo phase. We take the threshold value K to be 0. This completes the construction of the DEC-MDP by polynomial-time reduction from the selected tiling instance. An example of a zero-reward trajectory of the process is shown in Figure 3.2. We now turn to correctness.

3.4.5 Formal Correctness Argument

Next we show that the reduction presented above is indeed correct. Our main claim is that there exists a policy that achieves expected total reward zero at the start state if and only if there is a solution to the tiling problem we started with.

To make our notation easier to read, we define the following abbreviations.

Definition: Given an observation sequence $\overline{o}_1$ over Ω_1, we write $\mathrm{loc}_1(\overline{o}_1)$ for the location value represented by the bits transmitted to agent 1 in the generate phase of the process. We note that during the select and generate phases this value may be only partially specified (because not all of the bits have been generated). More precisely, $\mathrm{loc}_1(\overline{o}_1) = b_k \cdots b_0$, where the b_i values are chosen by the first match of the following sequence in $\overline{o}_1$ (with k as large as possible while allowing a match):

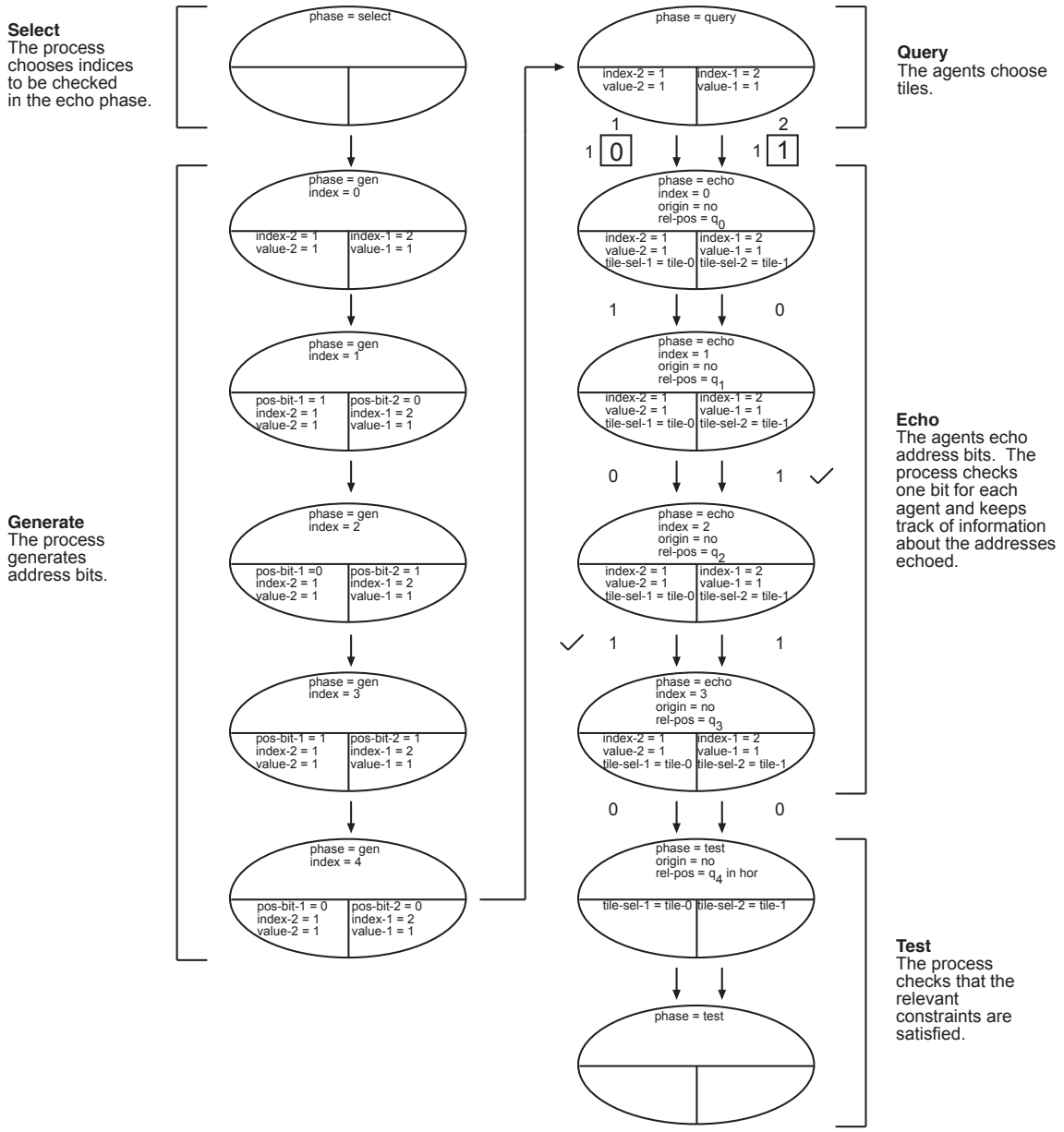


Figure 3.2: An example of a zero-reward trajectory of the process constructed from the tiling example given in Figure 3.1. The total reward is zero because the agents echo the "checked" bits correctly and choose tiles that do not violate any constraints, given the two addresses that are echoed. (For clarity, some state components are not shown.)

$$\langle \mathsf{gen}, 1, *, *; *, *, b_0, * \rangle \cdots \langle \mathsf{gen}, k+1, *, *; *, *, b_k, * \rangle.$$

We define $\mathrm{loc}_2(\overline{o}_2)$ similarly. In addition, we define $\mathrm{bit}(i, l)$ to be b_i, where $b_k, \ldots, b_0$ is the binary coding of the (possibly only partially specified) location l — we take $\mathrm{bit}(i, l)$ to be undefined if the bit i is not specified in location l.

By abuse of notation we treat $\mathrm{loc}_1(\overline{o}_1)$ (or $\mathrm{loc}_2(\overline{o}_2)$) as a tiling location (x, y) sometimes (but only when it is fully specified) and as a bit string at other times.

The easier direction of correctness is stated in the following lemma.

Lemma 1 *If there exists a consistent tiling, then there must exist a policy achieving zero expected total reward.*

Proof. We assume there exists at least one consistent tiling, and we select a particular such mapping f. We describe a policy $\delta = \langle \delta_1, \delta_2 \rangle$ that achieves zero expected reward at the initial state. δ_1 is a mapping from sequences of observations in Ω_1 to actions in A_1, and δ_2 from sequences over Ω_2 to actions in A_2. Only the reachable part of the mapping δ_1 is specified below — any unspecified observation sequence maps to the action wait. We note that δ_1 and δ_2 are symmetric.

$\delta_1(\overline{o}_1) = a_1$ when one of the following holds:

Select phase: $\overline{o}_1 = \langle \mathsf{select}, *, *, *; *, *, *, * \rangle$ and $a_1 = \mathsf{wait}$.

Generate phase: $\overline{o}_1 = *\langle \mathsf{gen}, *, *, *; *, *, *, * \rangle$ and $a_1 = \mathsf{wait}$.

Query phase: $\overline{o}_1 = *\langle \mathsf{query}, *, *, *; *, *, *, * \rangle$ and $a_1 = f(\mathrm{loc}_1(\overline{o}_1))$.

Echo phase: $\overline{o}_1 = *\langle \mathsf{echo}, k, *, *; *, *, *, * \rangle$ and $a_1 = \mathrm{bit}(k, \mathrm{loc}_1(\overline{o}_1))$.

Test phase: $\overline{o}_1 = *\langle \mathsf{test}, *, *, *; *, *, *, * \rangle$ and $a_1 = \mathsf{wait}$.

The local policy δ_2 is defined identically to δ_1 except that loc_1 is replaced by loc_2.

We first characterize the set of all reachable states from s_0 under the policy δ. We then note that taking the action prescribed by δ from any of these states yields a reward of zero. Thus, the value of δ from the start state is 0.

It is straightforward to show by induction that $\overline{P}_\delta(s_0, \overline{o}_1, \overline{o}_2, s)$ is zero except where one of the following patterns applies:

- $s = s_0 = \langle \mathsf{select}, 0, \mathsf{yes}, q_0; 0, 0, 0, \mathsf{tile\text{-}0}; 0, 0, 0, \mathsf{tile\text{-}0} \rangle$.
- $s = \langle \mathsf{gen}, k, \mathsf{yes}, q_0; i_2, v_2, *, \mathsf{tile\text{-}0}; i_1, v_1, *, \mathsf{tile\text{-}0} \rangle$, where

  $0 \leq k \leq 2 \log n$,

  $(k \leq i_1$ or $v_1 = \mathrm{bit}(i_1, \mathrm{loc}_1(\overline{o}_1)))$, and

  $(k \leq i_2$ or $v_2 = \mathrm{bit}(i_2, \mathrm{loc}_2(\overline{o}_2)))$.
- $s = \langle \mathsf{query}, 0, \mathsf{yes}, q_0; i_2, v_2, *, \mathsf{tile\text{-}0}; i_1, v_1, *, \mathsf{tile\text{-}0} \rangle$, where

  $v_1 = \mathrm{bit}(i_1, \mathrm{loc}_1(\overline{o}_1))$, and

  $v_2 = \mathrm{bit}(i_2, \mathrm{loc}_2(\overline{o}_2))$.
- $s = \langle \mathsf{echo}, k, o, q; i_2, v_2, *, t_1; i_1, v_1, *, t_2 \rangle$, where

  $0 \leq k \leq 2 \log n - 1$,

  $v_1 = \mathrm{bit}(i_1, \mathrm{loc}_1(\overline{o}_1))$,

  $v_2 = \mathrm{bit}(i_2, \mathrm{loc}_2(\overline{o}_2))$,

  $t_1 = f(\mathrm{loc}_1(\overline{o}_1))$,

  $t_2 = f(\mathrm{loc}_2(\overline{o}_2))$,

  $o = \mathsf{yes}$ if and only if $b_j = c_j = 0$ for $0 \leq j \leq k - 1$, with b_j and c_j as in the next item, and

  $q = \mathrm{FSA}(b_0 \cdots b_{k-1}, c_0 \cdots c_{k-1})$,

  with $b_0 \cdots b_{k-1}$ and $c_0 \cdots c_{k-1}$ the least significant bits of $\mathrm{loc}_1(\overline{o}_1)$ and $\mathrm{loc}_2(\overline{o}_2)$, respectively.

- $s = \langle \mathsf{test}, *, o, r; *, *, *, t_1; *, *, *, t_2 \rangle$, where

  $o = \mathsf{yes}$ if and only if $\mathrm{loc}_1(\overline{o}_1) = (0,0)$ and $\mathrm{loc}_2(\overline{o}_2) = (0,0)$,

  $r = \mathrm{FSA}(\mathrm{loc}_1(\overline{o}_1), \mathrm{loc}_2(\overline{o}_2))$,

  $t_1 = f(\mathrm{loc}_1(\overline{o}_1))$, and

  $t_2 = f(\mathrm{loc}_2(\overline{o}_2))$.

It can then be shown that the reward for any action prescribed by the policy δ given any of these reachable state/observation sequence combinations is zero given that f is a consistent tiling. $\square$

We now discuss the more difficult reverse direction of the correctness proof. In the following subsections, we prove Claims 1 to 3 from Section 3.4.3 to show that if there is a policy which achieves nonnegative expected reward for horizon H, then there is also a consistent tiling. *Throughout the remainder of the proof, we focus on a fixed, arbitrary policy δ that achieves zero expected reward.* Given this policy, we must show that there is a consistent tiling.

Proof of Claim 1

Before proving the first claim, we need to formalize the notion of "faithfulness during echo."

Definition: A pair of observation sequences $\langle \overline{o}_1, \overline{o}_2 \rangle$ over Ω_1 and Ω_2, respectively, is said to be *reachable* if $\overline{P}_\delta(s_0, \overline{o}_1, \overline{o}_2, s)$ is nonzero for some state s. An observation sequence $\overline{o}_1$ over Ω_1 is said to be *reachable* if there exists an observation sequence $\overline{o}_2$ over Ω_2 such that the pair of observation sequences $\langle \overline{o}_1, \overline{o}_2 \rangle$ is reachable. Likewise $\overline{o}_2$ over Ω_2 is reachable if there is some $\overline{o}_1$ over Ω_1 such that $\langle \overline{o}_1, \overline{o}_2 \rangle$ is reachable.

Definition: The policy $\delta = \langle \delta_1, \delta_2 \rangle$ is *faithful during echo* if it satisfies both of the following conditions for all indices k in $[0, 2\log n - 1]$, and all reachable observation sequence pairs $\langle \overline{o}_1, \overline{o}_2 \rangle$:

1. $\delta_1(\overline{o}_1) = \mathrm{bit}(k, \mathrm{loc}_1(\overline{o}_1))$ when $\overline{o}_1 = \langle *, *, *, *; *, *, *, * \rangle \cdots \langle \mathsf{echo}, k, *, *; *, *, *, * \rangle$.

2. $\delta_2(\overline{o}_2) = \mathrm{bit}(k, \mathrm{loc}_2(\overline{o}_2))$ when $\overline{o}_2 = \langle *, *, *, *; *, *, *, * \rangle \cdots \langle \mathsf{echo}, k, *, *; *, *, *, * \rangle$.

We say the policy δ *lies during echo* otherwise. If the two conditions listed above are satisfied for all indices k in $[0, d-1]$, where $0 \leq d \leq 2 \log n$, we say that the policy *faithfully echoes the first d bits*.

Much of our proof revolves around showing that the reachability of a pair of observation sequences is not affected by making certain changes to the sequences. We focus without loss of generality on changes to the observations of agent 2, but similar results hold for agent 1. The changes of particular interest are changes to the (randomly selected) value of the index-1 state component — this is the component that remembers which bit of agent 1's queried location will be checked during echo. It is important to show that agent 1 cannot determine which bit is being checked before that bit has to be echoed. To show this, we define a way to vary the observation sequences seen by agent 2 (preserving reachability) such that without changing the observations seen by agent 1 we have changed which agent 1 address bit is being checked. We now present this approach formally.

Definition: We say that an observation sequence $\overline{o}_1$ over Ω_1 is *superficially consistent* if the values of the index-2 component and the value-2 component do not change throughout the sequence, and the value of the tile-sel-1 component is tile-0 for generate-phase and query-phase observations and some fixed tile type in L for echo-phase and test-phase observations. Given a superficially consistent observation sequence $\overline{o}_1$, we can write index-2$(\overline{o}_1)$ and value-2$(\overline{o}_1)$ to denote the value of the indicated component throughout the sequence. In addition, we can write tile-sel-1$(\overline{o}_1)$ to denote the fixed tile type for echo-phase and test-phase observations (we take tile-sel-1$(\overline{o}_1)$ to be tile-0 if the sequence contains no echo- or test-phase observations). Corresponding definitions hold for observation sequences over Ω_2, replacing "1" by "2" and "2" by "1" throughout.

Note that any reachable observation sequence must be superficially consistent, but the converse is not necessarily true. The following technical definition is necessary so that we can discuss the relationships between observation sequences without assuming reachability.

Definition: We say that two superficially consistent observation sequences $\overline{o}_1$ over Ω_1 and $\overline{o}_2$ over Ω_2 are *compatible* if

$$\text{bit}(\textsf{index-1}(\overline{o}_2), \text{loc}_1(\overline{o}_1)) = \textsf{value-1}(\overline{o}_2) \text{ or this bit of } \text{loc}_1(\overline{o}_1) \text{ is not defined,}$$

and

$$\text{bit}(\textsf{index-2}(\overline{o}_1), \text{loc}_2(\overline{o}_2)) = \textsf{value-2}(\overline{o}_1) \text{ or this bit of } \text{loc}_2(\overline{o}_2) \text{ is not defined.}$$

Definition: Given an index i in $[0, 2\log n - 1]$, a reachable pair of observation sequences $\langle \overline{o}_1, \overline{o}_2 \rangle$, and an observation sequence $\overline{o}_2'$ over Ω_2, we say that $\overline{o}_2'$ is an *i-index variant of $\overline{o}_2$ relative to $\overline{o}_1$* when $\overline{o}_2'$ is any sequence compatible with $\overline{o}_1$ that varies from $\overline{o}_2$ only as follows:

1. index-1 has been set to i throughout the sequence,
2. value-1 has been set to the same value v throughout the sequence,
3. pos-bit-2 can vary arbitrarily from $\overline{o}_2$, and
4. for any echo- or test-phase observations, tile-sel-2 has been set to the tile type selected by δ on the query-phase prefix of $\overline{o}_2'$, or to tile-0 if δ selects a non-tile-type action on that query.

If the pos-bit-2 components of $\overline{o}_2$ and $\overline{o}_2'$ are identical, we say that $\overline{o}_2'$ is a *same-address* index variant of $\overline{o}_2$.

We note that, given a reachable pair of observation sequences $\langle \overline{o}_1, \overline{o}_2 \rangle$, there exists an i-index variant of $\overline{o}_2$ relative to $\overline{o}_1$, for any i in $[0, 2\log n - 1]$. This remains true even if we allow only same-address index variants. The following technical lemma

asserts that index variation as just defined preserves reachability under very general conditions.

Lemma 2 *Suppose δ is faithful during echo for the first k bits of the echo phase, for some k. Let $\langle \overline{o}_1, \overline{o}_2 \rangle$ be a reachable pair of observation sequences that end no later than the kth bit of echo (i.e., the last observation in the sequence has* index *no greater than k if it is an echo-phase observation), and let $\overline{o}_2'$ be an i-index variant of $\overline{o}_2$ relative to $\overline{o}_1$, for some i. If the observation sequences are echo-phase or test-phase, then we require that the index variation be a same-address variation. We can then conclude that $\langle \overline{o}_1, \overline{o}_2' \rangle$ is reachable.*

Proof. We need some new notation to carry out this proof. Given a state s, a state component c and corresponding value v from the domain of c, we define the state *"s with c set to v"* (written $s[c := v]$) to be the state s' that agrees with s at all state components except possibly c and has value v for state component c. We also write $\overline{o}_{1,j}$ for the first j observations in the sequence $\overline{o}_1$, and likewise $\overline{o}_{2,j}$ and $\overline{o}'_{2,j}$.

For any state s_j reachable while observing $\langle \overline{o}_{1,j}, \overline{o}_{2,j} \rangle$, we define a state s'_j that we will show is reachable while observing $\langle \overline{o}_{1,j}, \overline{o}'_{2,j} \rangle$, as follows:

$$s'_j = s_j[\textsf{index-1} := \textsf{index-1}(\overline{o}'_{2,j})][\textsf{value-1} := \textsf{value-1}(\overline{o}'_{2,j})][\textsf{tile-sel-2} := \textsf{tile-sel-2}(\overline{o}'_{2,j})].$$

We can now show by an induction on sequence length j that for any state s_j such that $\overline{P}_\delta(s_0, \overline{o}_{1,j}, \overline{o}_{2,j}, s_j)$ is nonzero, then $\overline{P}_\delta(s_0, \overline{o}_{1,j}, \overline{o}'_{2,j}, s'_j)$ is also nonzero. From this we can conclude that $\langle \overline{o}_1, \overline{o}_2' \rangle$ is reachable, as desired.

For the base case of this induction, we take j to be 1, so that the observation sequences involved all have length 1, ending in the generate phase with index equal to zero. Inspection of the definition of the transition probabilities P shows that changing index-1 and value-1 arbitrarily has no effect on reachability.

For the inductive case, we suppose some state s_j is reachable by $\langle \overline{o}_{1,j}, \overline{o}_{2,j} \rangle$, and that state s'_j is reachable by $\langle \overline{o}_{1,j}, \overline{o}'_{2,j} \rangle$. Let a_1 be $\delta(\overline{o}_{1,j})$, a_2 be $\delta(\overline{o}_{2,j})$, and a'_2 be $\delta(\overline{o}'_{2,j})$. We must show that for any state s_{j+1} such that $P(s_j, a_1, a_2, s_{j+1})$ is nonzero, $P(s'_j, a_1, a'_2, s'_{j+1})$ is also nonzero, for s'_{j+1}. This follows from the following observations:

- When $\mathsf{phase}(s_j)$ is select or generate, neither agent 2's action a_2 nor the values of $\mathsf{index\text{-}1}(s_j)$ or $\mathsf{value\text{-}1}(s_j)$ have any affect on $P(s_j, a_1, a_2, s_{j+1})$ being nonzero, as long as either index-1 is not equal to j or $\mathsf{pos\text{-}bit\text{-}1}(s_{j+1})$ equals $\mathsf{value\text{-}1}(s_j)$. However, this last condition is ensured to hold of the index-1 and value-1 components of s_j and s'_j by the compatibility of $\overline{o}'_2$ with $\overline{o}_1$.

- When $\mathsf{phase}(s_j)$ is query, the action a'_2 must equal the tile-sel-2 state component of s'_{j+1} by the definitions of s'_{j+1} and "index variant," and changes to the index-1 and value-1 components have no effect on $P(s_j, a_1, a_2, s_{j+1})$ being nonzero during the query phase.

- When $\mathsf{phase}(s_j)$ is echo, the actions a_2 and a'_2 must be a faithful echo of the location address bit indicated by the index state component (since we have assumed as part of out inductive hypothesis that δ is faithful during echo for at least j bits), and this bit's value does not vary between $\overline{o}_2$ and $\overline{o}'_2$ because if the observation sequences reach the echo phase we have the assumption that these are same-address variants. Thus $a_2 = a'_2$ during echo. Again, changes to the index-1 and value-1 components have no effect on $P(s_j, a_1, a_2, s_{j+1})$ being nonzero during the echo phase.

□

We are now ready to assert and prove Claim 1 from Section 3.4.3.

Lemma 3 *(Claim 1) δ is faithful during echo.*

Proof: We argue by induction that δ faithfully echoes all $2\log n$ address bits. As an inductive hypothesis, we assume that δ faithfully echoes the first k bits, where $0 \leq k < 2\log n$. Note that if k equals zero, this is a null assumption, providing an implicit base case to our induction. Now suppose for contradiction that δ lies during the $k+1$st step of the echo phase. Then one of the agents' policies must incorrectly echo bit $k+1$; we assume without loss of generality that this is so for agent 1, i.e., under some reachable observation sequence pair $\langle \overline{o}_1, \overline{o}_2 \rangle$ of length $2\log n + k + 2$, the policy δ_1 dictates that the agent choose action $1 - \text{bit}(k, \text{loc}_1(\overline{o}_1))$. Lemma 2 implies that the observation sequence pair $\langle \overline{o}_1, \overline{o}_2' \rangle$ is also reachable, where $\overline{o}_2'$ is any same-address $k+1$-index variant of $\overline{o}_2$ relative to $\overline{o}_1$.

Since all of the agent 1 observations are the same for both $\langle \overline{o}_1, \overline{o}_2 \rangle$ and $\langle \overline{o}_1, \overline{o}_2' \rangle$, when the latter sequence occurs, agent 1 chooses the same action $1 - \text{bit}(k, \text{loc}_1(\overline{o}_1))$ as given above for the former sequence, and a reward of -1 is obtained (because in this case it is bit $k+1$ that is checked). Therefore, the expected total reward is not zero, yielding a contradiction. $\square$

Proof of Claim 2

Now we move on to prove Claim 2 from Section 3.4.3. We show that δ can be used to define a particular mapping from tile locations to tile types based on "dangerous queries." In Section 3.4.3, we defined an agent 1 observation sequence to be "dangerous" if it reveals a bit of agent 2's queried location that agrees with the corresponding bit of agent 1's queried location (and vice versa for agent 2 observation sequences). We now present this definition more formally.

Definition: A query-phase observation sequence $\overline{o}_1$ over Ω_1 is *dangerous* if it is reachable and

$$\text{bit}(\textsf{index-2}(\overline{o}_1), \text{loc}_1(\overline{o}_1)) \;=\; \textsf{value-2}(\overline{o}_1).$$

Likewise, a query-phase sequence $\overline{o}_2$ over Ω_2 is dangerous if it is reachable and $\mathrm{bit}(\mathsf{index\text{-}1}(\overline{o}_2), \mathrm{loc}_2(\overline{o}_2)) = \mathsf{value\text{-}1}(\overline{o}_2)$.

Dangerous query-phase sequences are those for which the agent's observations are consistent with the possibility that the other agent has been queried on the same location. We note that for any desired query location l, and for either agent, there exist dangerous observation sequences $\overline{o}$ such that $\mathrm{loc}_k(\overline{o}) = l$. Moreover, such sequences still exist when we also require that the value of $\mathsf{index\text{-}}k(\overline{o})$ be any particular desired value (where k is the number of the non-observing agent).

Lemma 4 *Two same-length query-phase observation sequences, $\overline{o}_1$ over Ω_1 and $\overline{o}_2$ over Ω_2, are reachable together as a pair if and only if they are compatible and each is individually reachable.*

Proof: The "only if" direction of the theorem follows easily — the reachability part follows from the definition of reachability, and the compatibility of jointly reachable sequences follows by a simple induction on sequence length given the design of D.

The "if" direction can be shown based on the following assertions. First, a generate-phase observation sequence (for either agent) is reachable if and only if it matches the following pattern:

$$\langle \mathsf{gen}, 0, \mathsf{yes}, q_0; i, v, *, \mathsf{tile\text{-}0} \rangle \cdots \langle \mathsf{gen}, k, \mathsf{yes}, q_0; i, v, *, \mathsf{tile\text{-}0} \rangle,$$

for some k, i, and v — this can be established by a simple induction on sequence length based on the design of D. A similar pattern applies to the query phase.

Given two compatible reachable sequences of the same length, $\overline{o}_1$ and $\overline{o}_2$, we know by the definition of reachability that there must be some sequence $\overline{o}_2'$ such that $\langle \overline{o}_1, \overline{o}_2' \rangle$ is reachable. But given the patterns just shown for reachable sequences, $\overline{o}_2$ and $\overline{o}_2'$ can differ only in their choice of i, v, and in the address given to agent 2 via the

pos-bit-2 component. It follows that $\overline{o}_2$ is an i-index variant of $\overline{o}_2'$ relative to $\overline{o}_1$, for some i. Lemma 2 then implies that the pair $\langle \overline{o}_1, \overline{o}_2 \rangle$ is reachable as desired. □

Lemma 5 *(Claim 2) There exists a mapping f from tiling locations to tile types such that $f(loc_i(\overline{o})) = \delta_i(\overline{o})$ on all dangerous queries $\overline{o}$ over Ω_i for both agents ($i \in \{1, 2\}$).*

Proof: To prove the lemma, we prove that for any two dangerous query sequences $\overline{o}_i$ and $\overline{o}_j$ over Ω_i and Ω_j respectively for arbitrary $i, j \in \{1, 2\}$, if $\mathrm{loc}_i(\overline{o}_i) = \mathrm{loc}_j(\overline{o}_j) = l$ then $\delta_i(\overline{o}_i) = \delta_j(\overline{o}_j)$. This implies that for any such $\overline{o}_i$ we can take $f(l) = \delta_i(\mathrm{loc}_i(\overline{o}_i))$ to construct f satisfying the lemma. Suppose not. Then there must be a counterexample for which $i \neq j$ — because given a counterexample for which $i = j$, either $\overline{o}_i$ or $\overline{o}_j$ must form a counterexample with any dangerous query $\overline{o}_k$ over Ω_{1-i} such that $\mathrm{loc}_{1-i}(\overline{o}_k) = l$.

We can now consider a counterexample where $i \neq j$. Let $\overline{o}_1$ and $\overline{o}_2$ be dangerous (and thus reachable) sequences over Ω_1 and Ω_2, respectively, such that $\mathrm{loc}_1(\overline{o}_1) = \mathrm{loc}_2(\overline{o}_2)$ but $\delta_1(\overline{o}_1) \neq \delta_2(\overline{o}_2)$. Note that $\mathrm{loc}_1(\overline{o}_1) = \mathrm{loc}_2(\overline{o}_2)$ together with the fact that $\overline{o}_1$ and $\overline{o}_2$ are dangerous implies that $\overline{o}_1$ and $\overline{o}_2$ are compatible and thus reachable together (using Lemma 4).

The faithfulness of echo under δ (proven in Claim 1, Lemma 3) then ensures that the extension (there is a single extension because D is deterministic in the echo and test phases) of these observation sequences by following δ to the test phase involves a faithful echo. The correctness of the FSA construction then ensures that the rel-pos state component after this extension will have the value equal. The reward structure of D during the test phase then ensures that to avoid a negative reward the tile types given during query, $\delta_1(\overline{o}_1)$ and $\delta_2(\overline{o}_2)$, must be the same, contradicting our choice of $\overline{o}_1$ and $\overline{o}_2$ above and thus entailing the lemma. □

Proof of Claim 3 and our Main Hardness Theorem

We now finish the proof of our main theorem by proving Claim 3 from Section 3.4.3. We start by showing the existence of a useful class of pairs of dangerous observation sequences that are reachable together.

Lemma 6 *Given any two locations l_1 and l_2 sharing a single bit in their binary representations, there are dangerous observation sequences $\overline{o}_1$ over Ω_1 and $\overline{o}_2$ over Ω_2 such that:*

$$\begin{aligned} loc_1(\overline{o}_1) &= l_1, \\ loc_2(\overline{o}_2) &= l_2, \text{ and} \\ \langle \overline{o}_1, \overline{o}_2 \rangle &\text{ is reachable.} \end{aligned}$$

Proof: It is straightforward to show that there exist dangerous observation sequences $\overline{o}_1$ over Ω_1 and $\overline{o}_2$ over Ω_2 such that $\mathrm{loc}_1(\overline{o}_1) = l_1$ and $\mathrm{loc}_2(\overline{o}_2) = l_2$ as desired. In these sequences, both index-1 and index-2 are set throughout to the index of a single bit shared by l_1 and l_2. Since this bit is in common, these sequences are compatible, so by Lemma 4 they are reachable together. □

Lemma 7 *(Claim 3) The mapping f defined in Lemma 5 is a consistent tiling.*

Proof: We prove the contrapositive. If the mapping f is not a consistent tiling, then there must be some particular constraint violated by f. It is easy to show that any such constraint is tested during the test phase if $\mathrm{loc}_1(\overline{o}_1)$ and $\mathrm{loc}_2(\overline{o}_2)$ have the appropriate values. (The faithfulness during echo claim proven in Lemma 3 implies that the origin and rel-pos components on entry to the test phase will have the correct values for comparing the two locations). For example, if a horizontal constraint fails for f, then there must be locations (i, j) and $(i + 1, j)$ such that the tile types $\langle f(i, j), f(i + 1, j) \rangle$ are not in H — since these two locations share a bit (in fact, all the bits in j, at least) Lemma 6 implies that there are dangerous $\overline{o}_1$ and $\overline{o}_2$ with $\mathrm{loc}_1(\overline{o}_1) = (i, j)$ and $\mathrm{loc}_2(\overline{o}_2) = (i + 1, j)$ that are reachable together. During the test

phase, the tile-sel-1 and tile-sel-2 state components are easily shown to be $f(i, j)$ and $f(i + 1, j)$, and then the definition of the reward function for D ensures a reachable negative reward. The arguments for the other constraints are similar. □

Claim 3 immediately implies that there exists a consistent tiling whenever there exists a policy achieving zero expected total reward. This completes the proof of the other direction of our main complexity result. We have thus shown that there exists a policy that achieves expected reward zero if and only if there exists a consistent tiling, demonstrating that the two agent DEC-MDP problem is NEXP-hard.

Theorem 5 *The finite-horizon DEC-MDP problem with two agents is NEXP-hard.*

From Theorems 4 and 5 we get the following corollary, which establishes completeness for nondeterministic exponential time.

Corollary 1 *The finite-horizon DEC-POMDP problem is NEXP-complete.*

3.5 Discussion

Using the tools of worst-case complexity analysis, we analyzed the finite-horizon DEC-POMDP problem. Specifically, we proved that the problem is NEXP-complete, even when there are two agents that jointly observe the system state.

These results have some significant implications. First, unlike the single agent problems, the DEC-POMDP problem *provably* does not admit a polynomial-time algorithm, since P $\neq$ NEXP. Furthermore, assuming that EXP $\neq$ NEXP, the DEC-POMDP problem requires superexponential time to solve. This is in contrast to the POMDP problem, which can be solved in exponential time. And from the point of view of complexity theory, we have drawn a connection between work on Markov decision processes and the body of work in complexity theory that deals with the exponential jump in complexity due to decentralization [55, 2].

These complexity results can provide guidance for algorithm design. It was previously not clear whether there was a straightforward way to convert a DEC-POMDP to a POMDP and apply single agent techniques. We now know that, though this may not be impossible, there are complexity barriers to overcome in doing the transformation. In subsequent chapters, we actually are able to extend dynamic programming for POMDPs to the multiagent case. However, dynamic programming already has doubly exponential worst-case complexity for POMDPs, so our extension does not contradict these complexity results.

CHAPTER 4

CENTRALIZED DYNAMIC PROGRAMMING

Having considered the complexity of solving sequential decision-making problems, we now turn to algorithms. In particular, we focus on dynamic programming approaches, which are the most widely used. We first describe dynamic programming for the fully observable case. Next, we describe the more complicated extension to the partially observable case. This will provide a foundation for the multiagent dynamic programming algorithm to be described in the following chapter. Our focus from this point on will be on the infinite-horizon discounted problem, though many of the central ideas are also applicable to the finite-horizon problem.

4.1 Dynamic Programming for MDPs

Markov decision processes have a convenient structure that allows for sequential decomposition. Intuitively speaking, the expected total reward can be divided into the immediate reward and the expected total reward for the second step onwards. As we will see, this plays a role in policy evaluation, the development of an optimality principle, and the development of efficient optimal algorithms.

4.1.1 Policy Evaluation

Suppose we have a policy δ and would like to compute its value function, $V^\delta(s)$, for all states $s \in S$. This can be done by solving the following system of $|S|$ linear equations in $|S|$ unknowns:

$$V^\delta(s) = R(s,a) + \beta \sum_{s'} P(s'|s,a) V^\delta(s').$$

These equations can be solved directly using Gaussian elimination, or iteratively.

One useful way to define a policy using a value function is via one-step lookahead. The policy $\delta(s)$ for each state $s \in S$ is

$$\delta(s) = \text{argmax}_{a \in A} \left[R(s,a) + \beta \sum_{s' \in S} P(s'|s,a) V(s') \right].$$

4.1.2 Bellman's Equation

It was shown by Bellman [4] that the optimal value function for an MDP satisfies the following set of nonlinear equations for all $s \in S$:

$$V^*(s) = \max_a \left[R(s,a) + \beta \sum_{s'} P(s'|s,a) V^*(s') \right].$$

This is often referred to as *Bellman's equation*. As we will see below, Bellman's equation provides the foundation for efficient dynamic programming algorithms.

It turns out that the one-step lookahead policy with respect to this value function is an optimal policy. So we can also say that if the values and policies are greedy with respect to each other, then they are both optimal.

4.1.3 Value Iteration

The *value iteration* algorithm is simply the iterative solution of Bellman's equation. The algorithm is initialized with an arbitrary value function V^0. On each iteration, the value of each state is updated using the following *DP update*:

$$V^{t+1}(s) := \max_a \left[R(s,a) + \beta \sum_{s'} P(s'|s,a) V^t(s') \right].$$

This operator is a contraction in the max norm, and thus value iteration converges to optimality in the limit. The *Bellman residual* is defined as $\|V^{t+1} - V^t\|_\infty$. If

$$\frac{2\beta \|V^{t+1} - V^t\|_\infty}{1-\beta} \leq \epsilon,$$

Input: An initial value function V^0, and a parameter ϵ.

1. Perform a DP update, using the following update rule for each $s \in S$:

$$V^{t+1}(s) := \max_a \left[R(s,a) + \beta \sum_{s'} P(s'|s,a) V^t(s') \right].$$

2. Test for convergence, and goto 1 if criteria not met.

Output: An ϵ-optimal policy, determined using the rule:

$$\delta(s) = \text{argmax}_{a \in A} \left[R(s,a) + \beta \sum_{s' \in S} P(s'|s,a) V(s') \right].$$

Table 4.1: Value iteration for MDPs.

then $\|V^* - V^{t+1}\|_\infty \leq \epsilon$. This gives a way of detecting whether the value function is within ϵ of optimal. And as the value function approaches optimality, the one-step lookahead policy based on the value function approaches optimality. Thus, the algorithm can be stopped when a good enough policy has been found. A summary of the value iteration algorithm is given in Table 4.1.

Each iteration of the algorithm requires $O(|A||S|^2)$ operations, and the number of iterations required to get close to optimal is bounded above by a polynomial in $|S|$, $|A|$, and $1/(1-\beta)$.

4.1.4 Policy Iteration

With *policy iteration*, the policy is considered its own entity, apart from the value function. The algorithm is initialized with an arbitrary policy, and each iteration consists of policy evaluation followed by improvement. As explained above, the policy evaluation step consists of solving a system of linear equations. For the policy improvement step, a DP update is performed on the value function, and the new policy is determined by one-step lookahead.

Input: An initial policy δ.

1. Compute the value function V^{δ} for δ by solving a set of linear equations.
2. Perform a DP update to find a new best action for each state $s \in S$ and thus form a new policy δ'.
3. If $\delta \neq \delta'$, goto step 1, otherwise terminate.

Output: An optimal policy.

Table 4.2: Policy iteration for MDPs.

Howard showed that if the current policy is not optimal, an iteration produces a new policy which has a value at least as high for all states and higher for at least one state [32]. Since the number of policies is finite, this algorithm converges in a finite number of iterations. The algorithm is summarized in Table 4.2.

Policy iteration can take fewer iterations than value iteration to get a near-optimal policy, but each iteration takes longer due to the policy evaluation step. The policy improvement step takes $O(|A||S|^2)$ operations, and the policy evaluation step takes $O(|S|^3)$ operations. When the discount factor is taken into account, the number of iterations is polynomial. However, it is currently not known whether the number of iterations is polynomial, independent of the discount factor.

4.2 Value Iteration for POMDPs

In this section, we describe a version of value iteration that can be used to solve POMDPs optimally. This algorithm is more complicated than its MDP counterpart, and does not have efficiency guarantees. However, in practice it can provide significant leverage in solving POMDPs.

We begin by explaining how every POMDP has an equivalent MDP with a continuous state space. Next, we describe how the value functions for this MDP have

special structure that can be exploited. These ideas are central to the value iteration algorithm.

4.2.1 Belief State MDPs

A convenient way to summarize the observation history of an agent in a POMDP is through a *belief state*, which is a distribution over system states. As it receives observations, the agent can update its belief state and then remove its observations from memory. Let π denote a belief state, and let $\pi(s)$ represent the probability assigned to state s by π. If an agent chooses action a from belief state π and subsequently observes o, each component of the successor belief state obeys the equation

$$\pi'(s') = \frac{P(o|a,s') \sum_{s \in S} P(s'|s,a)\pi(s)}{P(o|\pi,a)},$$

where

$$P(o|\pi,a) = \sum_{s' \in S} \left[P(o|a,s') \sum_{s \in S} P(s'|s,a)\pi(s) \right].$$

Note that this is a simple application of Bayes' rule.

It was shown by Astrom [1] that a belief state constitutes a sufficient statistic for the agent's observation history, and it is possible to define an MDP over belief states as follows. A *belief-state MDP* is a tuple $\langle \Pi, A, T, R \rangle$, where

- Π is the set of distributions over S.
- A is the set of actions (same as before).
- $T(\pi, a, \pi')$ is the transition function, defined as

$$T(\pi,a,\pi') = \sum_{o \in O} P(\pi'|\pi,a,o)P(o|\pi,a).$$

- $R(\pi, a)$ is a reward function, defined as

$$R(\pi, a) = \sum_{s \in S} \pi(s) R(s, a).$$

When combined with belief-state updating, an optimal solution to this MDP can be used as an optimal solution to the POMDP from which it was constructed. However, since the belief state MDP has a continuous, $|S|$-dimensional state space, the techniques given in the section on MDPs are not immediately applicable. In the following sections, we describe how to implement a dynamic programming update in finite time.

4.2.2 Value Function Representation

The key result in making dynamic programming practical was proved by Smallwood and Sondik [64], who showed that the Bellman operator preserves piecewise linearity and convexity of a value function. Starting with a piecewise linear and convex representation of V^t, the value function V^{t+1} is piecewise linear and convex, and can be computed in finite time.

To represent a piecewise linear and convex value function, one need only store the value of each facet for each system state. Denoting the set of facets Γ, we can store $|\Gamma|$ $|S|$-dimensional vectors of real values. To go from the set of vectors to the value of a belief state, we use the equation

$$V(\pi) = \max_{\gamma} \sum_{s \in S} \pi(s) \gamma(s).$$

Figure 4.1 shows a piecewise linear and convex value function for a POMDP with two states.

Smallwood and Sondik proved that the optimal value function for a finite-horizon POMDP is piecewise linear and convex. The optimal value function for an infinite-

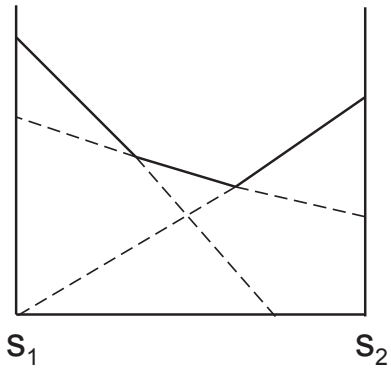


Figure 4.1: A piecewise linear and convex value function for a POMDP with two states.

horizon POMDP is convex, but may not be piecewise linear. However, it can be approximated arbitrarily closely by a piecewise linear and convex value function, and the value iteration algorithm constructs closer and closer approximations, as we shall see.

4.2.3 Pruning Vectors

Every piecewise linear and convex value function has a minimal set of vectors Γ that represents it. Of course, it is possible to use a non-minimal set to represent the same function. This is illustrated in Figure 4.2. Note that the removal of certain vectors does not change the value of any belief state. Vectors such as these are not necessary to keep in memory. Formally, we say that a vector γ is *dominated* if for all belief states π, there is a vector $\hat{\gamma} \in \Gamma \setminus \gamma$ such that $V(\pi, \gamma) \leq V(\pi, \hat{\gamma})$.

Because dominated vectors are not necessary, it would be useful to have a method for removing them. This task is often called *pruning*, and has an efficient algorithm based on linear programming. For a given vector γ, the linear program in Table 4.3 determines whether γ is dominated. If variables can be found to make ϵ is positive, then adding γ to the set improves the value function at some belief state. If not, then γ is dominated.

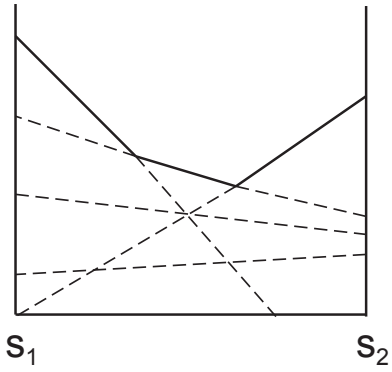


Figure 4.2: Non-minimal representation of a piecewise linear and convex value function for a POMDP.

Variables: ϵ, $\pi(s)$

Objective: Maximize ϵ.

Improvement constraints:

$$\forall \hat{\gamma} \quad \sum_s \pi(s)\hat{\gamma}(s) + \epsilon \leq \sum_s \pi(s)\gamma(s)$$

Probability constraints:

$$\sum_s \pi(s) = 1, \qquad \forall s \quad \pi(s) \geq 0$$

Table 4.3: The linear program for testing whether a vector γ is dominated.

This gives rise to a simple algorithm for pruning a set of vectors $\tilde{\Gamma}$ to obtain a minimal set Γ. The algorithm loops through $\tilde{\Gamma}$, removes each vector $\gamma \in \tilde{\Gamma}$, and solves the linear program using γ and $\tilde{\Gamma} \setminus \gamma$. If γ is not dominated, then it is returned to $\tilde{\Gamma}$.

A more efficient pruning technique was proposed by Lark [39]. With this technique, two sets of vectors, $\tilde{\Gamma}$ and Γ, are maintained. The set Γ is constructed by adding vectors from $\tilde{\Gamma}$ that are not dominated. The algorithm is given in detail in Table 4.4. The linear programs are smaller than in the simpler algorithm because vectors are tested against the set Γ instead of the complete set $\tilde{\Gamma}$.

Input: A set of vectors $\tilde{\Gamma}$ representing a value function.

1. Set Γ to be empty.
2. Repeat until $\tilde{\Gamma}$ is empty:
   (a) Choose a vector $\gamma \in \tilde{\Gamma}$, and solve the linear program of Table 4.3 with inputs γ and Γ.
   (b) If γ is dominated, remove it from $\tilde{\Gamma}$.
   (c) If γ is not dominated, take the belief state which solves the linear program, and find the vector $\gamma' \in \tilde{\Gamma}$ which optimizes the value at it. Remove γ' from $\tilde{\Gamma}$ and add it to Γ.

Output A minimal set of vectors Γ representing the same value function.

Table 4.4: Lark's method for finding a minimal set of vectors.

There is an important implementation detail missing from the description of Lark's algorithm in the table. Sometimes there is more than one vector which optimizes the value of a particular belief state. If a vector is chosen arbitrarily in the case of a tie, the set of vectors returned by the algorithm may not be minimal. Littman [41] showed that breaking ties based on lexicographic ordering results in a minimal set. A lexicographic ordering relies on a predetermined ordering of the states in S. If one vector has a higher value for the first state, it is chosen. In the case of a tie, the second state is considered, and so on.

4.2.4 The Dual Interpretation of Dominance

It turns out that there is an equivalent way to characterize dominance that can be useful. Recall that for a vector to be dominated, there does not have to be a single vector that has value at least as high for all states. It is sufficient for there to exist a set of vectors such that for all belief states, one of the vectors in the set has value at least as high as the vector in question.

It can be shown that such a set exists if and only if there is some convex combination of vectors that has value at least as high as the vector in question for all states.

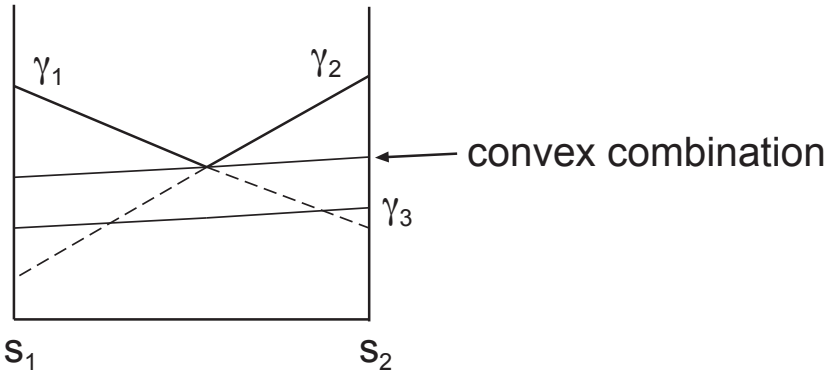


Figure 4.3: The dual interpretation of dominance. Vector γ_3 is dominated at all belief states by either γ_1 or γ_2. This is equivalent to the existence of a convex combination of γ_1 and γ_2 which dominates γ_3 for all belief states.

Variables: $\epsilon, x(\hat{\gamma})$

Objective: Maximize ϵ

Improvement constraints:

$$\forall s \quad V(s,\gamma) + \epsilon \leq \sum_{\hat{\gamma}} x(\hat{\gamma}) V(s,\hat{\gamma})$$

Probability constraints:

$$\sum_{\hat{\gamma}} x(\hat{\gamma}) = 1, \qquad \forall \hat{\gamma} \quad x(\hat{\gamma}) \geq 0$$

Table 4.5: The dual linear program for testing dominance for the vector γ. The variable $x(\hat{\gamma})$ represents $P(\hat{\gamma})$.

This is shown graphically in Figure 4.3. If we take the dual of the linear program for dominance given in the previous section, we get a linear program for which the solution is a vector of probabilities for the convex combination. This dual view of dominance was first used in a POMDP context by Poupart and Boutilier [58], and is useful for policy iteraion, as will be explained later.

4.2.5 Dynamic Programming Update

In this section, we describe how to implement a *dynamic programming update* to go from a value function V_t to a value function V_{t+1}. In terms of implementation, our aim is to take a minimal set of vectors Γ_t that represents V_t and produce a minimal set of vectors Γ_{t+1} that represents V_{t+1}.

Each vector that could potentially be included in Γ_{t+1} represents the value of an action a and assignment of vectors in Γ_t to observations. A combination of an action and transition rule will hereafter be called a *one-step policy*. The value vector for a one-step policy can be determined via the equation

$$\gamma_i^{t+1}(s) = R(s, \alpha(i)) + \beta \sum_{s',o} P(s'|s, \alpha(i)) P(o|\alpha(i), s') \gamma^t_{\tau(i,o)}(s'),$$

where i is the index of the vector, $\alpha(i)$ is its action, and $\tau(i, o)$ is the index of the vector in Γ_t to which to transition upon receiving observation o.

There are $|A||\Gamma_t|^{|\Omega|}$ possible one-step policies. A simple way to construct Γ_{t+1} is to evaluate all possible one-step policies and then apply a pruning algorithm such as Lark's method. Evaluating the entire set of one-step policies will hereafter be called performing an *exhaustive backup*. It turns out that there are ways to perform a dynamic programming update without first performing an exhaustive backup. Below we describe two approaches to doing this.

The first approach uses the fact that it is simple to find the optimal vector for any particular belief state. For a belief state π, an optimal action can be determined via the equation

$$\alpha = \text{argmax}_{a \in A} \left[R(\pi, a) + \beta \sum_{o \in \Omega} P(o|\pi, a) V^t(T(\pi|a, o)) \right].$$

For each observation o, there is a subsequent belief state, which can be computed using Bayes' rule. To get an optimal transition rule, $\tau(o)$, we take the optimal vector for the belief state corresponding to o.

Since the backed-up value function has finitely many vectors, there must be a finite set of belief states for which backups must be performed. Algorithms which identify these belief states include Smallwood and Sondik's "one-pass" algorithm [64], Cheng's linear support and relaxed region algorithms [12], and Kaelbling, Cassandra and Littman's Witness algorithm [34].

The second approach is based on generating and pruning sets of vectors. Instead of generating all vectors and then pruning, these techniques attempt to prune during the generation phase. The first algorithm along these lines was the incremental pruning algorithm [9]. Recently, improvements have been made to this approach [73, 18, 19].

It should be noted that there are theoretical complexity barriers for DP updates. Littman [42] showed that under certain widely-believed complexity-theoretic assumptions, there is no algorithm for performing a DP update that is worst-case polynomial in all the quantities involved. Despite this fact, dynamic programming updates have been successfully implemented as part of the value iteration and policy iteration algorithms, which will be described in the subsequent sections.

4.2.6 Value Iteration

To implement value iteration, we simply start with an arbitrary piecewise linear and convex value function, and proceed to perform DP updates. This corresponds to value iteration in the equivalent belief state MDP, and thus converges to an ϵ-optimal value function after a finite number of iterations.

Value iteration returns a value function, but a policy is needed for execution. As in the MDP case, we can use one-step lookahead, using the equation

$$\delta(\pi) = \text{argmax}_{a \in A} \left[\sum_{s \in S} R(s,a)\pi(s) + \beta \sum_{o \in \Omega} P(o|\pi,a) V(\tau(\pi,o,a)) \right],$$

where $\tau(\pi, o, a)$ is the belief state resulting from starting in belief state π, taking action a, and receiving observation o. We note that a state estimator must be used as well to track the belief state. Using the fact that each vector corresponds to a one-step policy, we can extract a policy from a value function in a simpler way:

$$\delta(\pi) = \alpha \left(\text{argmax}_k \sum_s \pi(s)\gamma_k(s) \right).$$

As in the completely observable case, the Bellman residual provides a bound on the distance to optimality. Recall that the Bellman residual is the maximum distance across all belief states between the value functions of successive iterations. It is possible to find the maximum distance between two piecewise linear and convex functions in polynomial time with an algorithm that uses linear programming (details can be found in [42]).

4.3 Policy Iteration for POMDPs

With value iteration, a POMDP is viewed as a belief-state MDP, and a policy is a mapping from belief states to actions. An early policy iteration algorithm developed by Sondik used this policy representation [65], but it was very complicated and did not meet with success in practice. We shall describe a different approach that has fared better on test problems. With this approach, a policy is represented as a finite-state controller.

4.3.1 Finite-State Controllers

Using a finite-state controller, an agent has a finite number of internal states. Its actions are based only on its internal state, and transitions between internal states

occur when observations are received. Internal states provide agents with a kind of memory, which can be crucial for difficult POMDPs. Of course, an agent's memory is limited by the number of internal states it possesses. In general, an agent cannot remember its entire history of observations, as this would require infinitely many internal states.

We allow for stochasticity in our definition of a finite-state controller. In introducing the POMDP framework, we noted that deterministic policies were sufficient for optimal behavior. However, this does not hold for agents with limited memory. A simple example illustrating this is given in [63]. Intuitively, randomness can help an agent to break out of a costly loops that result from forgetfulness.

We formally define a *controller* as a tuple $\langle Q, \Omega, A, \psi, \eta \rangle$, where

- Q is a finite set of controller nodes.
- Ω is a set of inputs, taken to be the observations of the POMDP.
- A is a set of outputs, taken to be the actions of the POMDP.
- $\psi : Q \to \Delta A$ is an action selection function.
- $\eta : Q \times A \times \Omega \to \Delta Q$ is a transition function.

For each state and starting node of the controller, there is an expected discounted sum of rewards over the infinite horizon. It can be computed using the following system of linear equations, one for each $s \in S$ and $q \in Q$:

$$V(s,q) = \sum_a P(a|q) \left[R(s,a) + \beta \sum_{s',o,q'} P(o,s'|s,a) P(q'|q,a,o) V(s',q') \right].$$

We sometimes refer to the value of the controller at a belief state. For a belief state π, this is defined as

$$V(\pi) = \max_q \sum_s \pi(s) V(s,q).$$

It is assumed that, given an initial state distribution, the controller is started in the joint node which maximizes value from that distribution. Once execution has begun, however, there is no belief state updating. In fact, it is possible for the agent to encounter the same belief state twice and be in a different internal state each time.

4.3.2 Algorithmic Framework

We will describe the policy iteration algorithm in abstract terms, focusing on the key components necessary for convergence. In subsequent sections, we present different possibilities for implementation.

Policy iteration takes as input an arbitrary finite-state controller. The first phase of an iteration consists of evaluating the controller, as described above. Recall that value iteration was initialized with an arbitrary piecewise linear and convex value function, represented by a set of vectors. In policy iteration, the piecewise linear and convex value function arises out of evaluation of the controller. Each controller node has a value when paired with each state. Thus, each node has a corresponding vector and thus a linear value function over belief state space. Choosing the best node for each belief state yields a piecewise linear and convex value function.

The second phase of an iteration is the dynamic programming update. In value iteration, an update produces an improved set of vectors, where each vector corresponds to a deterministic one-step policy. The same set of vectors is produced in this case, but the actions and transition rules for the one-step policy cannot be removed from memory. Each new vector is actually a node that gets added to the controller. All of the probability distributions for the added nodes are deterministic.

Finally, additional operations are performed on the controller. There are many such operations, and we describe two possibilities in the following section. The only restriction placed on these operations is that they do not decrease the value for any belief state. Such an operation is denoted a *value-preserving transformation*.

Input: A finite state controller, and a parameter ϵ.

1. Evaluate the finite-state controller by solving a system of linear equations.
2. Perform a dynamic programming update to add a set of deterministic nodes to the controller.
3. Perform value-preserving transformations on the controller.
4. Calculate the Bellman residual. If it is less than $\epsilon(1-\beta)/2\beta$, then terminate. Otherwise, go to step 1.

Output: An ϵ-optimal finite-state controller.

Table 4.6: Policy Iteration for POMDPs.

The complete algorithm is outlined in Table 4.6. It is guaranteed to converge to a finite-state controller that is ϵ-optimal for all belief states within a finite number of steps. Furthermore, the Bellman residual can be used to obtain a bound on the distance to optimality, as with value iteration.

4.3.3 Controller Reductions

In performing a DP update, potential nodes that are dominated do not get added to the controller. However, after the update is performed, some of the old nodes may have become dominated. These nodes cannot simply be removed, however, as other nodes may transition into them. This is where the dual view of dominance is useful. Recall that if a node is dominated, then there is a convex combination of other nodes with value at least as high from all states. Thus, we can remove the dominated node and *merge* it into the dominating convex combination by changing transition probabilities accordingly. This operation was proposed by Poupart and Boutilier [58] and built upon earlier work by Hansen [28].

Formally, a *controller reduction* attempts to replace a node $q \in Q$ with a distribution $P(\hat{q})$ over nodes $\hat{q} \in Q \setminus q$ such that for all $s \in S$,

$$V(s,q) \leq \sum_{\hat{q} \in Q \setminus q} P(\hat{q}) V(s,\hat{q}).$$

This can be achieved by solving the linear program in Table 4.5. It can be shown that if such a distribution is found and used for merging, the resulting controller is a value-preserving transformation of the original one.

4.3.4 Bounded Dynamic Programming Updates

In the previous section, we described a way to reduce the size of a controller without sacrificing value. The method described in this section attempts to increase the value of the controller while keeping its size fixed. It focuses on one node at a time, and attempts to change the parameters of the node such that the value of the controller is at least as high for all belief states. The idea for this approach originated with Platzman [57], and was made efficient by Poupart and Boutillier [58].

In this method, a node q is chosen, and parameters for the conditional distribution $P(a, q'|q, o)$ are to be determined. Determining these parameters works as follows. We assume that the original controller will be used from the second step on, and try to replace the parameters for q with better ones for just the first step. In other words, we look for parameters which satisfy the following inequality:

$$V(s,q) \leq \sum_{a} P(a|q) \left[R(s,a) + \beta \sum_{s',o,q'} P(q'|q,a,o) P(o,s'|s,a) V(s',q') \right]$$

for all $s \in S$. Note that the inequality is always satisfied by the original parameters. However, it is often possible to get an improvement.

The new parameters can be found by solving a linear program, as shown in Table 4.7. Note that the size of the linear program is polynomial in the sizes of the POMDP and the controller. We call this process a *bounded DP update* because it can be thought of as a DP update with memory constraints. To see this, consider the set of nodes generated by a DP update. These nodes dominate the original nodes across all belief

Variables: $\epsilon, x(a), x(a, o, q')$

Objective: Maximize ϵ

Improvement constraints:

$$\forall s \quad V(s,q) + \epsilon \leq \sum_a \left[x(a)R(s,a) + \beta \sum_{s',o,q'} x(a,o,q')P(o,s'|s,a)V(s',q') \right]$$

Probability constraints:

$$\sum_a x(a) = 1, \qquad \forall a, o \quad \sum_{q'} x(a,o,q') = x(a)$$

$$\forall a \quad x(a) \geq 0, \qquad \forall a, o, q' \quad x(a,o,q') \geq 0$$

Table 4.7: The linear program to be solved for a bounded DP update. The variable $x(a)$ represents $P(a|q)$, and the variable $x(a, o, q')$ represents $P(a, q'|q, o)$.

states, so for every original node, there must be a convex combination of the nodes in this set that dominate the original node for all states. A bounded DP update finds such a convex combination.

It can be shown that a bounded DP update yields a value-preserving transformation. Repeated application of bounded DP updates can lead to a local optimum, at which none of the nodes can be improved any further. Poupart and Boutilier showed that a local optimum has been reached when each node's value function is touching the value function produced by performing a full DP backup. This is illustrated in Figure 4.4.

4.4 Extending Dynamic Programming for POMDPs

Much research has followed from the introduction of the dynamic programming framework described above. One interesting line of work has been extending dynamic programming to work with different problem representations. For some problems, the states or observations are tuples of individual attributes. These attributes may

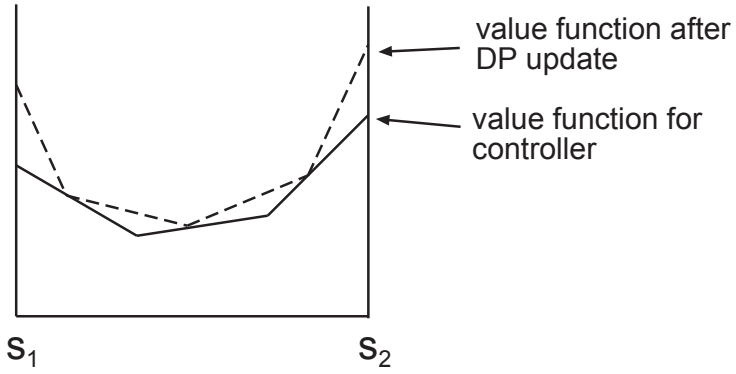


Figure 4.4: A local optimum for bounded DP updates. The solid line is the value function for the controller, and the dotted line is the value function for the controller that results from a full DP update.

interact only loosely, and it can be advantageous to exploit this special structure. Extensions of dynamic programming to deal with this type of problem representation are described in [8, 26].

It can also be useful to consider structured action sets for POMDPs. Planning with hierarchies of high-level actions can be more efficient than just using a "flat" action set. In [27, 68], dynamic programming approaches along these lines are presented.

In addition to the aforementioned work, there has been work on approximation algorithms that use dynamic programming as a foundation [43, 30, 56, 59]. Much of this work takes the view of a POMDP as a belief-state MDP, and attempts to find good ways of abstracting over the belief-state space.

CHAPTER 5

DECENTRALIZED DYNAMIC PROGRAMMING

In the previous chapter, we presented dynamic programming for POMDPs. An important part of POMDP theory is the fact that every POMDP has an equivalent belief-state MDP. No such result is known for DEC-POMDPs, making it difficult to generalize value iteration to the multiagent case. However, we were able to develop an optimal policy iteration algorithm for DEC-POMDPs that includes the POMDP version as a special case. This algorithm is the focus of the chapter.

We first show how to extend the definition of a stochastic controller to the multiagent case. Multiagent controllers include a *correlation device*, which is a source of randomness shared by all the agents. As in the single agent case, policy iteration alternates between exhaustive backups and value-preserving transformations. A convergence proof is given, along with efficient transformations that extend those presented in the previous chapter.

5.1 Correlated Finite-State Controllers

The joint policy for the agents is represented using a stochastic finite-state controller. In this section, we first define a type of controller in which the agents act independently. We then provide an example demonstrating the utility of correlation, and show how to extend the definition of a controller to allow for correlation among agents.

5.1.1 Local Finite-State Controllers

In a local controller, the agent's node is based on the local observations received, and the agent's action is based on the current node. As before, stochastic transitions and action selection are allowed.

We formally define a *local controller* for agent i as a tuple $\langle Q_i, \Omega_i, A_i, \psi_i, \eta_i \rangle$, where

- Q_i is a finite set of controller nodes.
- Ω_i is a set of inputs, taken to be the local observations for agent i.
- A_i is a set of outputs, taken to the actions for agent i.
- $\psi_i : Q_i \rightarrow \Delta A_i$ is an action selection function.
- $\eta_i : Q_i \times A_i \times \Omega_i \rightarrow \Delta Q_i$ is a transition function.

The functions ψ_i and η_i parameterize the conditional distribution $P(a_i, q_i' | q_i, o_i)$. When taken together, the agents' controllers determine the conditional distribution $P(\vec{a}, \vec{q}' | \vec{q}, \vec{o})$. This is denoted an *independent joint controller*. In the following subsection, we show that independence can be limiting.

5.1.2 The Utility of Correlation

The joint controllers described above do not allow the agents to correlate their behavior via a shared source of randomness. We will use a simple example to illustrate the utility of correlation in partially observable domains where agents have limited memory. This example generalizes the one given in [63] to illustrate the utility of stochastic policies in partially observable settings containing a single agent.

Consider the DEC-POMDP shown in Figure 5.1. This problem has two states, two agents, and two actions per agent (A and B). The agents each have only one observation, and thus cannot distinguish between the two states. For this example, we will consider only memoryless policies.

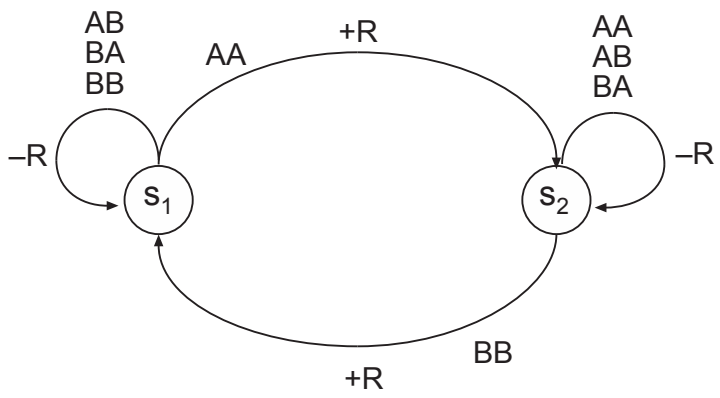


Figure 5.1: This figure shows a DEC-POMDP for which a correlated joint policy yields more reward than the optimal independent joint policy.

Suppose that the agents can independently randomize their behavior using distributions $P(a_1)$ and $P(a_2)$. If the agents each choose either A or B according to a uniform distribution, then they receive an expected reward of $-\frac{R}{2}$ per time step, and thus an expected long-term reward of $\frac{-R}{2(1-\gamma)}$. It is straightforward to show that no independent policy yields higher reward than this one for all states.

Next, let us consider the even larger class of policies in which the agents may act in a correlated fashion. In other words, we consider all joint distributions $P(a_1, a_2)$. Consider the policy that assigns probability $\frac{1}{2}$ to the pair AA and probability $\frac{1}{2}$ to the pair BB. This yields an average reward of 0 at each time step and thus an expected long-term reward of 0. The difference between the rewards obtained by the independent and correlated policies can be made arbitrarily large by increasing R.

5.1.3 Correlated Joint Controllers

In the previous subsection, we established that correlation can be useful in the face of limited memory. In this subsection, we extend our definition of a joint controller to allow for correlation among the agents. To do this, we introduce an additional finite-state machine, called a correlation device, that provides extra signals to the agents at

each time step. The device operates independently of the DEC-POMDP process, and thus does not provide agents with information about the other agents' observations. In fact, the random numbers necessary for its operation could be determined prior to execution time.

Formally, a *correlation device* is a tuple $\langle Q_c, \psi_c \rangle$, where Q_c is a set of nodes and $\psi_c : Q_c \to \Delta Q_c$ is a state transition function. At each step, the device undergoes a transition, and each agent observes its state.

We must modify the definition of a local controller to take the state of the correlation device as input. Now, a local controller for agent i is a conditional distribution of the form $P(a_i, q_i' | q_c, q_i, o_i)$. The correlation device together with the local controllers form a joint conditional distribution $P(\vec{a}, \vec{q}' | \vec{q}, \vec{o})$, where $\vec{q} = \langle q_c, q_1, \ldots, q_n \rangle$. We will refer to this as a *correlated joint controller*. Note that a correlated joint controller with $|Q_c| = 1$ is effectively an independent joint controller. Figure 5.2 contains a graphical representation of the probabilistic dependencies in a correlated joint controller.

The value function for a correlated joint controller can be computed by solving the following system of linear equations, one for each $s \in S$ and $\vec{q} \in \vec{Q}$:

$$V(s, \vec{q}) = \sum_{\vec{a}} P(\vec{a} | \vec{q}) \left[R(s, \vec{a}) + \beta \sum_{s', \vec{o}, \vec{q}'} P(s', \vec{o} | s, \vec{a}) P(\vec{q}' | \vec{q}, \vec{a}, \vec{o}) V(s', \vec{q}') \right].$$

We sometimes refer to the value of the controller for an initial state distribution. For a distribution π, this is defined as

$$V(\pi) = \max_{\vec{q}} \sum_{s} \pi(s) V(s, \vec{q}).$$

It is assumed that, given an initial state distribution, the controller is started in the joint node which maximizes value from that distribution.

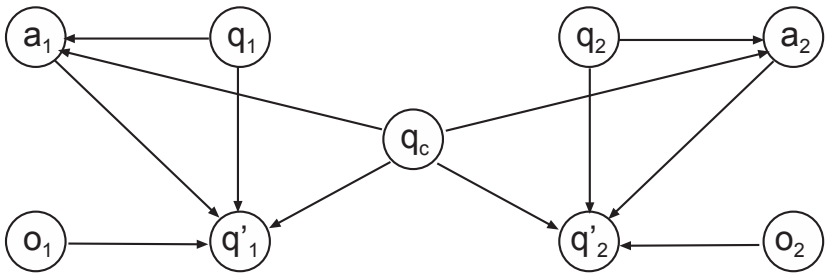


Figure 5.2: A graphical representation of the probabilistic dependencies in a correlated joint controller for two agents.

5.2 Policy Iteration

In this section, we describe the policy iteration algorithm. We first extend the definitions of exhaustive backup and value-preserving transformation to the multiagent case. Following that, we provide a description of the complete algorithm, along with a convergence proof.

5.2.1 Exhaustive Backups

We introduced exhaustive backups in the section on dynamic programming for POMDPs. We stated that one way to implement a DP update was to perform an exhaustive backup, and then prune dominated nodes that were created. More efficient implementations were described thereafter. These implementations involved interleaving pruning with node generation.

For the multiagent case, it is an open problem whether pruning can be interleaved with node generation. Nodes can be removed, as we will show in a later subsection, but for convergence we require exhaustive backups. We do not define DP updates for the multiagent case, and instead make exhaustive backups a central component of our algorithm.

An exhaustive backup adds nodes to the local controllers for all agents at once, and leaves the correlation device unchanged. For each agent i, $|A_i||Q_i|^{|\Omega_i|}$ nodes are

added to the local controller, one for each one-step policy. Thus, the joint controller grows by $|Q_c| \prod_i |A_i||Q_i|^{|O_i|}$ joint nodes.

Note that repeated application of exhaustive backups amounts to a brute force search in the space of deterministic policies. This converges to optimality, but is obviously quite inefficient. As in the single agent case, we must modify the joint controller in between adding new nodes. For convergence, these modifications must preserve value in a sense that will be made formal in the following section.

5.2.2 Value-Preserving Transformations

We now extend the definition of a value-preserving transformation to the multiagent case. In the following subsection, we show how this definition allows for convergence to optimality as the number of iterations grows.

The dual interpretation of dominance is helpful in understanding multiagent value-preserving transformations. Recall that for a POMDP, we say that a node is dominated if there is a convex combination of other nodes with value at least as high for all states. Though we defined a value-preserving transformation in terms of the value function across belief states, we could have equivalently defined it so that every node in the original controller has a dominating convex combination in the new controller.

For the multiagent case, we do not have the concept of a belief state MDP, so we take the second approach mentioned above. In particular, we require that dominating convex combinations exist for nodes of all the local controllers and the correlation device. A transformation of a controller C to a controller D qualifies as a value-preserving transformation if $C \leq D$, where $\leq$ is defined below.

Definition: Consider correlated joint controllers C and D with node sets $\vec{Q}$ and $\vec{R}$, respectively. We say that $C \leq D$ if there exist mappings $f_i : Q_i \rightarrow \Delta R_i$ for each agent i and $f_c : Q_c \rightarrow \Delta R_c$ such that

$$V(s, \vec{q}) \leq \sum_{\vec{r}} P(\vec{r}|\vec{q}) V(s, \vec{r})$$

for all $s \in S$ and $\vec{q} \in \vec{Q}$.

We sometimes describe the f_i and f_c as a single mapping $f : \vec{Q} \rightarrow \Delta\vec{R}$. Examples of efficient value-preserving transformations are given in a later section. In the following subsection, we show that alternating between exhaustive backups and value-preserving transformations yields convergence to optimality.

5.2.3 Algorithmic Framework

The policy iteration algorithm is initialized with an arbitrary correlated joint controller. In the first part of an iteration, the controller is evaluated via the solution of a system of linear equations. Next, an exhaustive backup is performed to add nodes to the local controllers. Finally, value-preserving transformations are performed.

In contrast to the single agent case, there is no Bellman residual for testing convergence to ϵ-optimality. We resort to a simpler test for ϵ-optimality based on the discount rate and the number of iterations so far. Let $|R_{\max}|$ be the largest absolute value of an immediate reward possible in the DEC-POMDP. Our algorithm terminates after iteration t if $\frac{\beta^{t+1}|R_{\max}|}{1-\beta} \leq \epsilon$. At this point, due to discounting, optimality over t steps is close enough to optimality over the infinite horizon. Justification for this test is provided in the convergence proof. The complete algorithm is sketched in Table 5.1.

Before proving convergence, we state and prove a key lemma regarding the ordering of exhaustive backups and value-preserving transformations. For ease of exposition, we prove the lemma under the assumption that there is no correlation device. Including a correlation device is straightforward but tedious.

Lemma 8 *Let C and D be correlated joint controllers, and let $\hat{C}$ and $\hat{D}$ be the results of performing exhaustive backups on C and D, respectively. Then $\hat{C} \leq \hat{D}$ if $C \leq D$.*

Input: A correlated joint controller, and a parameter ϵ.

1. Evaluate the correlated joint controller by solving a system of linear equations.
2. Perform an exhaustive backup to add deterministic nodes to the local controllers.
3. Perform value-preserving transformations on the controller.
4. If $\frac{\beta^{t+1}|R_{\max}|}{1-\beta} \leq \epsilon$, where t is the number of iterations so far, then terminate. Else go to step 1.

Output: A correlated joint controller that is ϵ-optimal for all states.

Table 5.1: Policy Iteration for DEC-POMDPs.

Proof. Suppose we are given controllers C and D, where $C \leq D$. Call the sets of joint nodes for these controllers $\vec{Q}$ and $\vec{R}$, respectively. It follows that there exists a function $f_i : Q_i \to \Delta R_i$ for each agent i such that

$$V(s, \vec{q}) \leq \sum_{\vec{r}} P(\vec{r}|\vec{q}) V(s, \vec{r})$$

for all $s \in S$ and $\vec{q} \in \vec{Q}$.

We now define functions $\hat{f}_i$ to map between the two controllers $\hat{C}$ and $\hat{D}$. For the old nodes, we define $\hat{f}_i$ to produce the same output as f_i. It remains to specify the results of $\hat{f}_i$ applied to the nodes added by the exhaustive backup. New nodes of $\hat{C}$ will be mapped to distributions involving only new nodes of $\hat{D}$.

To describe the mapping formally, we need to introduce some new notation. Recall that the new nodes are all deterministic. For each new node $\vec{r}$ in controller $\hat{D}$, the node's action is denoted $\vec{a}(\vec{r})$, and its transition rule is denoted $\vec{r}'(\vec{r}, \vec{o})$. Now, the mappings $\hat{f}_i$ are defined such that

$$P(\vec{r}|\vec{q}) = P(\vec{a}(\vec{r})|\vec{q}) \prod_{\vec{o}} \sum_{\vec{q}'} P(\vec{q}'|\vec{q}, \vec{a}(\vec{r}), \vec{o}) P(\vec{r}'(\vec{r}, \vec{o})|\vec{q}')$$

for all $\vec{q}$ in controller $\hat{C}$ and $\vec{r}$ in controller $\hat{D}$.

We must now show that the mapping $\hat{f}$ satisfies the inequality given in the definition of a value-preserving transformation. For the nodes that were not added by the exhaustive backup, this is straightforward. For the new nodes $\vec{q}$ of the controller $\hat{C}$, we have for all $s \in S$,

$$\begin{aligned}
V(s,\vec{q}) &= \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,\vec{a}) + \sum_{\vec{o},s',\vec{q}'} P(s',\vec{o}|s,\vec{a}) P(\vec{q}'|\vec{q},\vec{a},\vec{o}) V(s',\vec{q}') \right] \\
&\leq \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,\vec{a}) + \sum_{\vec{o},s',\vec{q}'} P(s',\vec{o}|s,\vec{a}) P(\vec{q}'|\vec{q},\vec{a},\vec{o}) \sum_{\vec{r}'} P(\vec{r}'|\vec{q}') V(s',\vec{r}') \right] \\
&= \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,\vec{a}) + \sum_{\vec{o},s',\vec{q}',\vec{r}'} P(s',\vec{o}|s,\vec{a}) P(\vec{q}'|\vec{q},\vec{a},\vec{o}) P(\vec{r}'|\vec{q}') V(s',\vec{r}') \right] \\
&= \sum_{\vec{r}} P(\vec{r}|\vec{q}) \left[R(s,\vec{a}(\vec{r})) + \sum_{\vec{o},s'} P(s',\vec{o}|s,\vec{a}(\vec{r})) V(s',\vec{r}'(\vec{r},\vec{o})) \right] \\
&= \sum_{\vec{r}} P(\vec{r}|\vec{q}) V(s,\vec{r}).
\end{aligned}$$

□

We can now state and prove the main convergence theorem for policy iteration.

Theorem 6 *For any ϵ, policy iteration returns a correlated joint controller that is ϵ-optimal for all initial states in a finite number of iterations.*

Proof. Repeated application of exhaustive backups amounts to a brute force search in the space of deterministic joint policies. Thus, after t exhaustive backups, the resulting controller is optimal for t steps from any initial state. Let t be any number large enough that $\frac{\beta^{t+1}|R_{\max}|}{1-\beta} \leq \epsilon$. Then any possible discounted sum of rewards after t time steps is small enough that optimality over t time steps implies ϵ-optimality over the infinite horizon.

Now recall the previous lemma, which states that exhaustive backups and value-preserving transformations commute. By an inductive argument, performing t steps of policy iteration is a value-preserving transformation of the result of t exhaustive

backups. We have argued that for large enough t, the value of the controller resulting from t exhaustive backups is within ϵ of optimal for all states. Thus, the result of t steps of policy iteration is also within ϵ of optimal for all states. $\square$

5.3 Efficient Value-Preserving Transformations

In this section, we describe how to extend controller reductions and bounded DP updates to the multiagent case. We will show that both of these operations are value-preserving transformations.

5.3.1 Controller Reductions

Recall that in the single agent case, a node can be removed if for all belief states, there is another node with value at least as high. The equivalent dual interpretation is that a node can be removed is there exists a convex combination of other nodes with value at least as high across the entire state space.

Using the dual interpretation, we can extend this to a rule for removing nodes in the multiagent case. The rule applies to removing nodes either from a local controller or from the correlation device. Intuitively, in considering the removal of a node from a local controller or the correlation device, we consider the nodes of the other controllers to be part of the hidden state.

More precisely, suppose we are considering removing node q_i from agent i's local controller. To do this, we need to find a distribution $P(\hat{q}_i)$ over nodes $\hat{q}_i \in Q_i \setminus q_i$ such that for all $s \in S$, $q_{-i} \in Q_{-i}$, and $q_c \in Q_c$,

$$V(s, q_i, q_{-i}, q_c) \leq \sum_{\hat{q}_i} P(\hat{q}_i) V(s, \hat{q}_i, q_{-i}, q_c).$$

Finding such a distribution can be formulated as a linear program, as shown in Table 5.2a. In this case, success is finding parameters such that $\epsilon \geq 0$. The linear program

is polynomial in the sizes of the DEC-POMDP and controllers, but exponential in the number of agents.

If we are successful in finding parameters that make $\epsilon \geq 0$, then we can merge the dominated node into the convex combination of other nodes by changing controller parameters accordingly. At this point, there is no chance of ever transitioning into q_i, and thus it can be removed.

The rule for the correlation device is very similar. Suppose that we are considering the removal of node q_c. In this case, we need to find a distribution $P(\hat{q}_c)$ over nodes $\hat{q}_c \in Q_c \setminus q_c$ such that for all $s \in S$ and $\vec{q} \in \vec{Q}$,

$$V(s, \vec{q}, q_c) \leq \sum_{\hat{q}_c} P(\hat{q}_c) V(s, \vec{q}, \hat{q}_c).$$

Note that we abuse notation here and use $\vec{Q}$ for the set of tuples of local controller nodes, excluding the nodes for the correlation device. As in the previous case, finding parameters can done using linear programming. This is shown in Table 5.2b. This linear program is also polynomial in the the sizes of the DEC-POMDP and controllers, but exponential in the number of agents.

We have the following theorem, which states that controller reductions are value-preserving transformations.

Theorem 7 *Any controller reduction applied to either a local node or a node of the correlation device is a value-preserving transformation.*

Proof. Suppose that we have replaced an agent i node q_i with a distribution over nodes in $Q_i \setminus q_i$. Let us take f_i to be the identity map for all nodes except q_i, which will map to the new distribution. We take f_c to be the identity map, and we take f_j to be the identity map for all $j \neq i$. This yields a complete mapping f. We must now show that f satisfies the condition given in the definition of a value-preserving transformation.

(a) Variables: $\epsilon, x(\hat{q}_i)$

Objective: Maximize ϵ

Improvement constraints:

$$\forall s, q_{-i}, q_c \quad V(s, q_i, q_{-i}, q_c) + \epsilon \leq \sum_{\hat{q}_i} x(\hat{q}_i) V(s, \hat{q}_i, q_{-i}, q_c)$$

Probability constraints:

$$\sum_{\hat{q}_i} x(\hat{q}_i) = 1, \qquad \forall \hat{q}_i \quad x(\hat{q}_i) \geq 0$$

(b) Variables: $\epsilon, x(q_c)$

Objective: Maximize ϵ

Improvement constraints:

$$\forall s, \vec{q} \quad V(s, \vec{q}, q_c) + \epsilon \leq \sum_{\hat{q}_c} x(\hat{q}_c) V(s, \vec{q}, \hat{q}_c)$$

Probability constraints:

$$\sum_{\hat{q}_c} x(\hat{q}_c) = 1, \qquad \forall \hat{q}_c \quad x(\hat{q}_c) \geq 0$$

Table 5.2: (a) The linear program to be solved to find a replacement for agent i's node q_i. The variable $x(\hat{q}_i)$ represents $P(\hat{q}_i)$. (b) The linear program to be solved to find a replacement for the correlation node q_c. The variable $x(\hat{q}_c)$ represents $P(\hat{q}_c)$.

Let V_o be the value function for the original controller, and let V_n be the value function for the controller with q_i removed. A controller reduction requires that

$$V_o(s, \vec{q}) \leq \sum_{\vec{r}} P(\vec{r}|\vec{q}) V_o(s, \vec{r})$$

for all $s \in S$ and $\vec{q} \in \vec{Q}$. Thus, we have

$$V_o(s, \vec{q}) \quad = \quad \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s, a) + \beta \sum_{s', \vec{o}, \vec{q}'} P(\vec{q}'|\vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) V_o(s', \vec{q}') \right]$$

$$\leq \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,a) + \beta \sum_{s',\vec{o},\vec{q}'} P(\vec{q}'|\vec{q},\vec{a},\vec{o}) P(s',\vec{o}|s,\vec{a}) \sum_{\vec{r}'} P(\vec{r}|\vec{q}) V_o(s,\vec{r}') \right]$$

$$= \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,a) + \beta \sum_{s',\vec{o},\vec{q}',\vec{r}'} P(\vec{q}'|\vec{q},\vec{a},\vec{o}) P(s',\vec{o}|s,\vec{a}) P(\vec{r}|\vec{q}) V_o(s,\vec{r}') \right]$$

for all $s \in S$ and $\vec{q} \in \vec{Q}$. Notice that the formula on the right is the Bellman operator for the new controller, applied to the old value function. Denoting this operator T_n, the system of inequalities implies that $T_n V_o \geq V_o$. By monotonicity, we have that for all $k \geq 0$, $T_n^{k+1}(V_o) \geq T_n^k(V_o)$. Since $V_n = \lim_{k\to\infty} T_n^k(V_o)$, we have that $V_n \geq V_o$. This is sufficient for f to satisfy the condition in the definition of value-preserving transformation.

The argument for removing a node of the correlation device is almost identical to the one given above. □

5.3.2 Bounded Dynamic Programming Updates

In the previous section, we described a way to reduce the size of a controller without sacrificing value. Recall that in the single agent case, we could also use bounded DP updates to increase the value of the controller while keeping its size fixed. This technique can be extended to the multiagent case. As in the previous section, the extension relies on improving a single local controller or the correlation device, while viewing the nodes of the other controllers as part of the hidden state.

We first describe in detail how to improve a local controller. To do this, we choose an agent i, along with a node q_i. Then, for each $o_i \in \Omega_i$, we search for new parameters for the conditional distribution $P(a_i, q_i'|q_i, o_i)$.

The search for new parameters works as follows. We assume that the original controller will be used from the second step on, and try to replace the parameters for q_i with better ones for just the first step. In other words, we look for parameters satisfying the following inequality:

$$V(s, \vec{q}) \leq \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,a) + \beta \sum_{s', \vec{o}, \vec{q}'} P(\vec{q}'|\vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) V(s', \vec{q}') \right]$$

for all $s \in S$, $q_{-i} \in Q_{-i}$, and $q_c \in Q_c$. The search for new parameters can be formulated as a linear program, as shown in Table 5.3a. Its size is polynomial in the sizes of the DEC-POMDP and the joint controller, but exponential in the number of agents.

The procedure for improving the correlation device is very similar to the procedure for improving a local controller. We first choose a device node q_c, and consider changing its parameters for just the first step. We look for parameters satisfying the following inequality:

$$V(s, \vec{q}) \leq \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s,a) + \beta \sum_{s', \vec{o}, \vec{q}'} P(\vec{q}'|\vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) V(s', \vec{q}') \right]$$

for all $s \in S$ and $\vec{q} \in \vec{Q}$.

As in the previous case, the search for parameters can be formulated as a linear program. This is shown in Table 5.3b. This linear program is also polynomial in the sizes of the DEC-POMDP and joint controller, but exponential in the number of agents.

The following theorem states that bounded DP updates preserve value.

Theorem 8 *Performing a bounded DP update on a local controller or the correlation device produces a new correlated joint controller which is a value-preserving transformation of the original.*

Proof. Consider the case in which some node q_i of agent i's local controller is changed. We define f to be a deterministic mapping from nodes in the original controller to the corresponding nodes in the new controller.

(a) Variables: $\epsilon, x(q_c, a_i), x(q_c, a_i, o_i, q_i')$

Objective: Maximize ϵ

Improvement constraints:

$$\forall s, q_{-i}, q_c \quad V(s, \vec{q}, q_c) + \epsilon \leq \sum_{\vec{a}} P(a_{-i}|q_c, q_{-i})[x(q_c, a_i)R(s, \vec{a}) + \beta \sum_{s', \vec{o}, \vec{q}', q_c'} x(c, a_i, o_i, q_i')P(q_{-i}'|q_c, q_{-i}, a_{-i}, o_{-i}) \cdot P(\vec{o}, s'|s, \vec{a})P(q_c'|q_c)V(s', \vec{q}', q_c')]$$

Probability constraints:

$$\forall q_c \quad \sum_{a_i} x(q_c, a_i) = 1, \qquad \forall q_c, a_i, o_i \quad \sum_{q_i'} x(q_c, a_i, o_i, q_i') = x(q_c, a_i)$$

$$\forall q_c, a_i \quad x(q_c, a_i) \geq 0, \qquad \forall q_c, a_i, o_i, q_i' \quad x(q_c, a_i, o_i, q_i') \geq 0$$

(b) Variables: $\epsilon, x(q_c')$

Objective: Maximize ϵ

Improvement constraints:

$$\forall s, \vec{q} \quad V(s, \vec{q}, q_c) + \epsilon \leq \sum_{\vec{a}} P(\vec{a}|q_c, \vec{q})[R(s, \vec{a}) + \beta \sum_{s', \vec{o}, \vec{q}', q_c'} P(\vec{q}'|q_c, \vec{q}, \vec{a}, \vec{o}) \cdot P(s', \vec{o}|s, \vec{a})x(q_c')V(s', \vec{q}', q_c')]$$

Probability constraints:

$$\forall q_c' \quad \sum_{q_c'} x(q_c') = 1, \qquad \forall q_c' \quad x(q_c') \geq 0$$

Table 5.3: (a) The linear program used to find new parameters for agent i's node q_i. The variable $x(q_c, a_i)$ represents $P(a_i|q_i, q_c)$, and the variable $x(q_c, a_i, o_i, q_i')$ represents $P(a_i, q_i'|q_c, q_i, o_i)$. (b) The linear program used to find new parameters for the correlation device node q_c. The variable $x(q_c')$ represents $P(q_c'|q_c)$.

Let V_o be the value function for the original controller, and let V_n be the value function for the new controller. Recall that the new parameters for $P(a_i, q_i'|q_c, q_i, o_i)$ must satisfy the following inequality for all $s \in S$, $q_{-i} \in Q_{-i}$, and $q_c \in Q_c$:

$$V_o(s, \vec{q}) \leq \sum_{\vec{a}} P(\vec{a}|\vec{q}) \left[R(s, a) + \beta \sum_{s', \vec{o}, \vec{q}'} P(\vec{q}'|\vec{q}, \vec{a}, \vec{o}) P(s', \vec{o}|s, \vec{a}) V_o(s', \vec{q}') \right].$$

Notice that the formula on the right is the Bellman operator for the new controller, applied to the old value function. Denoting this operator T_n, the system of inequalities implies that $T_n V_o \geq V_o$. By monotonicity, we have that for all $k \geq 0$, $T_n^{k+1}(V_o) \geq T_n^k(V_o)$. Since $V_n = \lim_{k \to \infty} T_n^k(V_o)$, we have that $V_n \geq V_o$. Thus, the new controller is a value-preserving transformation of the original one.

The argument for changing nodes of the correlation device is almost identical to the one given above. □

5.4 Open Issues

We noted at the beginning of the chapter that there is no known way to convert a DEC-POMDP into an equivalent belief-state MDP. Despite this fact, we were able to develop a provably convergent policy iteration algorithm. However, the policy iteration algorithm for POMDPs has other desirable properties besides convergence, and we have not yet been able to extend these to the multiagent case. Two such properties are described below.

5.4.1 Error Bounds

The first property is the existence of a Bellman residual. In the single agent case, it is possible to compute a bound on the distance to optimality using two successive value functions. In the multiagent case, policy iteration produces a sequence of controllers, each of which has a value function. However, we do not have a way to obtain an error

bound from these value functions. For now, to bound the distance to optimality, we must consider the discount rate and the number of iterations completed.

5.4.2 Avoiding Exhaustive Backups

In performing a DP update for POMDPs, it is possible to remove certain nodes from consideration without first generating them. In the previous chapter, we gave a high-level description of a few different approaches to doing this. For DEC-POMDPs, however, we did not define a DP update and instead used exhaustive backups as the way to expand a controller. Since exhaustive backups are expensive, it would be useful to extend the more sophisticated pruning methods for POMDPs to the multiagent case.

Unfortunately, in the case of POMDPs, the proofs of correctness for these methods all use the fact that there exists a Bellman equation. Roughly speaking, this equation allows us to determine whether a potential node is dominated by just analyzing the nodes that would be its successors. Because we do not currently have an analog of the Bellman equation for DEC-POMDPs, we have not been able to generalize these results.

There is one exception to the above statement, however. When an exhaustive backup has been performed for all agents except one, then a type of belief state space can be constructed for the agent in question using the system states and the nodes for the other agents. The POMDP node generation methods can then be applied to just that agent. In general, though, it seems difficult to rule out a node for one agent before generating all the nodes for the other agents.

5.5 Discussion

In this chapter, we presented a policy iteration algorithm for DEC-POMDPs. To represent policies, the algorithm uses correlated joint controllers, which consist of

a local controller for each agent, along with a correlation device. Using a simple example, we showed that correlation devices are essential for obtaining the maximum reward possible given memory constraints.

We generalized the notion of a value-preserving transformation from the single agent case, and showed that alternating between exhaustive backups and value-preserving transformations leads to optimality. We further provided two provably efficient transformations that generalize those given for POMDPs. The transformations allow for the modification of both local controllers and the correlation device.

Having drawn a connection to powerful ideas from the POMDP literature, we expect to be able to solve more difficult DEC-POMDPs than could be solved before. Optimality may still be difficult to achieve for many problems. However, the tools we have developed provide efficient ways of improving policies to generate good suboptimal solutions. Our bridge to the literature on dynamic programming for POMDPs may allow for the import of other tools, such as dynamic programming algorithms for compact problem representations. The concluding chapter contains a discussion of this topic.

This algorithmic framework leaves many design decisions open. First, there is the matter of how to choose an initial correlated joint controller. The algorithm is guaranteed to converge regardless of the initial controller, but some will allow for faster convergence than others. Second is the question of which value-preserving transformation to apply and exactly how to apply it. By removing nodes or applying bounded DP updates in different orders, different results are possible. Finally, there are more possibilities for value-preserving transformations beyond the ones that we have mentioned.

Given this huge space of possibilities, we will not attempt to find a "best" implementation. Instead, we present in the following chapter some experimental results

for a few different approaches. We expect these to provide practical guidance to supplement the theoretical results given.

CHAPTER 6

DYNAMIC PROGRAMMING EXPERIMENTS

This chapter describes the results of experiments performed using policy iteration. Because of the flexibility of the algorithm, it is impossible to explore all possible ways of implementing it. However, we did experiment with a few different implementation strategies to gain an idea of how the algorithm works in practice. Two main sets of experiments were performed on a single set of test problems.

Our first set of experiments focused on exhaustive backups and controller reductions. The results confirm that value improvement can be obtained through iterated application of these two operations. However, because exhaustive backups are expensive, the algorithm was unable to complete more than a few iterations on any of our test problems.

In the second set of experiments, we addressed the complexity issues by using only bounded DP updates, and no exhaustive backups. With bounded DP updates, we were able to obtain higher-valued controllers while keeping memory requirements fixed. We examined how the sizes of the initial local controllers and correlation device affected the value of the final solution.

6.1 Test Domains

In this section, we describe three test domains, ordered by the size of the problem representation. For each problem, the transition function, observation function, and reward functions are described. In addition, an initial state is specified. Although policy iteration does not require an initial state as input, we decided to focus on a

single state to make value comparisons easier. A few different initial states were tried for each problem, and qualitatively similar results were obtained.

6.1.1 Recycling Robot Problem

Our first problem domain is an extension of the recycling robot problem [66] to the multiagent case. In the original problem, a robot has the task of picking up cans in an office building. It has sensors to find a can and motors to move around the office in order to look for cans. The robot is able to control a gripper arm to grasp each can and then place it in an on-board receptical. A battery is used with power levels of high and low, and the robot is capable of the following high level actions: (1) actively search for a can, (2) wait for a can to be brought, or (3) recharge the battery. Searching for a can causes the robot to lose battery power with a certain probability, but also results in a reward (unless the battery is depleted). Waiting does not change the battery level and results in a reward, while completely draining the battery results in a penalty. If the robot exhausts the battery, it is picked up and plugged into the charger and continues to act.

In the multiagent case we remove the wait action and introduce a larger can that can only be retrieved if both robots pick it up at the same time. Each agent can now decide to independently search for a small can or to attempt to cooperate in order to receive a larger reward. If only one agent chooses to retreive the large can, no reward is given. The robots have the same battery states of high and low, with an increased likelihood of transitioning to a lower state after attempting to pick up the large can. Each robot's battery power depends only on its own actions and each can fully observe its own battery level, but not that of the other agent.

The probability of transitioning from the high battery state to the low one for each agent is 0.5 after the attempting to pick up the large can and 0.3 after searching for a small can. The probability of each robot depleting the battery while on a low

charge is 0.3 for the big can and 0.2 for the small can. The reward received for both agents picking up the big can is 5. There is a reward of 2 for each agent that picks up the small can and a -10 reward for each agent that depletes the battery.

The problem has 2 agents, 4 states, 2 observations per agent, and 3 actions per agent. The discount factor is set to 0.9. The start state is such that the battery levels for both robots are high.

6.1.2 Multiple Access Broadcast Channel

Our next domain is an idealized model of control of a multi-access broadcast channel [48]. In this problem, nodes need to broadcast messages to each other over a channel. Only one node may broadcast at a time, otherwise a collision occurs. The nodes share the common goal of maximizing the throughput of the channel.

At the start of each time step, each node decides whether or not to send a message. The nodes receive a reward of 1 when a message is successfully broadcast and a reward of 0 otherwise. At the end of the time step, each node observes its own buffer, and whether the previous step contained a collision, a successful broadcast, or nothing attempted.

The message buffer for each agent has space for only one message. If a node is unable to broadcast a message, the message remains in the buffer for the next time step. If a node i is able to send its message, the probability that its buffer will fill up on the next step is p_i. Our problem has two nodes, with $p_1 = 0.9$ and $p_2 = 0.1$. There are 4 states, 2 actions per agent, and 6 observations per agent. The discount factor is 0.9. In the start state, node 1 has a message in its buffer, and node 2 does not.

6.1.3 Meeting on a Grid

In this problem, we have two robots navigating on a two-by-two grid. Each robot can only sense whether there are walls to its left or right, and their goal is to spend

as much time as possible on the same square. The actions are to move up, down, left, or right, or to stay on the same square. When a robot attempts to move to an open square, it only goes in the intended direction with probability 0.6, otherwise it either goes in another direction or stays in the same square. Any move into a wall results in staying in the same square. The robots do not interfere with each other and cannot sense each other.

This problem has 16 states, since each robot can be in any of 4 squares at any time. Each robot has 4 observations, since it has a bit for sensing a wall to its left or right. The total number of actions for each agent is 5. The reward is 1 when the agents share a square, and 0 otherwise, and the discount factor is 0.9. The initial state places both robots on the same grid cell.

6.2 Exhaustive Backups and Controller Reductions

In this section, we present the results of using exhaustive backups together with controller reductions.

6.2.1 Experimental Setup

For each domain, our initial controllers for each agent contained a single node with a self loop, and there was no correlation device. For the recycling robot problem, the repeated action was to recharge the battery; for the broadcast channel problem, it was to not send a message; and for the grid problem it was to move up. The reason for starting with the smallest possible controllers was to see how many iterations we could complete before running out of memory.

On each iteration, we performed an exhaustive backup, and then alternated between agents, performing controller reductions until no more nodes could be removed. For each iteration, we recorded the sizes of the controllers produced, and noted what

Recycling Robot			
Iteration	Controller Sizes (no red.)	Controller Sizes (red.)	Value from s_0
0	$(1, 1)$	$(1, 1)$	0
1	$(3, 3)$	$(3, 3)$	5.0
2	$(27, 27)$	$(6, 6)$	24.4
3	$(2187, 2187)$	$(24, 24)$	25.6

Multi-Acces Broadcast Channel			
Iteration	Controller Sizes (no red.)	Controller Sizes (red.)	Value from s_0
0	$(1, 1)$	$(1, 1)$	0.0
1	$(2, 2)$	$(2, 2)$	0.0
2	$(128, 128)$	$(10, 12)$	0.81

Meeting on a Grid			
Iteration	Controller Sizes (no red.)	Controller Sizes (red.)	Value from s_0
0	$(1, 1)$	$(1, 1)$	2.8
1	$(5, 5)$	$(5, 5)$	3.4
2	$(3125, 3125)$	$(80, 80)$	3.7

Table 6.1: Results of applying exhaustive backups and controller reductions to our test problems. The second column contains the sizes of the controllers if only exhaustive backups had been performed. The third column contains the sizes of the controllers with controller reductions being performed on each iteration.

the sizes would be had no controller reductions been performed. In addition, we recorded the value from the initial state.

6.2.2 Results

The results are shown in Table 6.1. Because exhaustive backups add many nodes, we were unable to complete many iterations without exceeding memory limits. As expected, the smallest problem led to the largest number of iterations being completed. Although we could not complete many iterations before running out of memory, the use of controller reductions led to a significant improvement over the approach of just applying exhaustive backups.

6.3 Bounded Dynamic Programming Updates

As we saw from the previous experiments, exhaustive backups can fill up memory very quickly. This leads naturally to the question of how much improvement is possible without exhaustive backups. In this section, we describe an experiment in which we repeatedly applied bounded DP updates, which left the size of the controller fixed. We experimented with different starting sizes for the local controllers and the correlation device.

6.3.1 Experimental Setup

We will define a *trial run* of the algorithm as follows. At the start of a trial run, a size is chosen for each of the local controllers and the correlation device. The action selection and transition functions are initialized to be deterministic, with the outcomes drawn according to a uniform distribution. A *step* consists of choosing a node uniformly at random from the correlation device or one of the local controllers, and performing a bounded DP update on that node. After 50 steps, the run is considered over. In practice, we found that values usually stabilized within 15 steps.

We varied the sizes of the local controllers from 1 to 7 (the agents' controllers were always the same sizes as each other), and we varied the size of the correlation device from 1 to 2. Thus, the number of joint nodes ranged from 1 to 98. It was not possible to use larger controllers without running out of memory. For each combination of sizes, we performed 20 trial runs. We recorded the highest value obtained across all runs, as well as the average value over all runs.

6.3.2 Results

For each of the three problems, we were able to obtain solutions with higher value than with exhaustive backups. Thus, we see that even though repeated application of bounded DP updates does not have an optimality guarantee, it can be competitive with an algorithm which does. However, it should be noted that we have not

performed an exhaustive comparison. We could have made different design decisions for both approaches concerning the starting controllers, the order in which nodes are considered, and other factors.

Besides comparing to the exhaustive backup approach, we wanted to examine the effect of the sizes of the local controllers and correlation device on value. For each combination of controller sizes, we looked at the best solutions found across all trial runs. The values for these solutions were the same for all controller sizes except for the few smallest.

It was more instructive to compare average values over all trial runs. Figure 6.1 shows a graph of average values plotted against controller size. We found that, for the most part, the average value increases when we increase the size of the correlation device from one node to two nodes (essentially moving from independent to correlated).

For small controllers, the average value tends to increase with controller size. However, for larger controllers, there is no clear trend. This can be explained by considering how a bounded DP update works. For new node parameters to be acceptable, they must not decrease the value for any combination of states, nodes for the other controllers, and nodes for the correlation device. This becomes more difficult as the numbers of nodes increase, and thus it is easier to get stuck in a local optimum.

Improving multiple controllers at once could help to alleviate the aforementioned problem. We do not currently have a way to do this using linear programming, and it thus remains an interesting topic for future work.

6.4 Discussion

We have demonstrated how policy iteration can be used to improve a correlated joint controller. We showed that using controller reductions together with exhaustive backups is more efficient in terms of memory than using exhaustive backups alone.

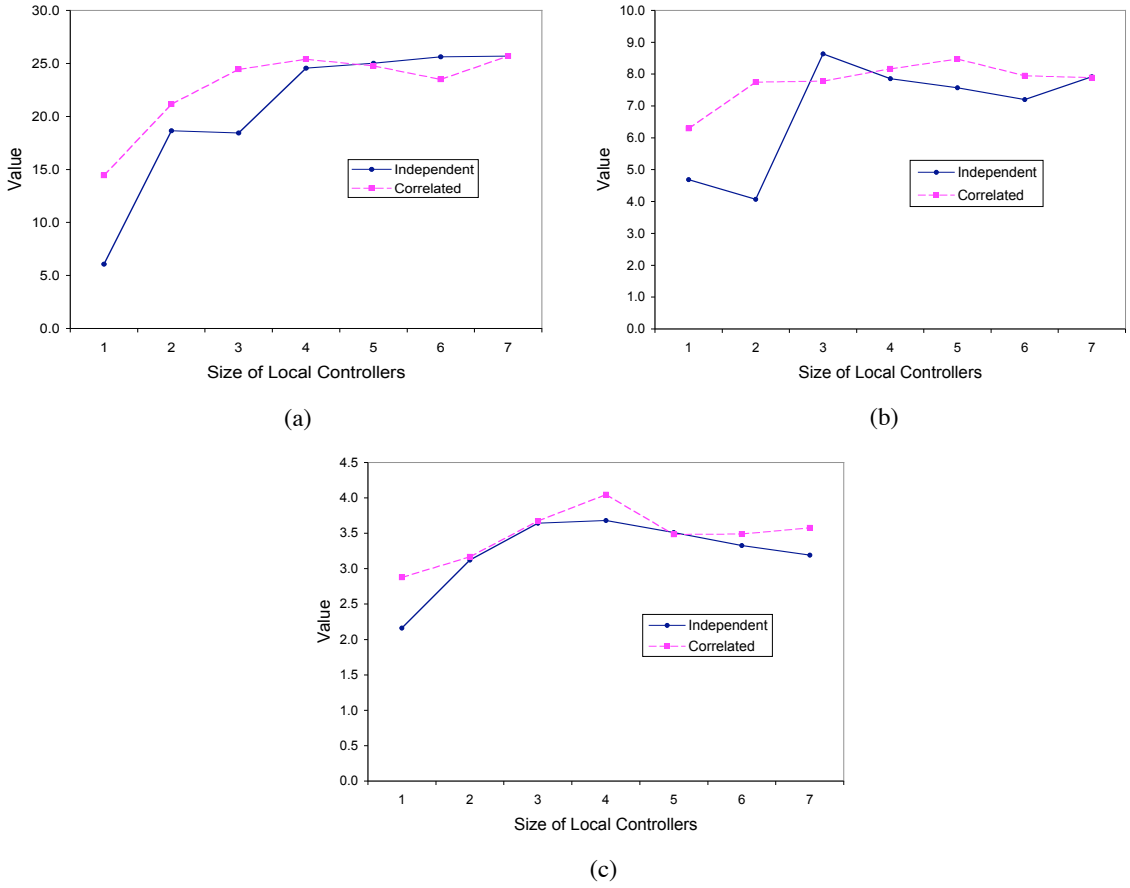


Figure 6.1: Average value per trial run plotted against the size of the local controllers, for (a) the recycling robot problem, (b) the multi-access broadcast channel problem, and (c) the robot navigation problem. The solid line represents independent controllers (a correlation device with one node), and the dotted line represents a joint controller including a two-node correlation device.

However, due to the complexity of exhaustive backups, even that approach could only complete a few iterations on each of our test problems.

Using bounded DP updates alone provided a good way to deal with the complexity issues. With bounded DP updates, we were able to find higher-valued policies than with the previous approach. Through our experiments, we were able to understand how the sizes of the local controllers and correlation device affect the final values obtained.

For significantly larger problems or controllers, or problems with more than two agents, we run into space complexity issues. One potential solution to this problem is to mix controller reductions in with bounded DP updates in an attempt to both shrink *and* improve controllers. Other ideas for scaling up to larger problems are presented in the concluding chapter.

6.5 Computational Environment

The experimental results reported in this chapter were obtained on a Dell GX260 with a 2.2GHz processor and 1GB of RAM. Code was written in Java and executed under a LINUX operating system. For the linear programming subroutines, the commercial package CPLEX was used.

CHAPTER 7

CONCLUSION

7.1 Summary of Contributions

In this dissertation, we took a formal approach to automated planning for distributed agents. By taking such an approach, we were able to formulate and answer precise questions about the problem. We adopted the DEC-POMDP framework, which captures the essential aspects of distributed decision making, and the questions that we answered pertained to its computational aspects.

Specifically, our first result regards the worst-case complexity of finding optimal decision-making policies. We showed that the finite-horizon DEC-POMDP problem is NEXP-complete, even when there are only two agents whose observations jointly determine the system state. The finite-horizon POMDP problem is known to be only PSPACE-complete. Since P $\neq$ NEXP, the DEC-POMDP problem provably does not admit a polynomial-time algorithm. This is in contrast to the POMDP problem, as it is not known whether P = PSPACE. Furthermore, assuming that EXP $\neq$ NEXP, the DEC-POMDP problem requires super-exponential time to solve in the worst case. Thus, our complexity result illustrates a fundamental difference between planning for one agent and planning for multiple agents.

Our second result is a policy iteration algorithm for DEC-POMDPs. The algorithm uses a novel policy representation consisting of stochastic finite-state controllers for each agent along with a correlation device. We defined value-preserving transformations and showed that alternating between exhaustive backups and value-preserving transformations leads to convergence to optimality. Finally, we extended

controller reductions and bounded DP updates from the single agent case to the multiagent case. Both of these operations are value-preserving transformations and are provably efficient. Our algorithm serves as the first nontrivial exact algorithm for DEC-POMDPs, and provides a bridge to the large body of work on dynamic programming for POMDPs.

Together, our results shed new light on the problem of distributed decision making. It has been known for years that the general problem of planning for distributed agents is a challenging one. In light of our complexity results, it is now possible to make more precise statements about the problem. In addition, our algorithmic results provide powerful tools for solving problems of this nature. We expect that our policy iteration algorithm will provide a useful foundation for addressing realistic-sized problems in the future.

7.2 Future Work

Our work provides a solid foundation for solving distributed decision making problems, but much work remains in addressing more challenging domains. Below we describe some key challenges, along with some preliminary algorithmic ideas to extend our work on policy iteration.

7.2.1 Approximation with Error Bounds

Often, strict optimality requirements cause computational difficulties. A good compromise is to search for policies that are within some bound of optimal. Our framework is easily generalized to allow for this.

Instead of a value-preserving transformation, we could define an ϵ-value-preserving transformation, which insures that the value at all states decreases by at most ϵ. We can perform such transformations with no modifications to any of our linear programs. We simply need to relax the requirement on the value for ϵ that is returned. It is

easily shown that using an ϵ-value-preserving transformation at each step leads to convergence to a policy that is within $\frac{\epsilon\beta}{1-\beta}$ of optimal for all states.

For controller reductions, relaxing the tolerance may lead to smaller controllers because some value can be sacrificed. For bounded DP updates, it may help in escaping from local optima. Though relaxing the tolerance for a bounded DP update could lead to a decrease in value for some states, a small "downward" step could lead to higher value overall in the long run. We are currently working on testing these hypotheses empirically.

7.2.2 Compact Problem Representations

The convergence results for our policy iteration algorithm apply regardless of how the DEC-POMDP is represented. However, if the problem is represented compactly, a naive implementation of the algorithm may be very inefficient. As in the POMDP case, we would like to extend dynamic programming to work more efficiently with compact representations.

One type of structure to be exploited is loose interactions between state or observation features. We are looking into whether the POMDP approaches contained in [8, 26] can be extended to the multiagent case. We would also like to extend the approaches that work with hierarchies of high-level actions [27, 68]. In extending these to the multiagent case, there will be synchronization issues to deal with. If multiple agents are executing actions which take more than one time step, it is possible for one agent to complete an action while others are still executing theirs.

7.2.3 Forward Search

The policy iteration algorithm that we presented does not require information about the initial state. This can certainly be viewed as a strength of the algorithm. However, if information about the initial state is available, there should be a way to use it.

For POMDPs, forward search algorithms have been developed to take advantage of knowledge of the start state [51, 28]. These algorithms use the fact that a POMDP has an equivalent belief state MDP to form a search tree with belief states for nodes.

Since there is no known way to convert a DEC-POMDP to an MDP, it is not obvious how these search algorithms can be extended to the multiagent case. In fact, from our worst-case complexity results, we should expect any search tree to be doubly exponential in the horizon length.

One interesting forward search algorithm was recently proposed for the finite-horizon case [67]. This algorithm forms a tree containing all possible joint policies at the leaves. Since it covers all joint policies, the tree can be doubly exponential in the worst case. However, some branches of the tree can be pruned. When only the first few steps of a policy are specified, the underlying MDP is used to provide an optimistic estimate of the value for the remaining steps. This estimate can be used to determine whether the rest of the joint policy should be examined. It remains to extend this approach to the infinite-horizon case.

7.2.4 General-Sum Games

In a general-sum game, there is a set of agents, each with its own set of strategies, and a strategy profile is defined to be a tuple of strategies for all agents. Each agent assigns a payoff to each strategy profile. The agents may be noncooperative, so the same strategy profile may be assigned different values for each agent.

The DEC-POMDP model can be extended to a general-sum game by allowing each agent to have its own reward function. In this case, the strategies are the local policies, and a strategy profile is a joint policy. This model is often called a *partially observable stochastic game (POSG)*. In [29], a dynamic programming algorithm was given for finite-horizon POSGs. The algorithm was shown to perform iterated elimination of

dominated strategies in the game. Roughly speaking, it eliminates strategies that are not useful for an agent, regardless of the strategies of the other agents.

Work remains to be done on extending the notion of a value-preserving transformation to the noncooperative case. One possibility is to redefine value-preserving transformations so that value is preserved for *all* agents. This is closely related to the idea of *Pareto optimality*. In a general-sum game, a strategy profile is said to be Pareto optimal if there does not exist another strategy profile that yields higher payoff for all agents. It seems that policy iteration using the revised definition of value-preserving transformation would tend to move the controller in the direction of the Pareto optimal set. Another possibility is to define value-preserving transformations with respect to specific agents. As each agent transforms its own controller, the joint controller should move towards a Nash equilibrium.

7.2.5 Handling Large Numbers of Agents

The problem representation presented in this dissertation grows exponentially with the number of agents, and is thus not feasible for large numbers of agents. However, a compact representation is possible if each agent interacts directly with just a few other agents. We can have a separate state space for each agent, factored transition probabilities, and a reward function that is the sum of local reward functions for clusters of agents. In this case, the problem size is exponential only in the maximum number of agents interacting directly. This idea is closely related to recent work on graphical games [36, 38].

Once we have a compact representation, the next question to answer is whether we can adapt policy iteration to work efficiently with the representation. This indeed seems possible. With the value-preserving transformations we presented, the nodes of the other agents are considered part of the hidden state of the agent under consideration. These techniques modify the controller of the agent to get value improvement

for all possible hidden states. When an agent's state transitions and rewards do not depend on some other agent, it should not need to consider that agent's nodes as part of its hidden state. A specific compact representation along with extensions of different algorithms was recently proposed in [47].

7.2.6 Distributed Planning and Learning

The algorithms in this thesis require a complete and accurate model of the system. In many realistic scenarios, each agent only has a partial model of the system, and they must come up with the best joint policy they can given limited information and communication. If each agent only interacts with a few other agents as described in the previous section, it should only need to model these local interactions. However, coordination problems may still arise because there are multiple optimal joint policies to consider.

For many problems, even the assumption of a partial model is unrealistic. In this case, reinforcement learning techniques are necessary. For MDPs, the basic dynamic programming algorithms have played a key role in the development of reinforcement learning algorithms. In addition, there has been some work along these lines for POMDPs. Assuming an agent has a way of updating a belief state, reinforcement learning methods can be applied in the belief state MDP. This is the approach taken in [52].

Whether the policy iteration framework for DEC-POMDPs can be used as the basis for a reinforcement learning algorithm is an interesting open problem. In a sense, each agent has a local belief state which includes uncertainty about the other agents' internal states. This could be used as the state for a single-agent learning algorithm. One problem is that there is not a fixed belief state MDP any more, as modifying one agent's controller changes the other agents' state spaces.

BIBLIOGRAPHY

[1] Astrom, Karl J. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications 10* (1965), 174–205.

[2] Babai, László, Fortnow, Lance, and Lund, Carsten. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity 1* (1991), 3–40.

[3] Becker, Raphen, Zilberstein, Shlomo, Lesser, Victor, and Goldman, Claudia V. Solving transition independent Markov decision processes. *Journal of Artificial Intelligence Research 22* (2004), 423–455.

[4] Bellman, Richard E. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[5] Bernstein, Daniel S., Givan, Robert, Immerman, Neil, and Zilberstein, Shlomo. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research 27*, 4 (2002), 819–840.

[6] Bernstein, Daniel S., Hansen, Eric A., and Zilberstein, Shlomo. Bounded policy iteration for decentralized POMDPs. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence* (2005).

[7] Boutilier, Craig, Dean, Thomas, and Hanks, Steve. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research 1* (1999), 1–93.

[8] Boutilier, Craig, and Poole, David. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (1996), pp. 1168–1175.

[9] Cassandra, Anthony, Littman, Michael L., and Zhang, Nevin L. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence* (1997), pp. 54–61.

[10] Cassandra, Anthony R. Optimal policies for partially observable Markov decision processes. Tech. Rep. 94-14, Brown University, 1994.

[11] Chalkiadakis, Georgios, and Boutilier, Craig. Coordination in multiagent reinforcement learning: A Bayesian approach. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems* (2003).

[12] Cheng, Hsien-Te. *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia, 1988.

[13] Claus, Caroline, and Boutilier, Craig. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (1998).

[14] Cogill, Randy, Rotkowitz, Michael, Van Roy, Benjamin, and Lall, Sanjay. An approximate dynamic programming approach to decentralized control of stochastic systems. In *Proceedings of the Forty-Second Allerton Conference on Communication, Control, and Computing* (2004).

[15] Dutech, Alain, Buffet, Olivier, and Charpillet, François. Multi-agent systems by incremental gradient reinforcement learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (2001), pp. 833–838.

[16] Emery-Montemerlo, Rosemary, Gordon, Geoff, Schneider, Jeff, and Thrun, Sebastian. Game theoretic control for robot teams. In *Proceedings of the IEEE International Conference on Robotics and Automation* (2005).

[17] Emery-Montemerlo, Rosemary, Gordon, Geoff, Schnieder, Jeff, and Thrun, Sebastian. Approximate solutions for partially observable stochastic games with common payoffs. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems* (2004).

[18] Feng, Zhengzhu, and Zilberstein, Shlomo. Region-based incremental pruning for POMDPs. In *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence* (2004), pp. 146–153.

[19] Feng, Zhengzhu, and Zilberstein, Shlomo. Efficient maximization in solving POMDPs. In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (2005).

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

[21] Gmytrasiewicz, Piotr, and Doshi, Prashant. A framework for sequential planning in multi-agent settings. *Journal of Artificial Intelligence Research 24* (2005), 1–31.

[22] Goldman, Claudia V., and Zilberstein, Shlomo. Optimizing information exchange in cooperative multi-agent systems. In *Proceedings of the Second International Conference on Autonomous Agents and Multi-agent Systems* (2003).

[23] Goldman, Claudia V., and Zilberstein, Shlomo. Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research 22* (2004), 143–174.

[24] Guestrin, Carlos, Koller, Daphne, and Parr, Ronald. Multiagent planning with factored MDPs. In *Proceedings of Advances in Neural Information Processing Systems 15* (2001), pp. 1523–1530.

[25] Guestrin, Carlos, Venkataraman, Shobha, and Koller, Daphne. Context specific multiagent coordination and planning with factored MDPs. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence* (2002), pp. 253–259.

[26] Hansen, Eric, and Feng, Zhengzhu. Dynamic programming for POMDPs using a factored state representation. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling* (2000), pp. 130–139.

[27] Hansen, Eric, and Zhou, Rong. Synthesis of hierarchical finite-state controllers for POMDPs. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling* (2003).

[28] Hansen, Eric A. *Finite-Memory Control of Partially Observable Systems*. PhD thesis, University of Massachusetts Amherst, Amherst, Massachusetts, 1998.

[29] Hansen, Eric A., Bernstein, Daniel S., and Zilberstein, Shlomo. Dynamic programming for partially observable stochastic games. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence* (2004), pp. 709–715.

[30] Hauskrecht, Milos. Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research 13* (2000), 33–94.

[31] Ho, Yu-Chi. Team decision theory and information structures. *Proceedings of the IEEE 68*, 6 (1980), 644–654.

[32] Howard, Ronald A. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[33] Hsu, Kai, and Marcus, Steven I. Decentralized control of finite state Markov processes. *IEEE Transactions on Automatic Control 27*, 2 (1982), 426–431.

[34] Kaelbling, Leslie P., Littman, Michael L., and Cassandra, Anthony R. Planning and acting in partially observable stochastic domains. *Artificial Intelligence 101*, 1-2 (1998), 99–134.

[35] Kapetanakis, Spiros, and Kudenko, Daniel. Reinforcement learning of coordination in cooperative multi-agent systems. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence* (2002), pp. 326–331.

[36] Kearns, Michael, Littman, Michael L., and Singh, Satinder. Graphical models for game theory. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence* (2001).

[37] Koller, Daphne, Megiddo, Nimrod, and von Stengel, Bernhard. Efficient computation of equilibria for extensive two-person games. *Games and Economic Behavior 14*, 2 (1994), 247–259.

[38] Koller, Daphne, and Milch, Brian. Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior 45*, 1 (2003), 181–221.

[39] Lark III, James W. *Applications of Best-First Heuristic Search to Finite-Horizon Partially Observed Markov Decision Processes.* PhD thesis, University of Virginia, 1990.

[40] Lewis, Harry. Complexity of solvable cases of the decision problem for predicate calculus. In *Proceedings of the Nineteenth Symposium on the Foundations of Computer Science* (1978), pp. 35–47.

[41] Littman, Michael L. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning* (1994), pp. 157–163.

[42] Littman, Michael L., Cassandra, Anthony R., and Kaelbling, Leslie Pack. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning* (1995), pp. 362–370.

[43] Lovejoy, William S. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research 28* (1991), 47–66.

[44] Madani, Omid, Hanks, Steve, and Condon, Anne. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence 147*, 1-2 (2003), 5–34.

[45] Murphy, Kevin. A survey of POMDP solution techniques. Tech. rep., University of California at Berkeley, 2000.

[46] Nair, Ranjit, Pynadath, David, Yokoo, Makoto, Tambe, Milind, and Marsella, Stacy. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (2003).

[47] Nair, Ranjit, Varakantham, Pradeep, Tambe, Milind, and Yokoo, Makoto. Networked distributed POMDPs: A synthesis of distributed constraint optimization and POMDPs. In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (2005).

[48] Ooi, James M., and Wornell, Gregory W. Decentralized control of a multiple access broadcast channel: Performance bounds. In *Proceedings of the 35th Conference on Decision and Control* (1996), pp. 293–298.

[49] Papadimitriou, Christos H. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[50] Papadimitriou, Christos H., and Tsitsiklis, John. On the complexity of designing distributed protocols. *Information and Control 53* (1982), 211–218.

[51] Papadimitriou, Christos H., and Tsitsiklis, John N. The complexity of Markov decision processes. *Mathematics of Operations Research 12*, 3 (1987), 441–450.

[52] Parr, Ronald, and Russell, Stuart. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (1995).

[53] Peshkin, Leonid. *Reinforcement Learning by Policy Search*. PhD thesis, Brown University, 2002.

[54] Peshkin, Leonid, Kim, Kee-Eung, Meuleau, Nicolas, and Kaelbling, Leslie Pack. Learning to cooperate via policy search. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence* (2000), pp. 489–496.

[55] Peterson, Gary L., and Reif, John R. Multiple-person alternation. In *Twentieth Annual Symposium on Foundations of Computer Science* (1979), pp. 348–363.

[56] Pineau, Joelle, Gordon, Geoffrey, and Thrun, Sebastian. Point-based value iteration: An anytime algorithm for POMDPs. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (2003).

[57] Platzman, Loren K. A feasible computational approach to infinite-horizon partially-observed Markov decision processes. Tech. rep., Georgia Institute of Technology, 1980. Reprinted in *Working Notes of the 1998 AAAI Fall Symposium on Planning Using Partially Observable Markov Decision Processes*.

[58] Poupart, Pascal, and Boutilier, Craig. Bounded finite state controllers. In *Proceedings of Advances in Neural Information Processing Systems 16* (2003).

[59] Poupart, Pascal, and Boutilier, Craig. VDCBPI: An approximate scalable algorithm for large scale POMDPs. In *Proceedings of Advances in Neural Information Processing Systems 17* (2004).

[60] Pynadath, David V., and Tambe, Milind. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research 16* (2002), 389–423.

[61] Radner, Roy. Team decision problems. *Annals of Mathematical Statistics 33* (1962), 857–881.

[62] Romanovskii, I.V. Reduction of a game with complete memory to a matrix game. *Soviet Mathematics 3* (1962), 678–681.

[63] Singh, Satinder P., Jaakkola, Tommi, and Jordan, Michael I. Learning without state-estimation in partially observable markovian decision processes. In *Proceedings of the Eleventh International Conference on Machine Learning* (1994).

[64] Smallwood, Richard D., and Sondik, Edward J. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research 21*, 5 (1973), 1071–1088.

[65] Sondik, Edward J. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research 26* (1978), 282–304.

[66] Sutton, Richard S., Precup, Doina, and Singh, Satinder. Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. Tech. Rep. 98-74, University of Massachusetts, Amherst, 1998.

[67] Szer, Daniel, and Charpillet, François. MAA*: A heuristic search algorithm for solving decentralized POMDPs. In *Proceedings of the Twenty-first Conference on Uncertainty in Artificial Intelligence* (2005).

[68] Theocharous, Georgios, and Kaelbling, Leslie. Approximate planning in POMDPs with macro-actions. In *Proceedings of Advances in Neural Information Processing Systems 16* (2003).

[69] Varaiya, Pravin, and Walrand, Jean. On delayed sharing patterns. *IEEE Transactions on Automatic Control 23*, 3 (1978), 443–445.

[70] Wang, Xiao Feng, and Sandholm, Tuomas. Reinforcement learning to play an optimal Nash equilibrium in team Markov games. In *Proceedings of Advances in Neural Information Processing Systems 10* (2002), pp. 1010–1016.

[71] Weiss, Gerhard. *Multiagent Systems*. The MIT Press, Cambridge, MA, 1999.

[72] Xuan, Ping, and Lesser, Victor. Multi-agent policies: From centralized ones to decentralized ones. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-agent Systems* (2002).

[73] Zhang, Nevin L., and Lee, Stephen S. Planning with partially observable Markov decision processes: Advances in exact solution methods. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence* (1998).