3

Decision Problems

Mykel J. Kochenderfer

The previous chapter focused on uncertainty, including how to build probabilistic models of uncertainty and use them to make inferences. This chapter focuses on how to make rational decisions based on a probabilistic model and utility function. We will focus on single-step decisions, reserving discussion of sequential decision problems for the next chapter. This chapter begins by introducing the foundations of utility theory and showing how it forms the basis for rational decision making under uncertainty. We will then show how notions of utility theory can be incorporated into the probabilistic graphical models introduced in the previous chapter to form what are called decision networks. Because many important decision problems involve interacting with other agents, we will briefly discuss game-theoretic models.

3.1 Utility Theory

The previous chapter began by discussing the need to compare the degree of belief with respect to two different statements. This chapter requires the ability to compare the degree of desirability of two different outcomes. We state our preferences using the following operators:

- $A \succ B$ if we prefer A over B.
- $A \sim B$ if we are indifferent between A and B.
- $A \succeq B$ if we prefer A over B or are indifferent.

Just as beliefs can be subjective, so can preferences.

In addition to comparing events, our preference operators can be used to compare preferences over uncertain outcomes. A *lottery* is a set of probabilities associated with a set of outcomes. For example, if $S_{1:n}$ is a set of outcomes and $p_{1:n}$ are their associated probabilities, then the lottery involving these outcomes and probabilities is written

$$[S_1 : p_1; \ldots; S_n : p_n]. \tag{3.1}$$

This section discusses how the existence of a real-valued measure of utility emerges from a set of assumptions about preferences. From this utility function, it is possible to define what it means to make rational decisions under uncertainty.

3.1.1 Constraints on Rational Preferences

Just as we imposed a set of constraints on beliefs, we will impose some constraints on preferences. These constraints are sometimes called the *von Neumann-Morgenstern axioms*, named after John von Neumann and Oskar Morgenstern, who formulated a variation of these axioms in the 1940s.

- *Completeness.* Exactly one of the following hold: $A \succ B$, $B \succ A$, or $A \sim B$.
- *Transitivity.* If $A \succeq B$ and $B \succeq C$, then $A \succeq C$.
- *Continuity.* If $A \succeq C \succeq B$, then there exists a probability p such that $[A : p; B : 1-p] \sim C$.
- *Independence.* If $A \succ B$, then for any C and probability p, $[A : p; C : 1-p] \succ [B : p; C : 1-p]$.

These are constraints on *rational preferences*. They say nothing about the preferences of actual human beings; in fact, there is strong evidence that humans are not very rational (Section 3.1.7). Our objective in this book is to understand rational decision making from a computational perspective so that we can build useful systems. The possible extension of this theory to understanding human decision making is of only secondary interest.

3.1.2 Utility Functions

Just as constraints on the comparison of plausibility of different statements lead to the existence of a real-valued probability measure, constraints on rational preferences lead to the existence of a real-valued *utility* measure. It follows from our constraints on rational preferences that there exists a real-valued utility function U such that

- $U(A) > U(B)$ if and only if $A \succ B$, and
- $U(A) = U(B)$ if and only if $A \sim B$.

The utility function is unique up to an *affine transformation*. In other words, for any constants $m > 0$ and b, $U'(S) = mU(S) + b$ if and only if the preferences induced by U' are the same as U. Utilities are like temperatures: you can compare temperatures using Kelvin, Celsius, or Fahrenheit, all of which are affine transformations of each other.

It follows from the constraints on rational preferences that the utility of a lottery is given by

$$U([S_1 : p_1; \ldots; S_n : p_n]) = \sum_{i=1}^{n} p_i U(S_i). \tag{3.2}$$

Suppose we are building a collision avoidance system. The outcome of an encounter of an aircraft is defined by whether the system alerts (A) and whether a collision results (C). Because A and C are binary, there are four possible outcomes. So long as our preferences are rational, we can write our utility function over the space of possible lotteries in terms of four parameters: $U(a^0, c^0)$, $U(a^1, c^0)$, $U(a^0, c^1)$, and $U(a^1, u^1)$. For example,

$$U([a^0, c^0 : 0.5; a^1, c^0 : 0.3; a^0, c^1 : 0.1; a^1, c^1 : 0.1]) \tag{3.3}$$

is equal to

$$0.5U(a^0, c^0) + 0.3U(a^1, c^0) + 0.1U(a^0, c^1) + 0.1U(a^1, c^1). \tag{3.4}$$

If the utility function is bounded, then we can define a *normalized utility function* where the best possible outcome is assigned utility 1 and the worst possible outcome is assigned utility 0. The utility of each of the other outcomes is scaled and translated as necessary.

3.1.3 Maximum Expected Utility Principle

We are interested in the problem of making rational decisions with imperfect knowledge of the state of the world. Suppose we have a probabilistic model $P(s' \mid o, a)$, which represents the probability that the state of the world becomes s' given that we observe o and take action a. We have a utility function $U(s')$ that encodes our preferences over the space of outcomes. Our *expected utility* of taking action a given observation o is given by

$$EU(a \mid o) = \sum_{s'} P(s' \mid a, o)U(s'). \tag{3.5}$$

The *principle of maximum expected utility* says that a rational agent should choose the action that maximizes expected utility

$$a^* = \arg\max_a EU(a \mid o). \tag{3.6}$$

Because we are interested in building rational agents, Equation (3.6) plays a central role in this book.

3.1.4 Utility Elicitation

In building a decision making or decision support system, it is often helpful to infer the utility function from a human or a group of humans. This approach is called *utility elicitation* or *preference elicitation*. One way to go about doing this is to fix the utility of the worst outcome $S_\bot$ to 0 and the best outcome $S_\top$ to 1. So long as the utilities of the outcomes are bounded, we can translate and scale the utilities without altering our preferences. If we want to determine the utility of outcome S, then we determine the probability p such that $S \sim [S_\top : p; S_\bot : 1 - p]$. It follows that $U(S) = p$.

In our collision avoidance example, the best possible event is to not alert and not have a collision, and so we set $U(a^0, c^0) = 1$. The worst possible event is to alert and have a collision, and so we set $U(a^1, c^1) = 0$. We define the lottery $L(p)$ to be $[a^0, c^0 : p; a^1, c^1 : 1 - p]$. To determine $U(a^1, c^0)$, we must find the p such that $(a^1, c^0) \sim L(p)$. Similarly, to determine $U(a^0, c^1)$, we find the p such that $(a^0, c^1) \sim L(p)$.

3.1.5 Utility of Money

It may be tempting to use monetary values to infer utility functions. For example, when one is building a decision support system for managing wildfires, it would be natural to define a utility function in terms of the monetary cost incurred by property damage and the monetary cost for deploying fire suppression resources. However, it is well known in economics that the utility of money, in general, is not linear. If there were a linear relationship between utility and money, then decisions should be made in terms of maximizing expected monetary value. Someone who tries to maximize expected monetary value would have no use for insurance because the expected monetary values of insurance policies are generally negative.

We can determine the utility of money using the elicitation process in Section 3.1.4. Of course, different people have different utility functions, but the function generally follows the curve shown in Figure 3.1. For small amounts of money, the curve is roughly linear—$100 is about twice as good at $50. For larger amounts of money, the relationship is often treated as logarithmic. The flattening of the curve makes sense; after all, $1000 is worth less to a billionaire than it is to the average person.

When discussing monetary utility functions, the three terms below are often used. To illustrate, assume A represents being given $50 and B represents a 50% chance of winning $100.

- *Risk neutral.* The utility function is linear. There is no preference between $50 and the 50% chance of winning $100 ($A \sim B$).
- *Risk seeking.* The utility function is concave up. There is a preference for the 50% chance of winning $100 ($A \prec B$).

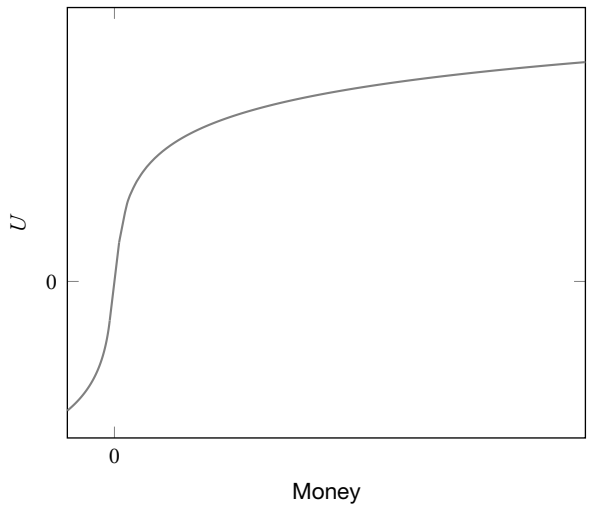


Figure 3.1 Utility of money.

- *Risk averse.* The utility function is concave down. There is a preference for the \$50 ($A \succ B$).

In the process of building decision-making systems, it is useful to use monetary value to inform the construction of the utility function. However, it is important to keep in mind the potentially nonlinear relationship between money and utility.

3.1.6 Multiple Variable Utility Functions

The collision avoidance utility function (Section 3.1.4) depended on two binary variables: whether there was an alert and whether there was a collision. We had to define the utility over all possible combinations of assignments to these two variables. If we had n binary variables, then we would have to specify 2^n parameters in the utility function. If we are able to normalize the utility function, then at least one of the parameters will be 0 and at least one of the parameters will be 1. We can attempt to represent utility functions compactly by leveraging different forms of independence between variables, similar to how Bayesian networks compactly represent joint probability distributions.

Under certain assumptions about the preference structure, we can represent a multiple variable utility function by using a sum of single-variable utility functions. If we have n variables $X_{1:n}$, then we can write

$$U(x_{1:n}) = \sum_{i=1}^{n} U(x_i). \tag{3.7}$$

To represent the utility function, assuming all variables are binary, we need only $2n$ parameters:

$$U(x_1^0), U(x_1^1), \ldots, U(x_n^0), U(x_n^1). \tag{3.8}$$

To illustrate the value of an additive decomposition of the utility function, we will add two additional variables to our collision avoidance example:

- *Strengthening* (S), which indicates whether the collision avoidance system instructed the pilots to strengthen (or increase) their climb or descent.
- *Reversal* (R), which indicates whether the collision avoidance system instructed the pilots to change direction (i.e., up to down or down to up).

If we did not try to leverage the preference structure over A, C, S, and R, then we would need $2^4 = 16$ parameters. With an additive decomposition, we need only eight parameters.

Although it is common to normalize utilities between 0 and 1, it may be more natural to use a different scheme for constructing the utility function. In the collision avoidance problem, it is easy to formulate the utility function in terms of costs associated with an alert, collision, strengthening, and reversal. If there is no alert, collision, strengthening, or reversal, then there is no cost—in other words, $U(a^0)$, $U(c^0)$, $U(s^0)$, and $U(r^0)$ are all equal to 0. The highest cost outcome is collision, and so we set $U(c^1) = -1$ and normalize the other utilities with respect to this outcome. An alert provides the lowest contribution to the overall cost, and a reversal costs more than a strengthening because it is more disruptive to the pilots. Given our assumptions and keeping $U(c^1)$ fixed at -1, we only have three free parameters to define our utility function.

The utility function for many problems cannot be additively decomposed into utility functions over individual variables as in Equation (3.7). For example, suppose our collision avoidance utility function was defined over three binary variables: whether the intruder comes close horizontally (H), whether the intruder comes close vertically (V), and whether the system alerts (A). A collision threat only really exists if both h^1 and v^1. Therefore, we cannot additively decompose the utility function over the variables independently. However, we can write $U(h, v, a) = U(h, v) + U(a)$.

We can make the additive decomposition explicit in a diagram, as shown in Figure 3.2. The *utility nodes* are indicated by diamonds. The parents of a utility node are *uncertainty nodes* representing the variables on which the utility node depends. If the parents of a utility node are discrete, then the utility function associated with that node can be represented as a table. If the parents of a utility node are continuous, then any real-valued function can be used to represent the utility. If the diagram has multiple utility nodes, their values are summed together to provide an overall utility value.

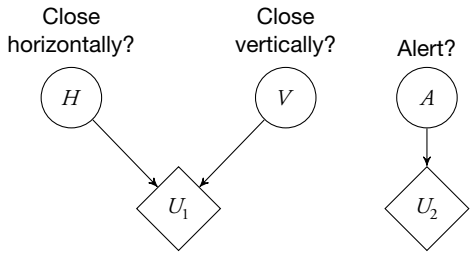


Figure 3.2 Additive decomposition of utility function.

3.1.7 Irrationality

Decision theory is a normative theory that is prescriptive, not a descriptive theory that is predictive of human behavior. Human judgment and preference often do not follow the rules of rationality outlined in Section 3.1.1. Even human experts may have an inconsistent set of preferences, which can be problematic when designing a decision support system that attempts to maximize expected utility.

Tversky and Kahneman studied the preferences of university students who answered questionnaires in a classroom setting. They presented students with questions dealing with the response to an epidemic. The students were to reveal their preference between the following two outcomes:

- A: 100% chance of losing 75 lives
- B: 80% chance of losing 100 lives

Most preferred B over A. From Equation (3.2), we know

$$U(\text{lose } 75) < 0.8\,U(\text{lose } 100). \tag{3.9}$$

They were then asked to choose between the following two outcomes:

- C: 10% chance of losing 75 lives
- D: 8% chance of losing 100 lives

Most preferred C over D. Hence, $0.1\,U(\text{lose } 75) > 0.08\,U(\text{lose } 100)$. We multiply both sides by 10 and get

$$U(\text{lose } 75) > 0.8\,U(\text{lose } 100). \tag{3.10}$$

Of course, Equations (3.9) and (3.10) result in a contradiction. We have made no assumption about the actual value of $U(\text{lose } 75)$ and $U(\text{lose } 100)$—we did not even assume that losing 100 lives was worse than losing 75 lives. Because Equation (3.2) follows directly from the von Neumann-Morgenstern axioms in Section 3.1.1, there must be a violation of at least one of the axioms—even though many people who select B and C seem to find the axioms agreeable.

The experiments of Tversky and Kahneman show that certainty often exaggerates losses that are certain relative to losses that are merely probable. They found that this *certainty effect* works with gains as well. A smaller gain that is certain is often preferred over a much greater gain that is only probable, in a way that the axioms of rationality are necessarily violated.

Tversky and Kahneman also demonstrated the *framing effect* using a hypothetical scenario in which an epidemic is expected to kill 600 people. They presented students with the following two outcomes:

- E: 200 people will be saved
- F: $1/3$ chance that 600 people will be saved and $2/3$ chance that no people will be saved

The majority of students chose E over F. They then asked them to choose between:

- G: 400 people will die
- H: $1/3$ chance that nobody will die and $2/3$ chance that 600 people will die

The majority of students chose H over G, even though E is equivalent to G and F is equivalent to H. This inconsistency is due to how the question is framed.

Many other cognitive biases can lead to deviations from what is prescribed by utility theory. Special care must be given when trying to elicit utility functions from human experts to build decision support systems. Although the recommendations of the decision support system may be rational, they may not exactly reflect human preferences in certain situations.

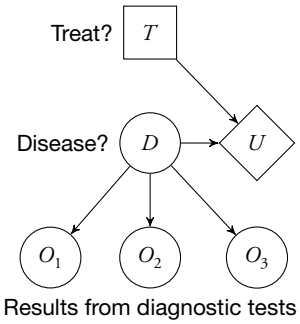
3.2 Decision Networks

We can extend the notion of Bayesian networks introduced in the last chapter to *decision networks* that incorporate actions and utilities. Decision networks are composed of three types of nodes:

- A *chance node* corresponds to a random variable (indicated by a circle).
- A *decision node* corresponds to each decision to be made (indicated by a square).
- A *utility node* corresponds to an additive utility component (indicated by a diamond).

There are three kinds of directed edges:

- A *conditional edge* ends in a chance node and indicates that the uncertainty in that chance node is conditioned on the values of all of its parents.
- An *informational edge* ends in a decision node and indicates that the decision associated with that node is made with knowledge of the values of its parents. (These edges are often drawn with dashed lines and are sometimes omitted from diagrams for simplicity.)



T	D	$U(T,D)$
0	0	0
0	1	−10
1	0	−1
1	1	−1

Figure 3.3 Diagnostic test decision network and utility function.

- A *functional edge* ends in a utility node and indicates that the utility node is determined by the outcomes of its parents.

Decision networks are sometimes called *influence diagrams*. Like Bayesian networks, decision networks cannot have cycles. Representing a decision problem as a decision network allows us to take advantage of the structure of the problem when computing the optimal decision with respect to a utility function. This chapter focuses on *single-shot* decision problems, in which decisions are made simultaneously. The next chapter will focus on problems for which decisions can be made sequentially.

Figure 3.3 shows an example decision network. In this network, we have a set of results from diagnostic tests that may indicate the presence of a particular disease. We need to decide whether to apply a treatment based on what is known about the diagnostic tests. The utility is a function of whether a treatment is applied and whether the disease is actually present. This network is used as a running example in this section.

3.2.1 Evaluating Decision Networks

As introduced in Section 3.1.3, the expected utility of a given o is

$$EU(a \mid o) = \sum_{s'} P(s' \mid a, o)U(s'). \tag{3.11}$$

The s' in Equation (3.11) represents an instantiation of the nodes in the decision network. We can use the equation to compute the expected utility of treating a disease for the decision network in Figure 3.3. For now, we will assume that we have the result from only the first diagnostic test and that it came back positive. If we wanted to make the knowledge of the first diagnostic test explicit in the diagram, then we would draw

an informational edge from O_1 to T, and we would have

$$EU(t^1 \mid o_1^1) = \sum_{o_3} \sum_{o_2} \sum_{d} P(d, o_2, o_3 \mid t^1, o_1^1) U(t^1, d, o_1^1, o_2, o_3). \tag{3.12}$$

We can use the chain rule for Bayesian networks and the definition of conditional probability to compute $P(d, o_2, o_3 \mid t^1, o_1^1)$. Because the utility node depends only on whether the disease is present and whether we treat it, we can simplify $U(t^1, d, o_1^1, o_2, o_3)$ to $U(t^1, d)$. Hence,

$$EU(t^1 \mid o_1^1) = \sum_{d} P(d \mid t^1, o_1^1) U(t^1, d). \tag{3.13}$$

Any of the exact or approximate inference methods introduced in the previous section can be used to evaluate $P(d \mid t^1, o_1^1)$. To decide whether to apply a treatment, we compute $EU(t^1 \mid o_1^1)$ and $EU(t^0 \mid o_1^1)$ and make the decision that provides the highest expected utility.

To evaluate general single-shot decision networks, we begin by instantiating the action nodes and observed chance nodes. We then apply any inference algorithm to compute the posterior over the parents of the utility nodes. Instead of summing over the instantiation of all variables in Equation (3.11), we only need to sum over the instantiations of the parents of the utility nodes. The optimal decision is the one that, when instantiated in the decision network, provides the highest expected utility.

A variety of methods have been developed over the years to make evaluating decision networks more efficient. One method involves removing action and chance nodes from decision networks if they have no children, as defined by conditional, informational, or functional edges. For example, in Figure 3.3, we can remove O_2 and O_3 because they have no children. We cannot remove O_1 because we treated it as observed, indicating there is an informational edge from O_1 to T (although it is not drawn explicitly).

3.2.2 Value of Information

So far, we have assumed that for the decision network in Figure 3.3, we have only observed o_1^1. Given the positive result from that one diagnostic test alone, then we may decide against treatment. However, it may be beneficial to administer additional diagnostic tests to reduce the risk of not treating a disease that is really present. One way to decide which diagnostic test to administer is to compute the *value of information*.

In computing the value of information, we will use $EU^*(o)$ to denote the expected utility of an optimal action given observation o. The value of information about variable O' given o is

$$VOI(O' \mid o) = \left(\sum_{o'} P(o' \mid o) EU^*(o, o') \right) - EU^*(o). \tag{3.14}$$

In other words, the value of information about a variable is the increase in expected utility with the observation of that variable. The expected utility can only increase if the observation of the variable can lead to a different optimal decision. If observing a new variable O' makes no difference in the choice of action, then $EU^*(o, o') = EU^*(o)$ for all o', in which case Equation (3.14) evaluates to 0. For example, if the optimal decision is to treat the disease regardless of the outcome of the diagnostic test, then the value of observing the outcome of the test is 0.

The value of information only captures the increase in expected utility from making an observation. There may be a cost associated with making a particular observation. Some diagnostic tests may be quite inexpensive, such as a temperature reading; other diagnostic tests are more costly and invasive, such as a lumbar puncture. The value of information obtained by a lumbar puncture may be much greater than a temperature reading, but the costs of the tests should be taken into consideration.

The value-of-information metric is an important and often used metric for choosing what to observe. Sometimes the value-of-information metric is used to determine an appropriate sequence of observations. After each observation, the value of information is determined for the remaining unobserved variables. The unobserved variable with the greatest value of information is then selected for observation. If there are costs associated with making different observations, then these costs are subtracted from the value of information when determining which variable to observe. The process continues until it is no longer beneficial to observe any more variables. The optimal action is then chosen. This greedy selection of observations is only a heuristic and may not represent the truly optimal sequence of observations. The optimal selection of observations can be determined by using the techniques for sequential decision making introduced in later chapters.

3.2.3 Creating Decision Networks

Decision networks are a powerful framework for building decision support systems. So far, we have discussed the key elements in building decision networks and how to use them to make optimal single-shot decisions. We will briefly discuss the procedure for creating a decision network.

The first step is to identify the space of possible actions. For an airborne collision avoidance system, the actions may be to climb, descend, or do nothing. In some problems, it may be desirable to factor the space of actions into multiple decision variables. In a collision avoidance system that can recommend both horizontal and vertical maneuvers, one decision variable may govern whether we climb or descend and another variable may govern whether we turn left or right.

The next step is identifying the observed and unobserved variables relevant to the problem. If we have an electro-optical sensor on our collision avoidance system, we may

be able to observe the relative angle to another aircraft, and so this angular measurement would correspond to one of the observed chance nodes. The true position of an intruding aircraft is relevant to the problem, but it cannot be observed directly, and so it is represented by an unobserved variable in the network.

We then identify the relationships between the various chance and decision nodes. Determining these relationships can be done by using expert judgment, learning from data as discussed in Section 2.4, or a combination of both. Often designers try to choose the direction of arrows to reflect causality from one node to another.

Once the relationships of the chance and decision nodes are identified, we choose models to represent the conditional probability distributions. For discrete nodes, an obvious model is a tabular representation. For continuous nodes, we can choose a parametric model such as a linear Gaussian model. The parameters of these models can then be specified by experts or estimated from the data by using the techniques introduced in Section 2.3.

We introduce the utility nodes and add functional edges from the relevant chance and decision nodes. The parameters of the utility nodes can be determined from preference elicitation from human experts (Section 3.1.4). The parameters can also be tuned so that the optimal decision according to the decision network matches the decisions of human experts.

The decision network should then be validated and refined by human experts. Given a decision scenario, the decision network can be used to determine the optimal action. That action can be compared against what a human expert would recommend. Generally, inspection of many decision scenarios is required before confidence can be established in a decision network.

If there is disagreement between the decision network and a human expert, then the decision network can be inspected to determine why that particular action was selected as optimal. In some cases, closer inspection of the model may result in revising the conditional probabilities, modifying the relationship between variables, changing the parameters in the utility nodes, or introducing new variables into to the model. Sometimes further study results in the human experts revising their choice of action. Several development iterations may be required before an appropriate decision network is found.

3.3 Games

This chapter has focused on making rational decisions with an assumed model of the environment. The methods introduced in this chapter so far can, of course, be applied to environments containing other agents so long as our probabilistic model captures the effects of the behavior of the other agents. However, there are many cases in which we do not have a probabilistic model of the behavior of the other agents, but we do have a

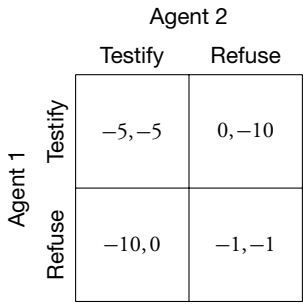


Figure 3.4 Prisoner's dilemma.

model of their utilities. Making decisions in such situations is the subject of *game theory*, which will be briefly discussed in this section.

3.3.1 Dominant Strategy Equilibrium

One of the most famous problems in game theory is the *prisoner's dilemma*. We have two agents who are prisoners and are being interrogated separately. Each is given the option to testify against the other. If one testifies and the other does not, then the one who testifies gets released, and the other gets ten years in prison. If both testify, then they both get five years. If both refuse to testify, then they both get one year.

The utilities of the two agents in the prisoner's dilemma are shown in Figure 3.4. The first component in the utility matrix is associated with Agent 1, and the second component is associated with Agent 2. It is assumed that the utility matrix is common knowledge between the two agents. The two agents must select their actions simultaneously without knowledge of the other agent's action.

In games like the prisoner's dilemma, the strategy chosen by an agent in a game can be either a

- *pure strategy*, in which an action is chosen deterministically, or
- *mixed strategy*, in which an action is chosen probabilistically.

Of course a pure strategy is just a special case of a mixed strategy in which probability 1 is assigned to a single action. The mixed strategy that assigns probability 0.7 to testifying in the prisoner's dilemma can be written using a notation similar to that used earlier for lotteries:

$$[\text{Testify} : 0.7; \text{Refuse} : 0.3].$$

The utility of a mixed strategy can be written in terms of utilities associated with pure strategies:

$$U([a_1 : p_1; \ldots; a_n : p_n]) = \sum_{i=1}^{n} p_i U(a_i). \tag{3.15}$$

The strategy of agent i is denoted s_i. A *strategy profile*, $s_{1:n}$, is an assignment of strategies to all n agents. The strategy profile of all agents except for that of agent i is written s_{-i}. The utility of agent i given a strategy profile $s_{1:n}$ is written $U_i(s_{1:n})$ or, alternatively, $U_i(s_i, s_{-i})$.

A *best response* of agent i to the strategy profile s_{-i} is a strategy s_i^* such that $U_i(s_i^*, s_{-i}) \geq U_i(s_i, s_{-i})$ for all strategies s_i. There may be, in general, multiple different best responses given s_{-i}. In some games, there may exist an s_i that is a best response to all possible s_{-i}, in which case s_i is called a *dominant strategy*. For example, in the prisoner's dilemma, Agent 1 is better off testifying regardless of whether Agent 2 testifies or refuses. Hence, testifying is the dominant strategy for Agent 1. Because the game is symmetric, testifying is also the dominant strategy for Agent 2. When all agents have a dominant strategy, their combination is called a *dominant strategy equilibrium*.

The prisoner's dilemma has generated significant interest because it shows how playing individual best responses can result in a suboptimal outcome for all agents. The dominant strategy equilibrium results in both agents testifying and receiving five years in prison. However, if they both refused to testify, then they would have both only received one year in prison.

3.3.2 Nash Equilibrium

Suppose we have two aircraft on a collision course, and the pilots of each aircraft must choose between climb or descend to avoid collision. If the pilots both choose the same maneuver, then there is a crash with utility -4 to both pilots. Because climbing requires more fuel than descending, there is an additional penalty of -1 to any pilot who decides to climb. The utility matrix is shown in Figure 3.5.

In the collision avoidance game, there does not exist a dominant strategy equilibrium. The best response of a particular pilot depends on the decision of the other pilot. There is an alternative equilibrium concept known as the *Nash equilibrium*. A strategy profile is a Nash equilibrium if no agent can benefit by switching strategy, given that the other agents adhere to the profile. In other words, $s_{1:n}$ is a Nash equilibrium if s_i is a best response to s_{-i} for all agents i.

There are two pure-strategy Nash equilibria in the collision avoidance game, namely, (Climb, Descend) and (Descend, Climb). It has been proven that every game has at least one Nash equilibrium, which may or may not include pure strategies. There are no known polynomial time algorithms for finding Nash equilibria for general games,

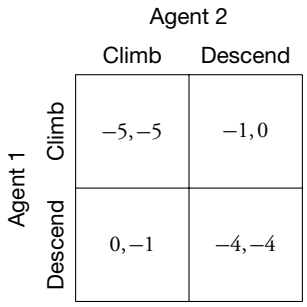


Figure 3.5 Collision avoidance game.

although the complexity of finding Nash equilibria is not NP-complete (instead it belongs to the complexity class known as PPAD).

3.3.3 Behavioral Game Theory

When one is building a decision-making system that must interact with humans, information about the Nash equilibrium is not always helpful. Humans often do not play a Nash equilibrium strategy. First of all, it may be unclear which equilibrium to adopt if there are many different equilibria in the game. For games with only one equilibrium, it may be difficult for a human to compute the Nash equilibrium because of cognitive limitations. Even if human agents can compute the Nash equilibrium, they may doubt that their opponents can perform that computation.

An area known as *behavioral game theory* aims to model human agents. Many different behavioral models exist, but the *logit level-k* model, sometimes called the *quantal level-k model*, has become popular recently and tends to work well in practice. The logit level-k model captures the assumption that humans are

- more likely to make errors when those errors are less costly, and
- limited in the number of steps of strategic look-ahead (as in "I think that you think that I think...").

The model is defined by

- a *precision parameter* $\lambda \geq 0$, which controls sensitivity to utility differences (0 is insensitive); and
- a *depth parameter* $k > 0$, which controls the depth of rationality.

In the logit level-k model, a level-0 agent selects actions uniformly. A level-1 agent assumes the opponents adopt level-0 strategies and selects actions according to the logit distribution

$$P(a_i) \propto e^{\lambda U_i(a_i, s_{-i})}, \tag{3.16}$$

where s_{-i} represents the assumed strategy profile of the other agents. A level-k agent assumes that the other agents adopt level $k-1$ strategies and select their own actions according to Equation (3.16). The parameters k and λ can be learned from data by using techniques discussed in the previous chapter.

To illustrate the logit level-k model, we will use the *traveler's dilemma*. In this game, an airline loses two identical suitcases from two travelers. The airline asks the travelers to write down the value of their suitcases, which can be between \$2 and \$100, inclusive. If both put down the same value, then they both get that value. The traveler with the lower value gets their value plus \$2. The traveler with the higher value gets the lower value minus \$2. In other words, the utility function is as follows:

$$U_i(a_i, a_{-i}) = \begin{cases} a_i & \text{if } a_i = a_{-i} \\ a_i + 2 & \text{if } a_i < a_{-i} \\ a_{-i} - 2 & \text{otherwise} \end{cases}. \tag{3.17}$$

Most people tend to put down between \$97 and \$100. However, somewhat counterintuitively, there is a unique Nash equilibrium of only \$2.

Figure 3.6 shows the strategies of the logit level-k model with different values for λ and k. The level-0 agent selects actions uniformly. The level-1 agent is peaked toward the upper end of the spectrum, with the precision parameter controlling the spread. As k increases, the difference between the strategies for $\lambda = 0.3$ and $\lambda = 0.5$ becomes less apparent. Human behavior can often be modeled well by logit level 2; as we can see, this provides a better model of human behavior than the Nash equilibrium.

3.4 Summary

- Rational decision making combines probability and utility theory.
- The existence of utility function follows from constraints on rational preferences.
- A rational decision is one that maximizes expected utility.
- We can build rational decision-making systems based on utility functions inferred from humans.
- Humans are not always rational.
- Decision networks compactly represent decision problems.
- Behavioral game theory is useful when making decisions involving multiple agents.

3.5 Further Reading

The theory of expected utility was initiated by Bernoulli in 1793 [2]. The axioms for rational decision making presented in Section 3.1.1 are based on those of Neumann and Morgenstern in their classic text, *Theory of Games and Economic Behavior* [3]. Neumann

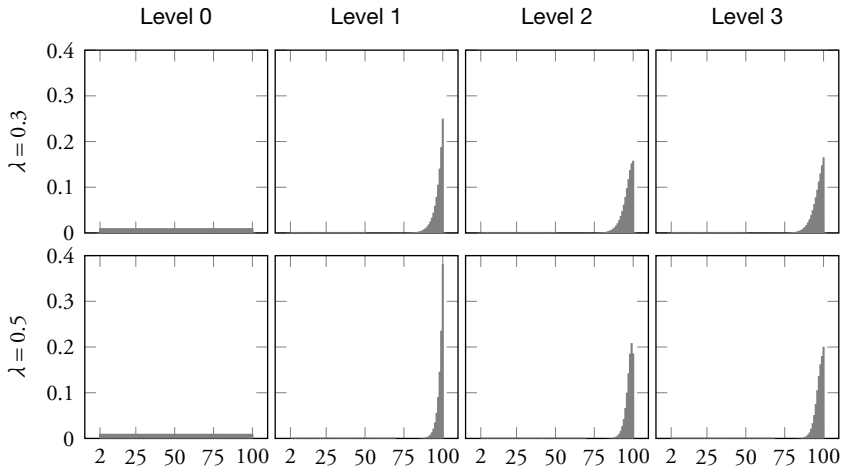


Figure 3.6 Logit level-k for traveler's dilemma.

and Morgenstern prove that those axioms lead to the existence of a utility function and establish the basis for the maximum expected utility principle [3]. Schoemaker provides an overview of the development of utility theory [4], and Fishburn surveys the field [5]. Russell and Norvig discuss the importance of the maximum expected utility principle to the field of artificial intelligence [6].

Farquhar surveys a variety of methods for utility elicitation [7], and Markowitz discusses the utility of money [8]. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs* by Keeney and Raiffa provides an overview of multiple attribute utility theory [9]. The book discusses the assumptions about preference structure that allow certain kinds of utility function decompositions, including the additive decomposition discussed in Section 3.1.6.

The example of irrational preferences in Section 3.1.7 is taken from Tversky and Kahneman [1]. Kahneman and Tversky provide a critique of expected utility theory and introduce an alternative model called prospect theory that appears to be more consistent with human behavior [10]. Several recent books discuss human irrationality, including *Predictably Irrational: The Hidden Forces That Shape Our Decisions* [11] and *How We Decide* [12].

The textbook *Bayesian Networks and Decision Graphs* by Jensen and Nielsen discusses decision networks. Early papers by Shachter provide algorithms for evaluating decision networks [14], [15]. Howard introduces the quantitative concept of the value of information [16], which has been applied to decision networks [17], [18].

Game theory is a broad field, and there are several standard introductory books [19]–[21]. Koller and Milch extend decision networks to a game-theoretic context [22].

Daskalakis, Goldberg, and Papadimitriou discuss the complexity of computing Nash equilibria [23]. Camerer provides an overview of behavioral game theory [24]. Wright and Leyton-Brown discuss behavioral game theory and show how to extract parameters from experimental data of human behavior [25], [26].

References

1. A. Tversky and D. Kahneman, "The Framing of Decisions and the Psychology of Choice," *Science*, vol. 211, no. 4481, pp. 453–458, 1981. DOI: 10.1126/science.7455683.
2. D. Bernoulli, "Exposition of a New Theory on the Measurement of Risk," *Econometrica*, vol. 22, no. 1, pp. 23–36, 1954. DOI: 10.2307/1909829.
3. J.V. Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, 3rd ed. Princeton, NJ: Princeton University Press, 1953.
4. P.J.H. Schoemaker, "The Expected Utility Model: Its Variants, Purposes, Evidence and Limitations," *Journal of Economic Literature*, vol. 20, no. 2, pp. 529–563, 1982.
5. P.C. Fishburn, "Utility Theory," *Management Science*, vol. 14, no. 5, pp. 335–378, 1968.
6. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ: Pearson, 2010.
7. P.H. Farquhar, "Utility Assessment Methods," *Management Science*, vol. 30, no. 11, pp. 1283–1300, 1984.
8. H. Markowitz, "The Utility of Wealth," *Journal of Political Economy*, vol. 60, no. 2, pp. 151–158, 1952.
9. R.L. Keeney and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. New York: Cambridge University Press, 1993.
10. D. Kahneman and A. Tversky, "Prospect Theory: An Analysis of Decision Under Risk," *Econometrica*, vol. 47, no. 2, pp. 263–292, 1979. DOI: 10.2307/1914185.
11. D. Ariely, *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. New York: Harper, 2008.
12. J. Lehrer, *How We Decide*. New York: Houghton Mifflin, 2009.
13. F.V. Jensen and T.D. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. New York: Springer, 2007.
14. R.D. Shachter, "Evaluating Influence Diagrams," *Operations Research*, vol. 34, no. 6, pp. 871–882, 1986.

15. ——, "Probabilistic Inference and Influence Diagrams," *Operations Research*, vol. 36, no. 4, pp. 589–604, 1988.
16. R.A. Howard, "Information Value Theory," *IEEE Transactions on Systems Science and Cybernetics*, vol. 2, no. 1, pp. 22–26, 1966. DOI: 10.1109/TSSC.1966.300074.
17. S.L. Dittmer and F.V. Jensen, "Myopic Value of Information in Influence Diagrams," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1997.
18. R.D. Shachter, "Efficient Value of Information Computation," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.
19. R.B. Myerson, *Game Theory: Analysis of Conflict*. Cambridge, MA: Harvard University Press, 1997.
20. Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. New York: Cambridge University Press, 2009.
21. D. Fudenberg and J. Tirole, *Game Theory*. Cambridge, MA: MIT Press, 1991.
22. D. Koller and B. Milch, "Multi-Agent Influence Diagrams for Representing and Solving Games," *Games and Economic Behavior*, vol. 45, no. 1, pp. 181–221, 2003. DOI: 10.1016/S0899-8256(02)00544-4.
23. C. Daskalakis, P.W. Goldberg, and C.H. Papadimitriou, "The Complexity of Computing a Nash Equilibrium," *Communications of the ACM*, vol. 52, no. 2, pp. 89–97, 2009. DOI: 10.1145/1461928.1461951.
24. C.F. Camerer, *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton, NJ: Princeton University Press, 2003.
25. J.R. Wright and K. Leyton-Brown, "Behavioral Game Theoretic Models: A Bayesian Framework for Parameter Analysis," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2012.
26. J.R. Wright and K. Leyton-Brown, "Beyond Equilibrium: Predicting Human Behavior in Normal Form Games," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

4

Sequential Problems

Mykel J. Kochenderfer

The previous chapter discussed problems in which a single decision is to be made, but many important problems require the decision maker to make a series of decisions. The same principle of maximum expected utility still applies, but optimal decision making requires reasoning about future sequences of actions and observations. This chapter will discuss sequential decision problems in stochastic environments.

4.1 Formulation

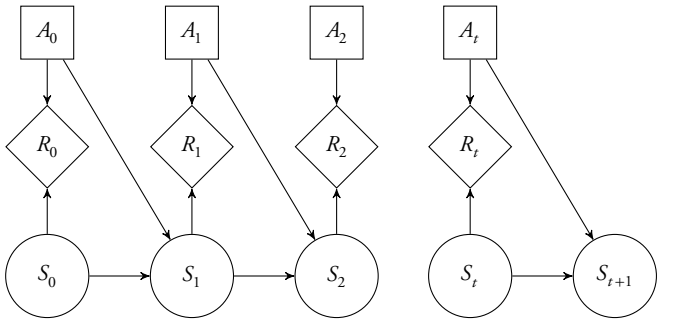
In this chapter, we will focus on a general formulation of sequential decision problems under the assumption that the model is known and that the environment is fully observable. Both of these assumptions will be relaxed in the next two chapters.

4.1.1 Markov Decision Processes

In a *Markov decision process* (MDP), an agent chooses action a_t at time t based on observing state s_t. The agent then receives a reward r_t. The state evolves probabilistically based on the current state and action taken by the agent. The assumption that the next state depends only on the current state and action and not on any prior state or action is known as the *Markov assumption*.

An MDP can be represented using a decision network as shown in Figure 4.1a. There are information edges (not shown in the figure) from $A_{0:t-1}$ and $S_{0:t}$ to A_t. The utility function is decomposed into rewards $R_{0:t}$.

We will focus on *stationary* MDPs in which $P(S_{t+1} \mid S_t, A_t)$ and $P(R_t \mid A_t, S_t)$ do not vary with time. Stationary MDPs can be compactly represented by a dynamic decision diagram as shown in Figure 4.1b. The *state transition function* $T(s' \mid s, a)$ represents the probability of transitioning from state s to s' after executing action a. The *reward function* $R(s, a)$ represents the expected reward received when executing



(a) General representation. (b) Stationary representation.

Figure 4.1 Markov decision process decision diagram.

action a from state s. We assume that the reward function is a deterministic function of s and a, but this need not be the case.

The problem of aircraft collision avoidance can be formulated as an MDP. The states represent the positions and velocities of our aircraft and the intruder aircraft, and the actions represent whether we climb, descend, or stay level. We receive a large negative reward for colliding with the other aircraft and a small negative reward for climbing or descending.

4.1.2 Utility and Reward

The rewards in an MDP are treated as components in an additively decomposed utility function (Section 3.1.6). In a *finite horizon* problem with n decisions, the utility associated with a sequence of rewards $r_{0:n-1}$ is simply

$$\sum_{t=0}^{n-1} r_t. \tag{4.1}$$

In an *infinite horizon* problem in which the number of decisions is unbounded, the sum of rewards can become infinite. Suppose strategy A results in a reward of 1 per time step and strategy B results in a reward of 100 per time step. Intuitively, a rational agent should prefer strategy B over strategy A, but both provide the same infinite expected utility.

There are several ways to define utility in terms of individual rewards in infinite horizon problems. One way is to impose a *discount factor* γ between 0 and 1. The utility

is given by

$$\sum_{t=0}^{\infty} \gamma^t r_t. \tag{4.2}$$

So long as $0 \leq \gamma < 1$ and the rewards are finite, the utility will be finite. The discount factor makes it so that rewards in the present is worth more than rewards in the future, a concept that also appears in economics.

Another way to define utility in infinite horizon problems is to use the *average reward* given by

$$\lim_{n \to \infty} \frac{1}{n} \sum_{t=0}^{n-1} r_t. \tag{4.3}$$

This book focuses primarily on optimizing with respect to discounted rewards over an infinite horizon.

4.2 Dynamic Programming

The optimal strategy can be found by using a computational technique called *dynamic programming*. Although we will focus on dynamic programming algorithms for MDPs, dynamic programming is a general technique that can be applied to a wide variety of other problems. For example, dynamic programming can be used in computing a Fibonacci sequence, finding the longest common subsequence between two strings, and finding the most likely sequence of states in a hidden Markov model. In general, algorithms that use dynamic programming for solving MDPs are much more efficient than brute force methods.

4.2.1 Policies and Utilities

A *policy* in an MDP determines what action to select given the past history of states and actions. The action to select at time t, given the history $h_t = (s_{0:t}, a_{0:t-1})$, is written $\pi_t(h_t)$. Because the future state sequence and rewards depend only on the current state and action (as made apparent in the conditional independence assumptions in Figure 4.1a), we can restrict our attention to policies that depend only on the current state.

In infinite horizon MDPs in which the transitions and rewards are stationary, we can further restrict our attention to stationary policies. We will write the action associated with stationary policy π in state s as $\pi(s)$, without the temporal subscript. In finite horizon problems, however, it may be beneficial to select a different action depending on how many time steps are remaining. For example, when playing basketball, it is generally not a good strategy to attempt a half-court shot unless there are just a couple seconds remaining in the game.

The expected utility of executing π from state s is denoted $U^\pi(s)$. In the context of MDPs, U^π is often referred to as the *value function*. An *optimal policy* π^* is a policy that maximizes expected utility:

$$\pi^*(s) = \arg\max_\pi U^\pi(s) \tag{4.4}$$

for all states s. Depending on the model, there may be multiple policies that are optimal.

4.2.2 Policy Evaluation

Computing the expected utility obtained from executing a policy is known as *policy evaluation*. We may use dynamic programming to evaluate the utility of a policy π for t steps. If we do not execute the policy at all, then $U_0^\pi(s) = 0$. If we execute the policy for one step, then $U_1^\pi(s) = R(s, \pi(s))$, which is simply the expected reward associated with the first step.

Suppose we know the utility associated with executing π for $t-1$ steps. Computing the utility associated with executing π for t steps can be computed as follows:

$$U_t^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U_{t-1}^\pi(s'), \tag{4.5}$$

where γ is the discount factor, which can be set to 1 if discounting is not desired. Algorithm 4.1 shows how to iteratively compute the expected utility of a policy up to an arbitrary horizon n.

Algorithm 4.1 Iterative policy evaluation

```
function IterativePolicyEvaluation(π, n)
    U_0^π(s) ← 0 for all s
    for t ← 1 to n
        U_t(s) ← R(s, π(s)) + γ Σ_{s'} T(s' | s, π(s)) U_{t-1}^π(s') for all s
    return U_n
```

For an infinite horizon with discounted rewards,

$$U^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^\pi(s'). \tag{4.6}$$

We can compute U^π arbitrarily well with enough iterations of iterative policy evaluation. An alternative is to solve a system of n linear equations where n is the number of states. We can represent the system of equations in matrix form:

$$\mathbf{U}^\pi = \mathbf{R}^\pi + \gamma \mathbf{T}^\pi \mathbf{U}^\pi, \tag{4.7}$$

where $\mathbf{U}^\pi$ and $\mathbf{R}^\pi$ are the utility and reward functions represented as an n-dimensional vector. The $n \times n$ matrix $\mathbf{T}^\pi$ contains state transition probabilities. The probability of transitioning from the ith state to the jth state is given by $\mathbf{T}^\pi_{ij}$.

We can easily solve for U^π as follows:

$$\mathbf{U}^\pi - \gamma \mathbf{T}^\pi \mathbf{U}^\pi = \mathbf{R}^\pi \tag{4.8}$$

$$(\mathbf{I} - \gamma \mathbf{T}^\pi)\mathbf{U}^\pi = \mathbf{R}^\pi \tag{4.9}$$

$$\mathbf{U}^\pi = (\mathbf{I} - \gamma \mathbf{T}^\pi)^{-1}\mathbf{R}^\pi. \tag{4.10}$$

Solving for $\mathbf{U}^\pi$ in this way requires $O(n^3)$ time.

4.2.3 Policy Iteration

Policy evaluation can be used in a general process called *policy iteration* for computing an optimal policy π^* as outlined in Algorithm 4.2. Policy iteration starts with any policy π_0 and iterates the following two steps:

- *Policy evaluation.* Given the current policy π_k, compute U^{π_k}.
- *Policy improvement.* Using U^{π_k}, compute a new policy using the equation in Line 5.

The algorithm terminates when there is no more improvement. Because every step leads to improvement and there are finitely many policies, the algorithm terminates with an optimal solution.

Algorithm 4.2 Policy iteration

```
1: function PolicyIteration(π_0)
2:     k ← 0
3:     repeat
4:         Compute U^{π_k}
5:         π_{k+1}(s) = arg max_a (R(s,a) + γ Σ_{s'} T(s' | s,a) U^{π_k}(s')) for all states s
6:         k ← k + 1
7:     until π_k = π_{k−1}
8:     return π_k
```

There are many variants of policy iteration. One method known as *modified policy iteration* involves approximating U^{π_k} using only a few iterations of iterative policy evaluation instead of computing the utility function exactly.

4.2.4 Value Iteration

An alternative to policy iteration is *value iteration* (Algorithm 4.3), which is often used because it is simple and easy to implement. First, let us compute the optimal value

function U_n associated with a horizon of n and no discounting. If $n = 0$, then $U_0(s) = 0$ for all s. We can compute U_n recursively from this base case

$$U_n(s) = \max_a \left(R(s,a) + \sum_{s'} T(s' \mid s,a) U_{n-1}(s') \right). \tag{4.11}$$

For an infinite horizon problem with discount γ, it can be proven that the value of an optimal policy satisfies the *Bellman equation*:

$$U^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s') \right). \tag{4.12}$$

The optimal value function U^* appears on both sides of the equation. Value iteration approximates U^* by iteratively updating the estimate of U^* using Equation (4.12). Once we know U^*, we can extract an optimal policy by using

$$\pi(s) \leftarrow \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U^*(s') \right). \tag{4.13}$$

Algorithm 4.3 Value iteration

```
function ValueIteration
    k ← 0
    U_0(s) ← 0 for all states s
    repeat
        U_{k+1}(s) ← max_a [R(s,a) + γ Σ_{s'} T(s' | s,a) U_k(s')] for all states s
        k ← k + 1
    until convergence
    return U_k
```

Algorithm 4.3 shows U_0 being initialized to 0, but value iteration can be proven to converge with any bounded initialization (i.e., $|U_0(s)| < \infty$ for all s). It is common to initialize the utility function to a guess of the optimal value function in an attempt to speed convergence.

A common termination condition for the loop in Algorithm 4.3 is when $||U_k - U_{k-1}|| < \delta$. In this context, $||\cdot||$ denotes the *max norm*, where $||U|| = \max_s |U(s)|$. The quantity $||U_k - U_{k-1}||$ is known as the *Bellman residual*.

If we want to guarantee that our estimate of the value function is within ϵ of U^* at all states, then we should choose δ to be $\epsilon(1-\gamma)/\gamma$. As γ approaches 1, the termination threshold becomes smaller, implying slower convergence. In general, the less future rewards are discounted, the more we have to iterate to look out to an acceptable horizon.

If we know that $||U_k - U^*|| < \epsilon$, then we can bound the *policy loss* of the policy extracted from U_k. If the extracted policy is π, then the policy loss is defined as $||U^\pi - U^*||$. It can be proven that $||U_k - U^*|| < \epsilon$ implies that the policy loss is less than $2\epsilon\gamma/(1-\gamma)$.

4.2.5 Grid World Example

To illustrate value iteration, we will use a 10×10 grid world problem. Each cell in the grid represents a state in an MDP. The available actions are up, down, left, and right. The effects of these actions are stochastic. We move one step in the specified direction with probability 0.7, and we move one step in one of the three other directions, each with probability 0.1. If we bump against the outer border of the grid, then we do not move at all.

We receive a cost of 1 for bumping against the outer border of the grid. There are four cells in which we receive rewards upon entering:

- $(8,9)$ has a reward of $+10$
- $(3,8)$ has a reward of $+3$
- $(5,4)$ has a reward of -5
- $(8,4)$ has a reward of -10

The coordinates are specified using the matrix convention in which the first coordinate is the row starting from the top and the second coordinate is the column starting from the left. The cells with rewards of $+10$ and $+3$ are absorbing states where no additional reward is ever received from that point onward.

Figure 4.2a shows the result of the first sweep of value iteration with a discount factor of 0.9. After this first sweep, the value function is simply the maximum expected immediate reward—i.e., $\max_a R(s,a)$. The gray pointers indicate the optimal actions from the cells as determined by Equation (4.13). As indicated in the figure, all actions are optimal for the interior cells. For the cells adjacent to the wall, the optimal actions are in the directions away from the wall.

Figure 4.2b shows the result of the second sweep. The values at the states with non-zero rewards remain the same, but the values are dispersed to adjacent cells. The value of the cells is based on expected discounted rewards after two time steps. Consequently, cells more than one step away from an absorbing cell or a cell bordering a wall have zero value. Cells within one step have had their optimal action set updated to direct us to positive rewards and away from negative rewards.

Figures 4.3a and 4.3b show the value function and policy after three and four sweeps, respectively. The value associated with the $+3$ and $+10$ cells spread outward over the grid. As the value is propagated further throughout the grid, there are fewer ties for optimal actions for the various cells.

Figures 4.4a and 4.4b show the value function and policy at convergence for $\gamma = 0.9$ and $\gamma = 0.5$, respectively. When $\gamma = 0.9$, even the cells on the left side of the grid have positive value. When the rewards are discounted more steeply with $\gamma = 0.5$, the +3 and +10 rewards do not propagate as far. The effect of steeper discounting can also be seen in the difference in policy at cell $(4, 8)$. With discounting at 0.5, the best strategy is to head straight to the +3 cell, whereas with discounting at 0.9, the best strategy is to head toward the +10 cell.

4.2.6 Asynchronous Value Iteration

The value iteration algorithm in Section 4.2.4 computes U_{k+1} based on U_k for *all* the states at each iteration. In *asynchronous value iteration*, only a subset of the states may be updated per iteration. It can be proven that, so long as the value function is updated at each state infinitely often, the value function is guaranteed to converge to the optimal value function.

Gauss-Seidel value iteration is a type of asynchronous value iteration. It sweeps through an ordering of the states and applies the following update:

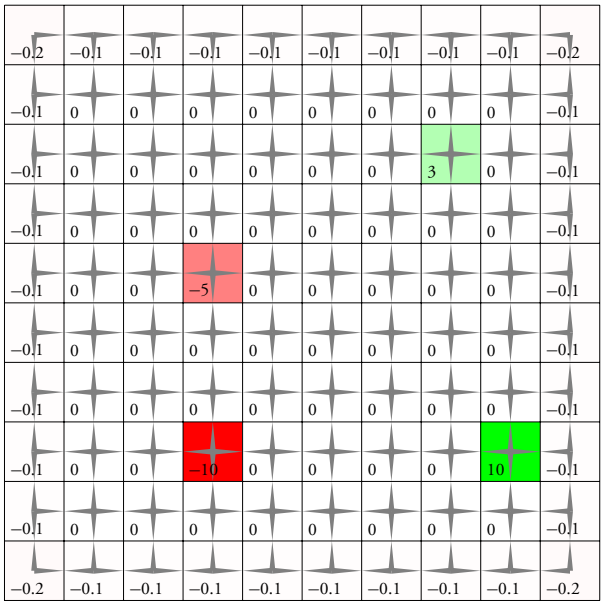
$$U(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right). \tag{4.14}$$

With Gauss-Seidel value iteration, we only have to keep one copy of state values in memory instead of two because the values are updated *in place*. In addition, Gauss-Seidel can converge more quickly than standard value iteration can depending on the ordering chosen.

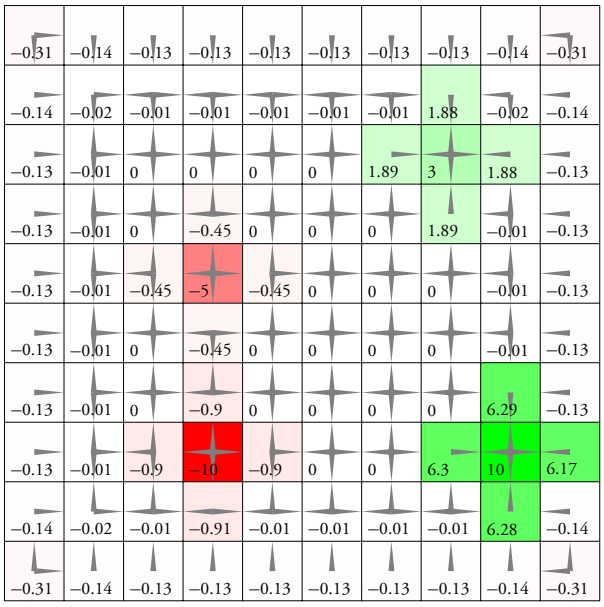
4.2.7 Closed- and Open-Loop Planning

The process of using a model to choose an action in a sequential problem is called *planning*. There are two general approaches to planning:

- *Closed-loop planning* accounts for future state information. The dynamic programming algorithms discussed in this chapter fall within this category. They involve developing a *reactive* plan (or policy) that can react to the different outcomes of the actions over time.
- *Open-loop planning* does not account for future state information. Many *path planning* algorithms fall within this category. They involve developing a static sequence of actions.



(a) Sweep 1.



(b) Sweep 2.

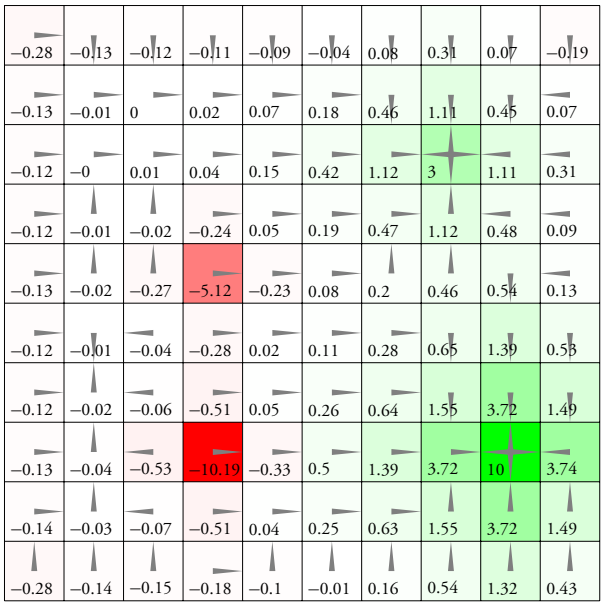
Figure 4.2 Value iteration with $\gamma = 0.9$.

(a) Sweep 3.

(b) Sweep 4.

Figure 4.3 Value iteration with $\gamma = 0.9$.

(a) $\gamma = 0.9$.



(b) $\gamma = 0.5$.

Figure 4.4 Value iteration at convergence.

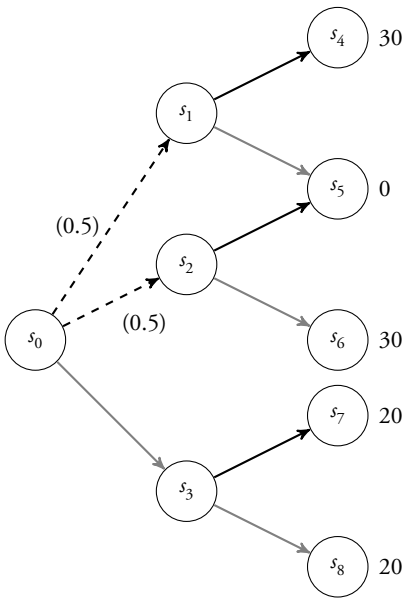


Figure 4.5 Example illustrating suboptimality of open-loop planning.

The advantage of closed-loop planning can be illustrated with the example in Figure 4.5. There are nine states, and we start in state s_0. There are two decision steps, where we must decide between going up (black arrows) or going down (gray arrows). The effects of the actions are deterministic, except that if we go up from s_0, then we end up in state s_1 half the time and in state s_2 half the time. We receive a reward of 30 in states s_4 and s_6 and a reward of 20 in states s_7 and s_8, as indicated in the figure.

There are exactly four open-loop plans: (up, up), (up, down), (down, up), and (down, down). In this simple example, it is easy to compute their expected utilities:

- $U(\text{up, up}) = 0.5 \times 30 + 0.5 \times 0 = 15$
- $U(\text{up, down}) = 0.5 \times 0 + 0.5 \times 30 = 15$
- $U(\text{down, up}) = 20$
- $U(\text{down, down}) = 20$

According to the set of open-loop plans, it is best to choose down from s_0 because our expected reward is 20 instead of 15.

Closed-loop planning, in contrast, takes into account the fact that we can base our next decision on the observed outcome of our first action. If we choose to go up from s_0, then we can choose to go down or up depending on whether we end up in s_1 or s_2, thereby guaranteeing a reward of 30.

In sequential problems where the effects of actions are uncertain, closed-loop planning can provide a significant benefit over open-loop planning. However, in some domains, the size of the state space can make the application of closed-loop planning methods, such as value iteration, infeasible. Open-loop planning algorithms, although suboptimal in principle, can provide satisfactory performance. There are many open-loop planning algorithms, but this book will focus on closed-loop methods and ways for addressing large problems without sacrificing the ability to account for the availability of future information.

4.3 Structured Representations

The dynamic programming algorithms described earlier in this chapter have assumed that the state space is discrete. If the state space is determined by n binary variables, then the number of discrete states is 2^n. This exponential growth of discrete states restricts the direct application of algorithms such as value iteration and policy iteration to problems with only a limited number of state variables. This section discusses methods that can help solve higher dimensional problems by leveraging their structure.

4.3.1 Factored Markov Decision Processes

A *factored Markov decision process* compactly represents the transition and reward functions using a dynamic decision network. The actions, rewards, and states may be factored into multiple nodes. Figure 4.6 shows an example of a factored MDP with two decision variables (A and F), three state variables (B, D, and G), and two reward variables (C and E).

We can use decision trees to compactly represent the conditional probability distributions and reward functions. For example, the conditional probability distribution $P(G_{t+1} \mid D_t, F_t, G_t)$ shown in tabular form in Figure 4.7a can be represented using the decision tree in Figure 4.7b.

Additional efficiency can be gained using *decision diagrams* instead of decision trees. In trees, all nodes (except for the root) have exactly one parent, but nodes in decision diagrams can have multiple parents. Figure 4.8 shows an example of a decision tree and an equivalent decision diagram. Instead of requiring four leaf nodes as in the decision tree, the decision diagram only requires two leaf nodes.

4.3.2 Structured Dynamic Programming

Several dynamic programming algorithms exist for finding policies for factored MDPs. Algorithms such as *structured value iteration* and *structured policy iteration* perform updates on the leaves of the decision trees instead of all the states. These algorithms improve efficiency by *aggregating states* and leveraging the additive decomposition of

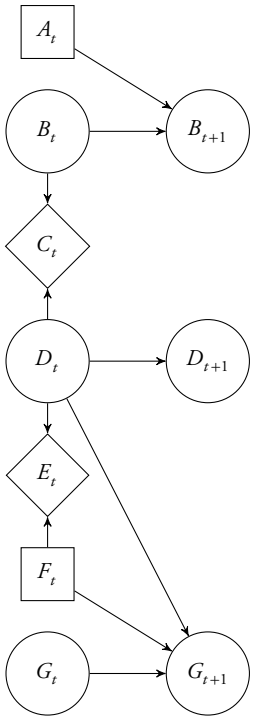
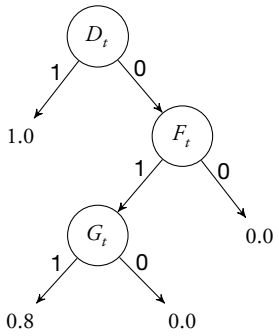


Figure 4.6 Factored Markov decision process.

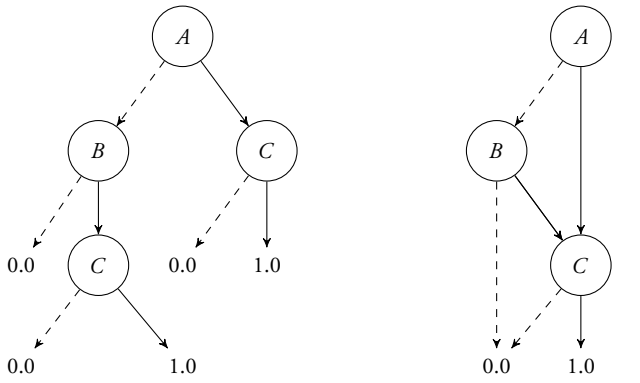
D_t	F_t	G_t	$P(g_{t+1}^1 \mid D_t, F_t, G_t)$
1	1	1	1.0
1	1	0	1.0
1	0	1	1.0
1	0	0	1.0
0	1	1	0.8
0	1	0	0.0
0	0	1	0.0
0	0	0	0.0

(a) Tabular form.



(b) Decision tree form.

Figure 4.7 Conditional distribution as a decision tree.



(a) Decision tree. (b) Decision diagram.

Figure 4.8 Tree- and graph-based representations of a conditional probability table. Dashed lines are followed from nodes when the outcome of the variable test is false.

the reward and value functions. The resulting policy is represented as a decision tree in which the interior nodes correspond to tests of the state variables and the leaf nodes correspond to actions.

4.4 Linear Representations

The methods presented in this chapter so far have required that the problems be discrete. Of course, we may discretize a problem that is naturally continuous, but doing so may not be feasible if the state or action spaces are large. This section presents a method for finding exact optimal policies for problems with continuous state and action spaces that meet certain criteria:

- *Dynamics are linear Gaussian.* The state transition function has the form

$$T(\mathbf{z} \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \boldsymbol{\Sigma}), \tag{4.15}$$

  where $\mathbf{T}_s$ and $\mathbf{T}_a$ are matrices that determine the mean of the next state $\mathbf{z}$ based on $\mathbf{s}$ and $\mathbf{a}$, and $\boldsymbol{\Sigma}$ is a covariance matrix that controls the amount of noise in the dynamics.
- *Reward is quadratic.* The reward function has the following form

$$R(\mathbf{s}, \mathbf{a}) = \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a}, \tag{4.16}$$

  where $\mathbf{R}_s = \mathbf{R}_s^\top \leq 0$ and $\mathbf{R}_a = \mathbf{R}_a^\top < 0$.

For simplicity, we will assume a finite horizon undiscounted reward problem, but the approach can be generalized to average reward and discounted infinite horizon problems as well. We may generalize Equation (4.11) for continuous state spaces by replacing the summation with an integral and making $T(s' \mid s, a)$ represent a probability density rather than a probability mass:

$$U_n(\mathbf{s}) = \max_{\mathbf{a}} \left(R(\mathbf{s}, \mathbf{a}) + \int T(\mathbf{z} \mid \mathbf{s}, \mathbf{a}) U_{n-1}(\mathbf{z}) d\mathbf{z} \right). \tag{4.17}$$

From our assumptions about T and R, we can rewrite the equation above:

$$U_n(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma) U_{n-1}(\mathbf{z}) d\mathbf{z} \right). \tag{4.18}$$

It can be proven inductively that $U_n(\mathbf{s})$ can be written in the form $\mathbf{s}^\top \mathbf{V}_n \mathbf{s} + q_n$. We can rewrite the equation above as

$$U_n(\mathbf{s}) = \max_{\mathbf{a}} \left(\mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma)(\mathbf{z}^\top V_{n-1} \mathbf{z} + q_{n-1}) d\mathbf{z} \right). \tag{4.19}$$

Simplifying, we get

$$U_n(\mathbf{s}) = q_{n-1} + \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + \int \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma) \mathbf{z}^\top \mathbf{V}_{n-1} \mathbf{z} d\mathbf{z} \right). \tag{4.20}$$

The integral in the equation above evaluates to

$$\text{Tr}(\Sigma \mathbf{V}_{n-1}) + (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a})^\top \mathbf{V}_{n-1} (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}), \tag{4.21}$$

where Tr represents the *trace* of a matrix, which is simply the sum of the main diagonal elements. We now have

$$U_n(\mathbf{s}) = q_{n-1} + \mathbf{s}^\top \mathbf{R}_s \mathbf{s} + \text{Tr}(\Sigma \mathbf{V}_{n-1}) + \max_{\mathbf{a}} \left(\mathbf{a}^\top \mathbf{R}_a \mathbf{a} + (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a})^\top \mathbf{V}_{n-1} (\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}) \right). \tag{4.22}$$

We can determine the $\mathbf{a}$ that maximizes the last term in the equation above by computing the derivative with respect to $\mathbf{a}$, setting it to 0, and solving for $\mathbf{a}$:

$$2\mathbf{a}^\top \mathbf{R}_a + 2(\mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a})^\top \mathbf{V}_{n-1} \mathbf{T}_a = 0 \tag{4.23}$$

$$\mathbf{a} = -(\mathbf{T}_a^\top \mathbf{V}_{n-1} \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_a \mathbf{V}_{n-1} \mathbf{T}_s \mathbf{s}. \tag{4.24}$$

Substituting Equation (4.24) into Equation (4.22) and simplifying, we get $U_n(\mathbf{s}) = \mathbf{s}^\top \mathbf{V}_n \mathbf{s} + q_n$ with

$$\mathbf{V}_n = \mathbf{T}_s^\top \mathbf{V}_{n-1} \mathbf{T}_s - \mathbf{T}_s^\top \mathbf{V}_{n-1} \mathbf{T}_a (\mathbf{T}_a^\top \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_s^\top \mathbf{V}_{n-1} \mathbf{T}_s + \mathbf{R}_s \tag{4.25}$$

$$q_n = q_{n-1} + \text{Tr}(\Sigma \mathbf{V}_{n-1}). \tag{4.26}$$

To compute $\mathbf{V}_n$ and q_n for arbitrary n, we first set $\mathbf{V}_0 = 0$ and $q_0 = 0$ and iterate using the equations above. Once we know $\mathbf{V}_{n-1}$ and q_{n-1}, we can extract the optimal n-step policy

$$\pi_n(\mathbf{s}) = -(\mathbf{T}_a^\top \mathbf{V}_{n-1} \mathbf{T}_a + \mathbf{R}_a)^{-1} \mathbf{T}_a \mathbf{V}_{n-1} \mathbf{T}_s \mathbf{s}. \tag{4.27}$$

Interestingly, $\pi_n(\mathbf{s})$ does not depend on the covariance of the noise Σ, although the optimal cost does depend on the noise. A linear system with quadratic cost that has no noise in the dynamics is known in control theory as a *linear quadratic regulator* and has been well studied.

4.5 Approximate Dynamic Programming

The field of *approximate dynamic programming* is concerned with finding approximately optimal policies for problems with large or continuous spaces. Approximate dynamic programming is an active area of research that shares ideas with reinforcement learning. In reinforcement learning, we try to quickly accrue as much reward as possible without a known model. Many of the algorithms for reinforcement learning (discussed in the next chapter) can be applied directly to approximate dynamic programming. This section focuses on several local and global approximation strategies for efficiently finding value functions and policies for known models.

4.5.1 Local Approximation

Local approximation relies on the assumption that states close to each other have similar values. If we know the value associated with a finite set of states $s_{1:n}$, then we can approximate the value of arbitrary states by using the equation

$$U(s) = \sum_{i=1}^{n} \lambda_i \beta_i(s) = \boldsymbol{\lambda}^\top \boldsymbol{\beta}(s), \tag{4.28}$$

where $\beta_{1:n}$ are weighting functions, such that $\sum_{i=1}^{n} \beta_i(s) = 1$. The value λ_i is the value of state s_i. In general, $\beta_i(s)$ should assign greater weight to states that are closer (in some sense) to s_i. A weighting function is often referred to as a *kernel*.

Algorithm 4.4 shows how to compute an approximation of the optimal value function by iteratively updating $\boldsymbol{\lambda}$. The loop is continued until convergence. Once an approximate value function is known, an approximately optimal policy can be extracted as follows:

$$\pi(s) \leftarrow \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \boldsymbol{\lambda}^\top \boldsymbol{\beta}(s') \right). \tag{4.29}$$

Algorithm 4.4 Local approximation value iteration

```
function LocalApproximationValueIteration
    λ ← 0
    loop
        for i ← 1 to n
            u_i ← max_a [R(s_i, a) + γ Σ_{s'} T(s' | s_i, a) λ^⊤ β(s')]
        λ ← u
    return λ
```

A simple approach to local approximation, called *nearest neighbor*, is to assign all weight to the closest discrete state, resulting in a piecewise constant value function. A smoother approximation can be achieved using *k-nearest neighbor*, where a weight of $1/k$ is assigned to each of the k nearest discrete states of s.

If we define a neighborhood function $N(s)$ that returns a set of states from $s_{1:n}$, then we can use *linear interpolation*. If the state space is one dimensional and $N(s) = \{s_1, s_2\}$, then the interpolated value is given by

$$U(s) = \lambda_1 \underbrace{\left(1 - \frac{s - s_1}{s_2 - s_1}\right)}_{\beta_1(s)} + \lambda_2 \underbrace{\left(1 - \frac{s_2 - s}{s_2 - s_1}\right)}_{\beta_2(s)}. \tag{4.30}$$

The equation above can be generalized to d-dimensional state spaces and is often called *bilinear interpolation* in two dimensions and *multilinear interpolation* in arbitrary dimensions.

If the state space has been discretized using a multidimensional grid and the vertices of the grid correspond to the discrete states, then $N(s)$ could be defined to be the set of vertices of the rectangular cell that encloses s. In d dimensions, there may be as many as 2^d neighbors.

Figure 4.9 shows an example grid-based discretization of a two-dimensional state space. To determine the interpolated value of state s (shown as a black circle in the figure), we look at the discrete states in $N(s) = \{s_{12}, s_{13}, s_{17}, s_{18}\}$ (shown as white circles). Using the neighboring values and weights shown in the figure, we compute the value at

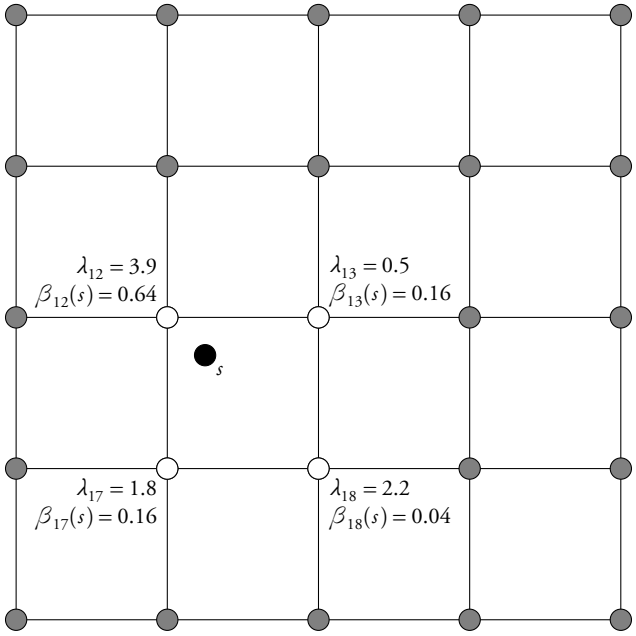
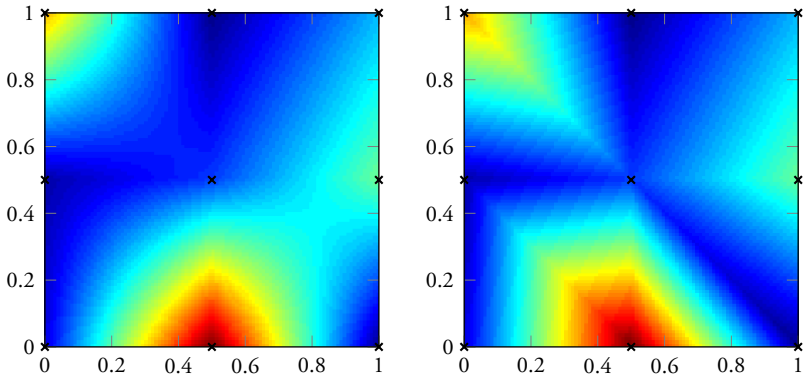


Figure 4.9 Multilinear interpolation in two dimensions.

s as follows:

$$U(s) = \lambda_{12}\beta_{12}(s) + \lambda_{13}\beta_{13}(s) + \lambda_{17}\beta_{17}(s) + \lambda_{18}\beta_{18}(s) \tag{4.31}$$
$$= 3.9 \times 0.64 + 0.5 \times 0.16 + 1.8 \times 0.16 + 2.2 \times 0.04 \tag{4.32}$$
$$= 2.952. \tag{4.33}$$

When the dimensionality of the problem is high, it may be prohibitive to interpolate over the 2^d vertices in the enclosing rectangular cell. An alternative is to use *simplex-based interpolation*. In the simplex method, the rectangular cells are broken into $d!$ multidimensional triangles, called *simplexes*. Instead of interpolating over rectangular cells, we interpolate over a simplex defined by up to $d + 1$ vertices. Hence, interpolating over the simplex scales linearly instead of exponentially with the dimensionality of the state space. However, rectangular interpolation can provide higher quality estimates that can lead to better policies for the same grid resolution. Figure 4.10 shows an example of two-dimensional rectangular and simplex interpolation.



(a) Rectangular interpolation.

(b) Simplex interpolation.

Figure 4.10 Interpolation methods. Data points are indicated by crosses.

4.5.2 Global Approximation

Global approximation uses a fixed set of parameters $\lambda_{1:m}$ to approximate the value function over the entire state space $\mathcal{S}$. One of the most commonly used global approximation methods is based on *linear regression*. We define a set of *basis functions* $\beta_{1:m}$, where $\beta_i : \mathcal{S} \to \mathbb{R}$. Sometimes these basis functions are called *features*. The approximation of $U(s)$ is a linear combination of the parameters and output of the basis functions:

$$U(s) = \sum_{i=1}^{m} \lambda_i \beta_i(s) = \lambda^\top \beta(s). \tag{4.34}$$

The approximation above has the same form as Equation (4.28), but the interpretation is different. The parameters $\lambda_{1:m}$ do not correspond to values at discrete states. The basis functions $\beta_{1:m}$ are not necessarily related to a distance measure, and they need not sum to 1.

Algorithm 4.5 shows how to incorporate linear regression into value iteration. The algorithm is nearly identical to Algorithm 4.4, except for Line 6. Instead of simply assigning $\lambda \leftarrow \mathbf{u}$ as done in local approximation, we call $\lambda_{1:m} \leftarrow \textsc{Regress}(\beta, s_{1:n}, u_{1:n})$. The Regress function finds the λ that leads to the best approximation of the target values $u_{1:n}$ at points $s_{1:n}$ using the basis function β. A common regression objective is to minimize the *sum-squared error*:

$$\sum_{i=1}^{n} \left(\lambda^\top \beta(s_i) - u_i\right)^2. \tag{4.35}$$

Linear least-squares regression can compute the $\boldsymbol{\lambda}$ that minimizes the sum-squared error through simple matrix operations. There are a wide variety of other well-studied regression approaches, both linear and non-linear.

Algorithm 4.5 Linear regression value iteration

```
function LinearRegressionValueIteration
    λ ← 0
    loop
        for i ← 1 to n
            u_i ← max_a[R(s_i, a) + γ Σ_{s'} T(s' | s_i, a) λ^⊤ β(s')]
        λ_{1:m} ← Regress(β, s_{1:n}, u_{1:n})
    return λ
```

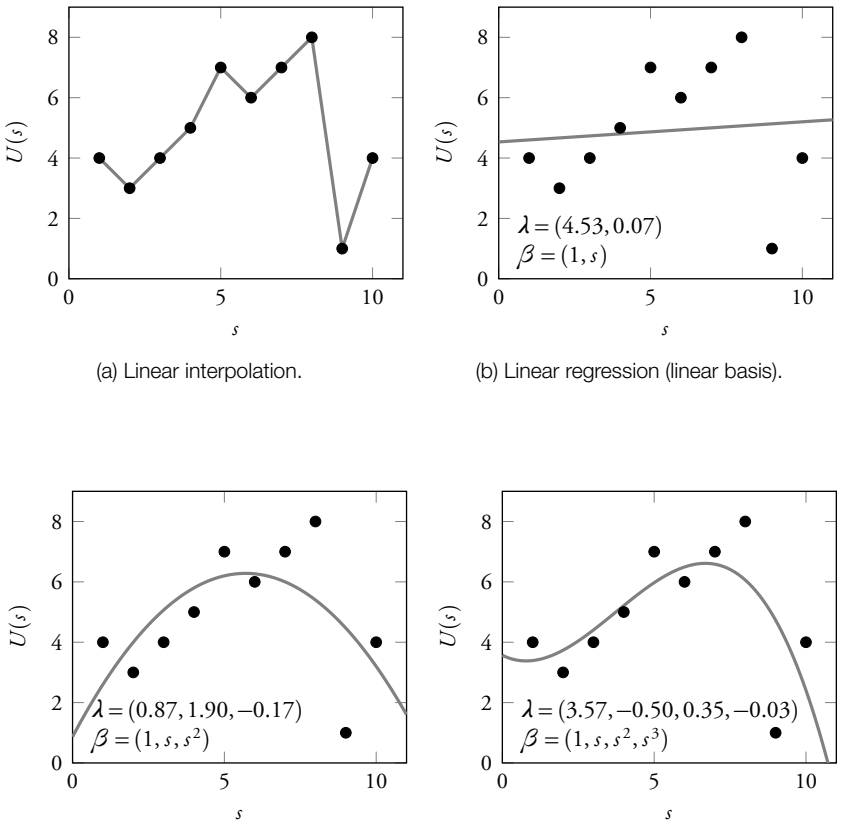
Figure 4.11 compares linear interpolation with linear regression by using different basis functions. For simplicity, the figure assumes a one-dimensional state space, and the states $s_{1:10}$ are evenly spaced. The target values $u_{1:10}$ obtained through dynamic programming are plotted as dots.

Figure 4.11a shows linear interpolation, which produces an approximate value function that matches $u_{1:10}$ exactly at the states $s_{1:10}$. Linear interpolation, of course, requires 10 parameters.

Figure 4.11b shows the result of linear least-squares regression with basis functions $\beta_1(s) = 1$ and $\beta_2(s) = s$. In this case, $\lambda_1 = 4.53$ and $\lambda_2 = 0.07$, meaning that $U(s)$ is approximated as $4.53 + 0.07s$. Although this $\boldsymbol{\lambda}$ minimizes the sum-squared error given those two basis functions, the plot shows that the resulting approximate value function is not especially accurate.

Figure 4.11c shows the result of adding an additional basis function $\beta_3(s) = s^2$. The approximate value function is now quadratic over the state space. Adding this additional basis function results in new values for λ_1 and λ_2. The sum-squared error of the quadratic value function at the states $s_{1:n}$ is much smaller than with the linear function.

Figure 4.11d adds an additional cubic basis function $\beta_4(s) = s^3$, further improving the approximation. All of the basis functions in this example are polynomials, but we could have easily added other basis functions such as $\sin(s)$ and e^s. Adding additional basis functions can generally improve the ability to match the target values at the known states, but too many basis functions can lead to poor approximations at other states. Principled methods exist for choosing an appropriate set of basis functions for regression.



(a) Linear interpolation.

(b) Linear regression (linear basis).

(c) Linear regression (quadratic basis).

(d) Linear regression (cubic basis).

Figure 4.11 Approximations of the value function.

4.6 Online Methods

All the methods presented in this chapter so far involve computing the policy for the entire state space *offline*—that is, prior to execution in the environment. Although factored representations and value function approximation can help scale dynamic programming to higher dimensional state spaces, computing and representing a policy over the full state space can still be intractable. This section discusses *online* methods that restrict computation to states that are reachable from the current state. Because the reachable state space can be orders of magnitude smaller than the full state space, online methods can significantly reduce the amount of storage and computation required to choose optimal (or approximately optimal) actions.

4.6.1 Forward Search

Forward search (Algorithm 4.6) is a simple online action-selection method that looks ahead from some initial state s_0 to some horizon (or depth) d. The forward search function $\textsc{SelectAction}(s, d)$ returns the optimal action a^* and its value v^*. The pseudocode uses $A(s)$ to represent the set of actions available from state s, which may be a subset of the full action space A. The set of possible states that can follow immediately from s after executing action a is denoted $S(s, a)$, which may be a small subset of the full state spaces S.

Algorithm 4.6 Forward search

```
function SelectAction(s, d)
    if d = 0
        return (NIL, 0)
    (a*, v*) ← (NIL, −∞)
    for a ∈ A(s)
        v ← R(s, a)
        for s′ ∈ S(s, a)
            (a′, v′) ← SelectAction(s′, d − 1)
            v ← v + γ T(s′ | s, a) v′
        if v > v*
            (a*, v*) ← (a, v)
    return (a*, v*)
```

Algorithm 4.6 iterates over all possible action and next state pairings and calls itself recursively until the desired depth is reached. The call tree has depth d with a worst-case

branching factor of $|S| \times |A|$ and proceeds depth-first. The computational complexity is $O((|S| \times |A|)^d)$.

4.6.2 Branch and Bound Search

Branch and bound search (Algorithm 4.7) is an extension to forward search that uses knowledge of the upper and lower bounds of the value function to prune portions of the search tree. This algorithm assumes that prior knowledge is available that allows us to easily compute a lower bound on the value function $\underline{U}(s)$ and an upper bound on the state-action value function $\overline{U}(s,a)$. The pseudocode is identical to Algorithm 4.6, except for the use of the lower bound in Line 3 and the pruning check in Line 6. The call to SelectAction(s,d) returns the action to execute and a lower bound on the value function.

The order in which we iterate over the actions in Line 5 is important. In order to prune, the actions must be in descending order of upper bound. In other words, if action a_i is evaluated before a_j, then $\overline{U}(s,a_i) \geq \overline{U}(s,a_j)$. The tighter we are able to make the upper and lower bounds, the more we can prune the search space and decrease computation time. The worst-case computational complexity, however, remains the same as for forward search.

Algorithm 4.7 Branch-and-bound search

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, U̲(s))
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          if Ū(s, a) < v*
7:              return (a*, v*)
8:          v ← R(s, a)
9:          for s′ ∈ S(s, a)
10:             (a′, v′) ← SelectAction(s′, d − 1)
11:             v ← v + γ T(s′ | s, a) v′
12:         if v > v*
13:             (a*, v*) ← (a, v)
14:     return (a*, v*)
```

4.6.3 Sparse Sampling

Sampling methods can be used to avoid the worst-case exponential complexity of forward and branch-and-bound search. Although these methods are not guaranteed to produce the optimal action, they can be shown to produce approximately optimal actions most of the time and can work well in practice. One of the simplest approaches is referred to as *sparse sampling* (Algorithm 4.8).

Sparse sampling uses a generative model G to produce samples of the next state s' and reward r. An advantage of using a generative model is that it is often easier to implement code for drawing random samples from a complex, multidimensional distribution rather than explicitly representing probabilities. Line 8 of the algorithm draws $(s', r) \sim G(s, a)$. All of the information about the state transitions and rewards is represented by G; the state transition probabilities $T(s' \mid s, a)$ and expected reward function $R(s, a)$ are not used directly.

Algorithm 4.8 Sparse sampling

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, 0)
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          v ← 0
7:          for i ← 1 to n
8:              (s', r) ~ G(s, a)
9:              (a', v') ← SelectAction(s', d − 1)
10:             v ← v + (r + γv')/n
11:         if v > v*
12:             (a*, v*) ← (a, v)
13:     return (a*, v*)
```

Sparse sampling is similar to forward search, except that it iterates over n samples instead of all the states in $S(s, a)$. Each iteration results in a sample of $r + \gamma v'$, where r comes from the generative model and v' comes from a recursive call to SelectAction$(s', d - 1)$. These samples of $r + \gamma v'$ are averaged together to estimate $Q(s, a)$. The run time complexity $O((n \times |A|)^d)$ is still exponential in the horizon but does not depend on the size of the state space.

4.6.4 Monte Carlo Tree Search

One of the most successful sampling-based online approaches in recent years is *Monte Carlo tree search*. Algorithm 4.9 is the Upper Confidence Bound for Trees (UCT) implementation of Monte Carlo tree search. In contrast with sparse sampling, the complexity of Monte Carlo tree search does not grow exponentially with the horizon. As in sparse sampling, we use a generative model.

Algorithm 4.9 Monte Carlo tree search

```
function SelectAction(s, d)
    loop
        Simulate(s, d, π_0)
    return arg max_a Q(s, a)
function Simulate(s, d, π_0)
    if d = 0
        return 0
    if s ∉ T
        for a ∈ A(s)
            (N(s, a), Q(s, a)) ← (N_0(s, a), Q_0(s, a))
        T = T ∪ {s}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(s, a) + c sqrt(log N(s) / N(s, a))
    (s', r) ~ G(s, a)
    q ← r + γ Simulate(s', d − 1, π_0)
    N(s, a) ← N(s, a) + 1
    Q(s, a) ← Q(s, a) + (q − Q(s, a)) / N(s, a)
    return q
```

Algorithm 4.10 Rollout evaluation

```
function Rollout(s, d, π_0)
    if d = 0
        return 0
    a ~ π_0(s)
    (s', r) ~ G(s, a)
    return r + γ Rollout(s', d − 1, π_0)
```

The algorithm involves running many simulations from the current state while updating an estimate of the state-action value function $Q(s, a)$. There are three stages in each simulation:

- *Search*. If the current state in the simulation is in the set T (initially empty), then we enter the search stage. Otherwise we proceed to the expansion stage. During the search stage, we update $Q(s, a)$ for the states and actions visited and tried in our search. We also keep track of the number of times we have taken an action from a state $N(s, a)$. During the search, we execute the action that maximizes

$$Q(s, a) + c\sqrt{\frac{\log N(s)}{N(s, a)}}, \tag{4.36}$$

  where $N(s) = \sum_a N(s, a)$ and c is a parameter that controls the amount of exploration in the search (exploration will be covered in depth in the next chapter). The second term is an *exploration bonus* that encourages selecting actions that have not been tried as frequently.
- *Expansion*. Once we have reached a state that is not in the set T, we iterate over all of the actions available from that state and initialize $N(s, a)$ and $Q(s, a)$ with $N_0(s, a)$ and $Q_0(s, a)$, respectively. The functions N_0 and Q_0 can be based on prior expert knowledge of the problem; if none is available, then they can both be initialized to 0. We then add the current state to the set T.
- *Rollout*. After the expansion stage, we simply select actions according to some *rollout* (or default) policy π_0 until the desired depth is reached (Algorithm 4.10). Typically, rollout policies are stochastic, and so the action to execute is sampled $a \sim \pi_0(s)$. The rollout policy does not have to be close to optimal, but it is a way for an expert to bias the search into areas that are promising. The expected value is returned and used in the search to update the value for $Q(s, a)$.

Simulations are run until some stopping criterion is met, often simply a fixed number of iterations. We then execute the action that maximizes $Q(s, a)$. Once that action has been executed, we can rerun the Monte Carlo tree search to select the next action. It is common to carry over the values of $N(s, a)$ and $Q(s, a)$ computed in the previous step.

4.7 Direct Policy Search

The previous sections have presented methods that involve computing or approximating the value function. An alternative is to search the space of policies directly. Although the state space may be high dimensional, making approximation of the value function difficult, the space of possible policies may be relatively low dimensional and can be easier to search directly.

4.7.1 Objective Function

Suppose we have a policy that is parametrized by λ. The probability that the policy selects action a given state s is written $\pi_\lambda(a \mid s)$. Given an initial state s, we can estimate

$$U^{\pi_\lambda}(s) \approx \frac{1}{n} \sum_{i=1}^{n} u_i, \tag{4.37}$$

where u_i is the ith rollout of the policy π_λ to some depth.

The objective in direct policy search is to find the parameter λ that maximizes the function

$$V(\lambda) = \sum_s b(s) U^{\pi_\lambda}(s), \tag{4.38}$$

where $b(s)$ is a distribution over the initial state. We can estimate $V(\lambda)$ using Monte Carlo simulation and a generative model G up to depth d as outlined in Algorithm 4.11.

Algorithm 4.11 Monte Carlo policy evaluation

```
function MonteCarloPolicyEvaluation(λ, d)
    for i ← 1 to n
        s ~ b
        u_i ← Rollout(s, d, π_λ)
    return (1/n) Σ_{i=1}^{n} u_i
```

The function $V(\lambda)$, as estimated by Algorithm 4.11, is a *stochastic function*; given the same input λ, it may give different outputs. As the number of samples (determined by n and m) increases, the variability of the outputs of the function decreases. Many different methods exist for searching the space of policy parameters that maximizes $V(\lambda)$, and we will discuss a few of them.

4.7.2 Local Search Methods

A common stochastic optimization approach is *local search*, also known as *hill climbing* or *gradient ascent*. Local search begins at a single point in the search space and then incrementally moves from neighbor to neighbor in the search space until convergence. The search operates with the assumption that the value of the stochastic function at a point in the search space is an indication of how close that point is to the global optimum. Therefore, local search generally selects the neighbor with the largest value.

Some local search techniques directly estimate the gradient $\nabla_\lambda V$ for a particular policy and then step some amount in the direction of steepest ascent. For some policy representations, it is possible to analytically derive the gradient. Other local search techniques evaluate a finite sampling of the neighborhood of the current search point and then move to the neighbor with the greatest value. Local search is susceptible to local optima and plateaus in $V(\lambda)$. Simulated annealing or some of the other methods suggested at the end of Section 2.4.2 can be applied to help find a global optimum.

4.7.3 Cross Entropy Methods

There is another class of policy search methods that maintain a distribution over policies and updates the distribution based on policies that perform well. One approach to updating this distribution is to use the *cross entropy* method. Cross entropy is a concept from information theory and is a measure of the difference between two distributions. If distributions p and q are discrete, then the cross entropy is given by

$$H(p, q) = -\sum_x p(x) \log q(x). \tag{4.39}$$

For continuous distributions, the summation is replaced with an integral. In the context of direct policy search, we are interested in distributions over λ. These distributions are parameterized by θ, which may be multivariate.

The cross entropy method takes as input an initial θ and parameters n and m that determine the number of samples to use. The process consists of two stages that are repeated until convergence or some other stopping criterion is met:

- *Sample.* We draw n samples from $P(\lambda; \theta)$ and evaluate their performance using Algorithm 4.11. Sort the samples in decreasing order of performance so that $i < j$ implies that $V(\lambda_i) \geq V(\lambda_j)$.
- *Update.* Use the top m performing samples (often called the *elite* samples) to update θ using cross entropy minimization, which boils down to:

$$\theta \leftarrow \arg\max_\theta \sum_{j=1}^{m} \log P(\lambda_j \mid \theta). \tag{4.40}$$

  The new θ happens to correspond to the maximum likelihood estimate based on the m top-performing samples. Further explanation can be found in the references at the end of the chapter.

Algorithm 4.12 Cross entropy policy search

```
function CrossEntropyPolicySearch(θ)
    repeat
        for i ← 1 to n
            λ_i ~ P(·; θ)
            v_i ← MonteCarloPolicyEvaluation(λ_i)
        Sort (λ_1, ... λ_n) in decreasing order of v_i
        θ ← arg max_θ Σ_{j=1}^{m} log P(λ_j | θ)
    until convergence
    return λ ← arg max P(λ | θ)
```

The full process is outlined in Algorithm 4.12. The initial distribution parameter θ, the number of samples n, and the number of elite samples m are the input parameters to this process. To prevent the search from focusing too much on local maxima, the initial θ should provide a diffuse distribution over λ. The choice of the number of samples and elite samples depends on the problem.

To illustrate the cross entropy method, we will assume that the space of policies is one dimensional and that $V(\lambda)$ is as shown in Figure 4.12. We will assume for this example that the parameter $\theta = (\mu, \sigma)$ and $P(\lambda \mid \theta) = \mathcal{N}(\lambda \mid \mu, \sigma^2)$. Initially, we set $\theta = (0, 10)$. To not overwhelm the plots, we use only $n = 20$ samples and $m = 5$ elite samples. Typically, especially for higher dimensional problems, we use one to two orders of magnitude more samples.

Figure 4.12a shows the initial distribution over λ. We draw 20 samples from this distribution. The 5 elite samples are shown with circles, and the 15 other samples are shown with crosses. Those 5 elite samples are used to update θ. Because we are using a Gaussian distribution, the update simply sets the mean to the mean of the elite samples and the standard deviation to the sample standard deviation of the elite samples. The updated distribution is shown in Figure 4.12b. The process is repeated. In the third iteration (Figure 4.12c), the distribution is moved toward the more promising area of the search space. By the fourth iteration (Figure 4.12d), we have found the global optimum.

4.7.4 Evolutionary Methods

Evolutionary search methods derive inspiration from biological evolution. A common approach is to use a *genetic algorithm* that evolves populations of (typically binary) strings representing policies, starting with an initial random population. The strings recombine through genetic crossover and mutation at a rate proportional to their measured fitness

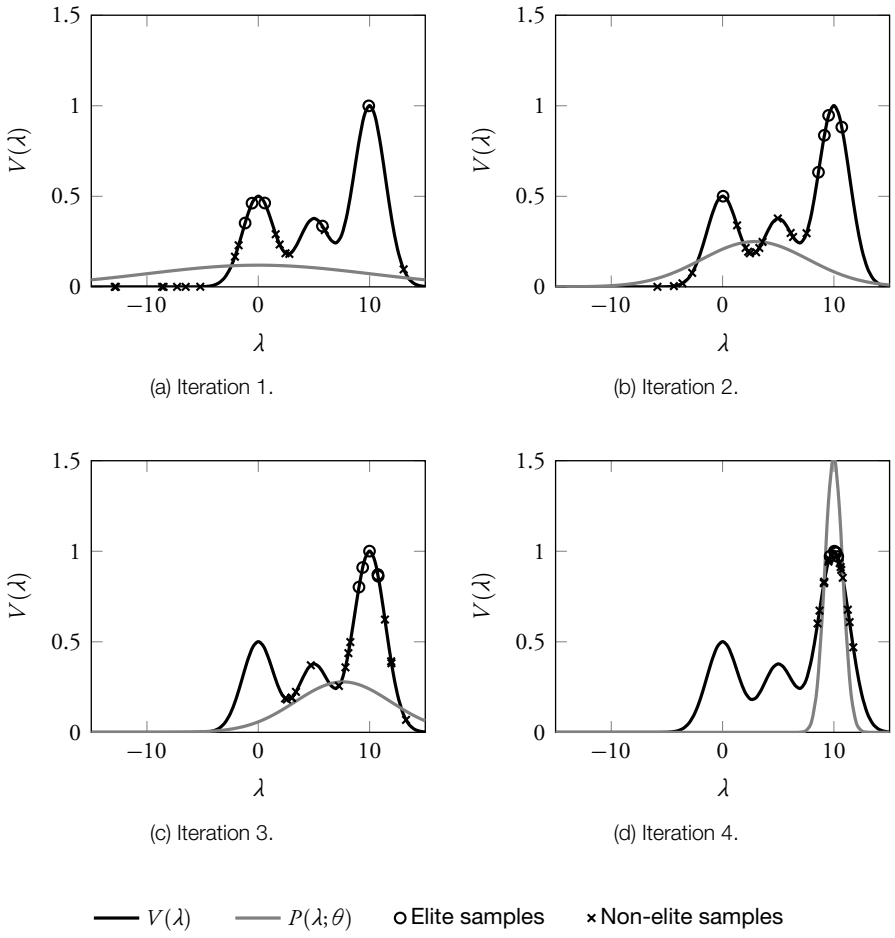


Figure 4.12 Cross entropy method iterations.

to produce a new generation. The process continues until arriving at a satisfactory solution.

A related approach is *genetic programming*, which involves evolving tree structures representing policies. Trees consist of symbols selected from predefined sets of terminals and non-terminals, allowing more flexible policy representations than fixed-length bit strings. Crossover works by swapping subtrees, and mutation works by randomly modifying subtrees.

Genetic algorithms and genetic programming may be combined with other methods, including local search. For example, a genetic algorithm might evolve a satisfactory policy and then use local search to further improve the policy. Such an approach is called *genetic local search* or *memetic algorithms*.

4.8 Summary

- Markov decision processes represent sequential decision-making problems using a transition and reward function.
- Optimal policies can be found using dynamic programming algorithms.
- Continuous problems with linear Gaussian dynamics and quadratic costs can be solved analytically.
- Structured dynamic programming can efficiently solve factored Markov decision processes.
- Problems with large or continuous state spaces can be solved approximately using function approximation.
- Instead of solving for the optimal strategy for the full state space offline, online methods search for the optimal action from the current state.
- In some problems, it can be easier to search the space of policies directly using stochastic optimization methods.

4.9 Further Reading

Much of the pioneering work on sequential decision problems was begun in 1949 by Richard Bellman [1]. Markov decision processes have since become the standard framework for modeling such problems, and there are several books on the subject [2]–[5]. The example grid world problem in Section 4.2.5 comes from *Artificial Intelligence: Foundations of Computational Agents* by Poole and Mackworth [6]. The website associated with the book contains an open-source software demonstration of the grid world example.

Boutilier, Dearden, and Goldszmidt present structured value iteration and policy iteration algorithms for factored MDPs using decision trees [7]. As mentioned in Section 4.3.2, it can be more efficient to use decision diagrams instead of trees [7]–[9].

Approximate linear programming approaches to factored MDPs have been explored by Guestrin et al. [10].

Optimal control in linear systems with quadratic costs has been well studied in the control theory community, and there are many books on the subject [11]–[13]. Section 4.4 presented a special case of the linear-quadratic-Gaussian (LQG) control problem in which the state of the system is known perfectly. Chapter 6 will present the more traditional version of LQG with imperfect state information.

An overview of the field of approximate dynamic programming is provided in *Approximate Dynamic Programming: Solving the Curses of Dimensionality* by Powell [14]. The book *Reinforcement Learning and Dynamic Programming Using Function Approximators* by Busoniu et al. outlines approximation methods and provides source code for cases where the model is known and unknown [15]. Solving problems in which the model is unknown is called reinforcement learning and is discussed in the next chapter. Reinforcement learning is often used in problems with a known model that is too complex or high dimensional to apply exact dynamic programming.

As discussed in Section 4.6, online methods are often appropriate when the state space is high dimensional and adequate computational resources are available to perform planning during execution. Land and Doig originally proposed the branch and bound method for discrete programming problems [16]. This method has been applied to a wide variety of optimization problems. Sparse sampling was developed by Kearns, Mansour, and Ng [17]. Other methods that can be used online include real-time dynamic programming [18] and LAO* [19].

Kocsis and Szepesvári originally introduced the idea of Monte Carlo tree search with the use of the exploration bonus in Algorithm 4.9 [20]. Since that paper's publication and with the successful application to the game Go, a tremendous amount of work has focused on Monte Carlo tree search methods [21]. One important extension to the algorithm presented in this chapter is the idea of progressive widening of the actions and states considered at each step in the search [22]. Progressive widening allows the algorithm to better handle large or continuous state or action spaces.

Many methods have been proposed for searching the space of policies directly. Examples of local search algorithms using gradient methods include those of Williams [23] and Baxter and Bartlett [24]. The cross entropy method [25], [26] has been applied to a variety of formulations of MDP policy search [27]–[29]. Any stochastic optimization technique can be applied to policy search. The evolutionary methods discussed in Section 4.7.4 date back to the 1950s [30]. Genetic algorithms, in particular, were made popular by the work of Holland [31], with more recent theoretical work done by Schmitt [32], [33]. Genetic programming was introduced by Koza [34].

References

1. S. Dreyfus, "Richard Bellman on the Birth of Dynamic Programming," *Operations Research*, vol. 50, no. 1, pp. 48–51, 2002.
2. R.E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.
3. D. Bertsekas, *Dynamic Programming and Optimal Control*. Belmont, MA: Athena Scientific, 2007.
4. M.L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Hoboken, NJ: Wiley, 2005.
5. O. Sigaud and O. Buffet, eds., *Markov Decision Processes in Artificial Intelligence*. New York: Wiley, 2010.
6. D.L. Poole and A.K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. New York: Cambridge University Press, 2010.
7. C. Boutilier, R. Dearden, and M. Goldszmidt, "Stochastic Dynamic Programming with Factored Representations," *Artificial Intelligence*, vol. 121, no. 1-2, pp. 49–107, 2000. DOI: 10.1016/S0004-3702(00)00033-3.
8. J. Hoey, R. St-Aubin, A.J. Hu, and C. Boutilier, "SPUDD: Stochastic Planning Using Decision Diagrams," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.
9. R. St-Aubin, J. Hoey, and C. Boutilier, "APRICODD: Approximate Policy Construction Using Decision Diagrams," in *Advances in Neural Information Processing Systems (NIPS)*, 2000.
10. C. Guestrin, D. Koller, R. Parr, and S. Venkataraman, "Efficient Solution Algorithms for Factored MDPs," *Journal of Artificial Intelligence Research*, vol. 19, pp. 399–468, 2003. DOI: 10.1613/jair.1000.
11. F.L. Lewis, D.L. Vrabie, and V.L. Syrmos, *Optimal Control*, 3rd ed. Hoboken, NJ: Wiley, 2012.
12. R.F. Stengel, *Optimal Control and Estimation*. New York: Dover Publications, 1994.
13. D.E. Kirk, *Optimal Control Theory: An Introduction*. Englewood Cliffs, NJ: Prentice-Hall, 1970.
14. W.B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, 2nd ed. Hoboken, NJ: Wiley, 2011.
15. L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, FL: CRC Press, 2010.

16. A.H. Land and A.G. Doig, "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960. DOI: 10.2307/1910129.

17. M.J. Kearns, Y. Mansour, and A.Y. Ng, "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes," *Machine Learning*, vol. 49, no. 2-3, pp. 193–208, 2002. DOI: 10.1023/A:1017932429737.

18. A.G. Barto, S.J. Bradtke, and S.P. Singh, "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, vol. 72, no. 1-2, pp. 81–138, 1995. DOI: 10.1016/0004-3702(94)00011-O.

19. E.A. Hansen and S. Zilberstein, "LAO*: A Heuristic Search Algorithm That Finds Solutions with Loops," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 35–62, 2001. DOI: 10.1016/S0004-3702(01)00106-0.

20. L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *European Conference on Machine Learning (ECML)*, 2006.

21. C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. DOI: 10.1109/TCIAIG.2012.2186810.

22. A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous Upper Confidence Trees," in *Learning and Intelligent Optimization (LION)*, 2011.

23. R.J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, pp. 229–256, 1992. DOI: 10.1007/BF00992696.

24. J. Baxter and P.L. Bartlett, "Infinite-Horizon Policy-Gradient Estimation," *Journal of Artificial Intelligence Research*, vol. 15, pp. 319–350, 2001. DOI: 10.1613/jair.806.

25. P.-T. de Boer, D.P. Kroese, S. Mannor, and R.Y. Rubinstein, "A Tutorial on the Cross-Entropy Method," *Annals of Operations Research*, vol. 134, no. 1, pp. 19–67, 2005. DOI: 10.1007/s10479-005-5724-z.

26. R.Y. Rubinstein and D.P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. New York: Springer, 2004.

27. S. Mannor, R.Y. Rubinstein, and Y. Gat, "The Cross Entropy Method for Fast Policy Search," in *International Conference on Machine Learning (ICML)*, 2003.

28. I. Szita and A. Lörincz, "Learning Tetris Using the Noisy Cross-Entropy Method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006. DOI: 10.1162/neco.2006.18.12.2936.

29. ——, "Learning to Play Using Low-Complexity Rule-Based Policies: Illustrations Through Ms. Pac-Man," *Journal of Artificial Intelligence Research*, vol. 30, pp. 659–684, 2007. DOI: 10.1613/jair.2368.

30. D.B. Fogel, ed., *Evolutionary Computation: The Fossil Record*. New York: Wiley-IEEE Press, 1998.

31. J.H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.

32. L.M. Schmitt, "Theory of Genetic Algorithms," *Theoretical Computer Science*, vol. 259, no. 1-2, pp. 1–61, 2001. DOI: 10.1016/S0304-3975(00)00406-0.

33. ——, "Theory of Genetic Algorithms II: Models for Genetic Operators over the String-Tensor Representation of Populations and Convergence to Global Optima for Arbitrary Fitness Function Under Scaling," *Theoretical Computer Science*, vol. 310, no. 1-3, pp. 181–231, 2004. DOI: 10.1016/S0304-3975(03)00393-1.

34. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

5

Model Uncertainty

Mykel J. Kochenderfer

The previous chapter discussed sequential decision problems with a known transition and reward model. In many problems, the dynamics and rewards are not known exactly, and the agent must learn to act through experience. By observing the outcomes of its actions in the form of state transitions and rewards, the agent is to choose actions that maximize its long-term accumulation of rewards. Solving such problems in which there is model uncertainty is the subject of the field of *reinforcement learning* and the focus of this chapter. Several challenges in addressing model uncertainty will be discussed. First, the agent must carefully balance exploration of the environment with the exploitation of that knowledge gained through experience. Second, rewards may be received long after the important decisions have been made, so credit for later rewards must be assigned to earlier decisions. Third, the agent must generalize from limited experience. This chapter will review the theory and some of the key algorithms for addressing these challenges.

5.1 Exploration and Exploitation

Reinforcement learning problems require us to carefully balance *exploration* of the environment with *exploitation* of knowledge obtained through evaluative feedback. If we continuously explore our environment, then we may be able to build a comprehensive model, but we will not be able to accumulate much reward. If we continuously make the decision we believe is best without ever trying a new strategy, then we may miss out on improving our strategy and accumulating more reward. This section introduces the challenges of balancing exploration with exploitation on problems with a single state.

5.1.1 Multi-Armed Bandit Problems

Some of the earliest studies of balancing exploration with exploitation were focused on slot machines—sometimes referred to as *one-armed bandits* because they are often controlled by a single lever and, on average, they take away gamblers' money. Bandit

problems appear in a wide variety of applications, such as the allocation of clinical trials and adaptive network routing. They were originally formulated during World War II and proved exceptionally challenging to solve. According to Peter Whittle, "efforts to solve [bandit problems] so sapped the energies and minds of Allied analysts that the suggestion was made that the problem be dropped over Germany as the ultimate instrument of intellectual sabotage" (see comment following article [1]).

Many different formulations of bandit problems are presented in the literature, but we will focus on a simple one involving a slot machine with n arms. Arm i pays off 1 with probability θ_i and 0 with probability $1-\theta_i$. There is no deposit to play, but we are limited to h pulls. We can view this problem as an h-step finite horizon Markov decision process (Chapter 4) with a single state, n actions, and an unknown reward function $R(s,a)$.

5.1.2 Bayesian Model Estimation

We can use the beta distribution introduced in Section 2.3.2 to represent our posterior over the win probability θ_i for arm i, and we will use the uniform prior distribution, which corresponds to Beta$(1,1)$. We just have to keep track of the number of wins w_i and the number of losses ℓ_i for each arm i. The posterior for θ_i is given by Beta(w_i+1, ℓ_i+1). We can then compute the posterior probability of winning:

$$\rho_i = P(\text{win}_i \mid w_i, \ell_i) = \int_0^1 \theta \times \text{Beta}(\theta \mid w_i+1, \ell_i+1)\, d\theta = \frac{w_i+1}{w_i+\ell_i+2}. \quad (5.1)$$

For example, suppose we have a two-armed bandit that we have pulled six times. The first arm had 1 win and 0 losses, and the second arm has 4 wins and 1 loss. Assuming a uniform prior, the posterior distribution for θ_1 is given by Beta$(2,1)$, and the posterior distribution for θ_2 is given by Beta$(5,2)$. The posteriors are plotted in Figure 5.1.

The maximum likelihood estimate for θ_1 is 1 and the maximum likelihood estimate for θ_2 is $4/5$. If we were to choose the next pull solely on the basis of the maximum-likelihood estimate, then we would want to go with the first arm—with a guaranteed win! Of course, just because we have not yet observed a loss with the first arm does not mean that a loss is impossible.

In contrast with the maximum likelihood estimate of the payoff probabilities, the Bayesian posterior shown in Figure 5.1 assigns non-zero probability to probabilities between 0 and 1. The density at 0 for both arms is 0 because at least one win was observed from both arms. There is also zero density at $\theta_2 = 1$ because a loss was observed. Using Equation (5.1), we can compute the payoff probabilities:

$$\rho_1 = 2/3 = 0.67 \quad (5.2)$$

$$\rho_2 = 5/7 = 0.71. \quad (5.3)$$

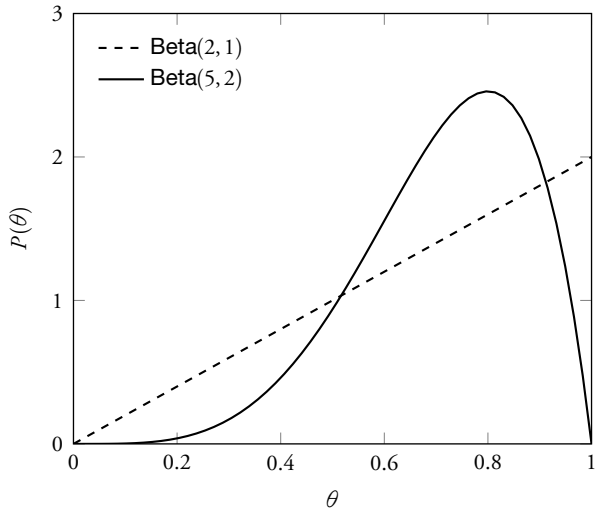


Figure 5.1 Posterior distribution of payoff probabilities.

Hence, if we assume that we only have one pull remaining, it is best to pull the second arm.

5.1.3 Ad Hoc Exploration Strategies

Several different ad hoc exploration strategies have been suggested in the literature. In one of the most common strategies, ϵ-greedy, we choose a random arm with probability ϵ; otherwise, we choose $\arg\max_i \rho_i$. Larger values of ϵ allow us to more quickly identify the best arm, but more pulls are wasted on suboptimal arms.

Directed exploration strategies involve using information gathered from previous pulls. For example, the *softmax* strategy is to pull arms according to the logit model (introduced in Section 3.3.3), where arm i is selected with probability proportional to $\exp(\lambda\rho_i)$. The precision parameter $\lambda \geq 0$ controls the amount of exploration, with uniform random selection as $\lambda \to 0$ and greedy selection as $\lambda \to \infty$. Another approach is to use *interval exploration*, by which we compute the α% confidence interval for θ_i and choose the arm with the highest upper bound. Larger values for α result in more exploration.

5.1.4 Optimal Exploration Strategies

The counts $w_1, \ell_1, \ldots, w_n, \ell_n$ represent a *belief state*, which summarizes our belief about payoffs. As discussed in Section 5.1.2, these $2n$ numbers can be used to represent n

continuous probability distributions over possible values for $\rho_{1:n}$. These belief states can be used as states in an MDP that represents the n-armed bandit problem. We can use dynamic programming to determine an optimal policy π^*, which specifies which arm to pull given the counts.

We use $Q^*(w_{1:n}, \ell_{1:n}, i)$ to represent the expected payoff after pulling arm i and then acting optimally. The optimal utility function and policy are given in terms of Q^*:

$$U^*(w_1, \ell_1, \ldots, w_n, \ell_n) = \max_i Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, i) \tag{5.4}$$

$$\pi^*(w_1, \ell_1, \ldots, w_n, \ell_n) = \arg\max_i Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, i). \tag{5.5}$$

We can decompose Q^* into two terms:

$$Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, i) = \frac{w_i + 1}{w_i + \ell_i + 2}\left(1 + U^*(\ldots, w_i + 1, \ell_i, \ldots)\right) + \left(1 - \frac{w_i + 1}{w_i + \ell_i + 2}\right)U^*(\ldots, w_i, \ell_i + 1, \ldots). \tag{5.6}$$

The first term is associated with a win for arm i, and the second term is associated with a loss. The value $(w_i + 1)/(w_i + \ell_i + 2)$ is the posterior probability of a win, which comes from Equation (5.1). The first U^* in the equation above assumes that pulling arm i brought a win, and the second U^* assumes a loss.

Assuming a horizon h, we can compute Q^* for the entire belief space. We start with the belief states with $\sum_i (w_i + \ell_i) = h$. With no pulls left, $U^*(w_1, \ell_1, \ldots, w_n, \ell_n) = 0$. We can then work backward to the states where $\sum_i (w_i + \ell_i) = h - 1$ and apply Equation (5.6).

Although this dynamic programming solution is optimal, the number of belief states—and consequently the amount of computation and memory required—is exponential in h. We can formulate an infinite horizon, discounted version of the problem that can be solved efficiently using the *Gittins allocation index*. The allocation index can be stored as a lookup table that specifies a scalar allocation index value given the number of pulls and the number of wins associated with an arm. The arm that has the highest allocation index is the one that should be pulled next.

5.2 Maximum Likelihood Model-Based Methods

A variety of reinforcement learning methods have been proposed for addressing problems with multiple states. Solving problems with multiple states is more challenging than bandit problems because we need to plan to visit states to determine their value. One approach to reinforcement learning involves estimating the transition and reward models directly from experience. We keep track of the counts of transitions $N(s, a, s')$ and the

sum of rewards $\rho(s,a)$. The maximum likelihood estimates of the transition and reward models are as follows:

$$N(s,a) = \sum_{s'} N(s,a,s') \tag{5.7}$$

$$T(s' \mid s,a) = N(s,a,s')/N(s,a) \tag{5.8}$$

$$R(s,a) = \rho(s,a)/N(s,a). \tag{5.9}$$

If we have prior knowledge about the transition probabilities or the rewards, then we can initialize $N(s,a,s')$ and $\rho(s,a)$ to values other than 0.

We can solve the MDP assuming the estimated models are correct. Of course, we have to incorporate some exploration strategy, such as those mentioned in Section 5.1.3, in order to ensure that we converge to an optimal strategy. The basic structure of maximum likelihood model-based reinforcement learning is outlined in Algorithm 5.1.

Algorithm 5.1 Maximum likelihood model-based reinforcement learning

```
1:  function MaximumLikelihoodModelBasedReinforcementLearning
2:      t ← 0
3:      s_0 ← initial state
4:      Initialize N, ρ, and Q
5:      loop
6:          Choose action a_t based on some exploration strategy
7:          Observe new state s_{t+1} and reward r_t
8:          N(s_t, a_t, s_{t+1}) ← N(s_t, a_t, s_{t+1}) + 1
9:          ρ(s_t, a_t) ← ρ(s_t, a_t) + r_t
10:         Update Q based on revised estimate of T and R
11:         t ← t + 1
```

5.2.1 Randomized Updates

Although we can use any dynamic programming algorithm to update Q in Line 10 of Algorithm 5.1, the computational expense is often not necessary. One algorithm that avoids solving the entire MDP at each time step is *Dyna*. Dyna performs the following update at the current state:

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} Q(s',a'). \tag{5.10}$$

Here, R and T are the estimated reward and transition functions. We then perform some number of additional updates of Q for random states and actions depending on how much time is available between decisions. Following the updates, we use Q to choose which action to execute—perhaps using softmax or one of the other exploration strategies.

5.2.2 Prioritized Updates

An approach known as *prioritized sweeping* uses a priority queue to help identify which states require updating Q the most (Algorithm 5.2). If we transition from s to s', then we update $U(s)$ based on our updated transition and reward models. We then iterate over the predecessor set, $\text{pred}(s) = \{(s', a') \mid T(s \mid s', a') > 0\}$, the set of all state-action pairs leading immediately to state s. The priority of s' is increased to $T(s \mid s', a') \times |U(s) - u|$, where u was the value of $U(s)$ prior to the update. Hence, the larger the change in $U(s)$, the higher the priority of the states leading to s. The process of updating the highest priority state in the queue continues for some fixed number of iterations or until the queue becomes empty.

Algorithm 5.2 Prioritized sweeping

```
function PrioritizedSweeping(s)
    Increase the priority of s to ∞
    while priority queue is not empty
        s ← highest priority state
        Update(s)
function Update(s)
    u ← U(s)
    U(s) ← max_a [R(s,a) + γ Σ_{s'} T(s' | s,a) U(s')]
    for (s', a') ∈ pred(s)
        p ← T(s | s', a') × |U(s) − u|
        Increase priority of s' to p
```

5.3 Bayesian Model-Based Methods

The previous section used maximum likelihood estimates of the transition probabilities and rewards and then relied on a heuristic exploration strategy to converge on an optimal strategy in the limit. Bayesian methods, in contrast, allow us to optimally balance exploration with exploitation without having to rely on heuristics. This section describes a generalization of the formulation for multi-armed bandit problems (discussed in Section 5.1.4) applied to general MDPs.

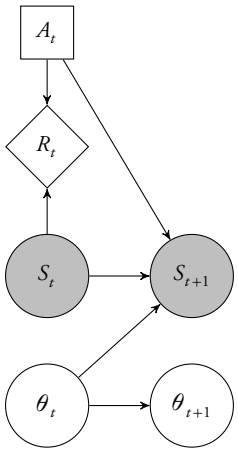


Figure 5.2 Markov decision process with model uncertainty.

5.3.1 Problem Structure

In Bayesian reinforcement learning, we specify a prior distribution over all the model parameters θ. These model parameters may include the parameters governing the distribution over immediate rewards, but we will focus on the parameters governing the state transition probabilities. If $\mathcal{S}$ represents the state space and $\mathcal{A}$ represents the action space, then the parameter vector θ consists of $|\mathcal{S}|^2|\mathcal{A}|$ components representing every possible transition probability. The component of θ that governs the transition probability $T(s' \mid s, a)$ is denoted $\theta_{(s,a,s')}$.

The structure of the problem can be represented using the dynamic decision network shown in Figure 5.2, which is an extension of the network shown in Figure 4.1b with the model parameters made explicit. As indicated by the shaded nodes, the states are observed, but the model parameters are not. We generally assume that the model parameters are time invariant, and so $\theta_{t+1} = \theta_t$. However, our belief about θ evolves with time as we transition to new states.

5.3.2 Beliefs over Model Parameters

We want to represent a prior belief over θ, and a natural way to do this for discrete state spaces is with a product of Dirichlet distributions. Each Dirichlet would represent a distribution over the next state given the current state s and action a. If $\theta_{(s,a)}$ is an $|\mathcal{S}|$-element vector representing the distribution over the next state, then the prior distribution is given by

$$\text{Dir}(\theta_{(s,a)} \mid \alpha_{(s,a)}). \tag{5.11}$$

The Dirichlet distribution above is governed by the $|\mathcal{S}|$ parameters in $\boldsymbol{\alpha}_{(s,a)}$. It is common to use a uniform prior with all the components of $\boldsymbol{\alpha}_{(s,a)}$ set to 1, but if we have prior knowledge about the dynamics, then we can set these parameters differently, as discussed in Section 2.3.2.

The prior distribution over θ is given by the product

$$b_0(\theta) = \prod_s \prod_a \text{Dir}(\theta_{(s,a)} \mid \boldsymbol{\alpha}_{(s,a)}). \tag{5.12}$$

The factorization shown above is often used for small discrete state spaces, but other lower dimensional parametric representations may be desirable.

The posterior distribution over θ after t steps is denoted b_t. Suppose that during the first t steps, we observe $m_{(s,a,s')}$ transitions from s to s' by action a. We compute the posterior using Bayes' rule. If $\mathbf{m}_{(s,a)}$ represents a vector of transition counts, then the posterior is given by

$$b_t(\theta) = \prod_s \prod_a \text{Dir}(\theta_{(s,a)} \mid \boldsymbol{\alpha}_{(s,a)} + \mathbf{m}_{(s,a)}). \tag{5.13}$$

5.3.3 Bayes-Adaptive Markov Decision Processes

We can formulate the problem of acting optimally in an MDP with an *unknown* model as a higher dimensional MDP with a *known* model. This higher dimensional MDP is known as a *Bayes-adaptive Markov decision process*, which is related to the partially observable Markov decision process discussed in the next chapter.

The state space in a Bayes-adaptive MDP is the Cartesian product $\mathcal{S} \times \mathcal{B}$, where $\mathcal{B}$ is the space of all possible beliefs over the model parameters θ. Although $\mathcal{S}$ is discrete, $\mathcal{B}$ is often a high-dimensional continuous space. A state in a Bayes-adaptive MDP is written as a pair (s, b) consisting of the state s of the base MDP and the belief state b. The action space and reward function are exactly the same as for the base MDP.

The transition function in a Bayes-adaptive MDP is $T(s', b' \mid s, b, a)$, which is the probability of transitioning to some new state s' with a new belief state b', given that you start in state s with belief b and execute action a. The new belief state b' is a deterministic function of s, b, a, and s' as computed by Bayes' rule in Section 5.3.2. Let us denote this deterministic function τ so that $b' = \tau(s, b, a, s')$. The Bayes-adaptive MDP transition function can be decomposed as follows:

$$T(s', b' \mid s, b, a) = \delta_{\tau(s,b,a,s')}(b') P(s' \mid s, b, a), \tag{5.14}$$

where $\delta_x(y)$ is the *Kronecker delta* function such that

$$\delta_x(y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}. \tag{5.15}$$

Computing $P(s' \mid s, b, a)$ requires integration:

$$P(s' \mid s, b, a) = \int_\theta b(\theta) P(s' \mid s, \theta, a)\, d\theta = \int_\theta b(\theta) \theta_{(s,a,s')}\, d\theta. \tag{5.16}$$

Similar to Equation (5.1), the integral above can be evaluated analytically.

5.3.4 Solution Methods

We can generalize the Bellman equation from Section 4.2.4 for MDPs with a known model to the case in which the model is unknown:

$$U^*(s, b) = \max_a \left(R(s, a) + \gamma \sum_{s'} P(s' \mid s, b, a) U^*(s', \tau(s, b, a, s')) \right). \tag{5.17}$$

Unfortunately, we cannot simply use the policy iteration and value iteration algorithms directly as presented in Chapter 4 because b is continuous. However, we can use the approximations in Section 4.5 as well as the online methods in Section 4.6. Methods that better leverage the structure of the Bayes-adaptive MDP will be presented in the next chapter.

An alternative to solving for the optimal value function over the belief space is to use a technique known as *Thompson sampling*. The idea here is to draw a sample θ from the current belief b_t and then assume θ is the true model. We use dynamic programming to solve for the best action. At the next time step, we update our belief, draw a new sample, and resolve the MDP. The advantage of this approach is that we do not have to decide on heuristic exploration parameters. However, Thompson sampling has been shown to over-explore, and resolving the MDP at every step can be expensive.

5.4 Model-Free Methods

In contrast to model-based methods, model-free reinforcement learning does not require building explicit representations of the transition and reward models. Avoiding explicit representations is attractive, especially when the problem is high dimensional.

5.4.1 Incremental Estimation

Many model-free methods involve incremental estimation of the expected discounted return from the various states in the problem. Suppose we have a random variable X and want to estimate the mean from a set of samples $x_{1:n}$. After n samples, we have the estimate:

$$\hat{x}_n = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{5.18}$$

We can show that

$$\hat{x}_n = \hat{x}_{n-1} + \frac{1}{n}(x_n - \hat{x}_{n-1}) \tag{5.19}$$

$$= \hat{x}_{n-1} + \alpha(n)(x_n - \hat{x}_{n-1}). \tag{5.20}$$

The function $\alpha(n)$ is referred to as the *learning rate*. The learning rate can be a function other than $1/n$; there are rather loose conditions on the learning rate to ensure convergence to the mean. If the learning rate is constant, which is common in reinforcement learning applications, then the weights of older samples decay exponentially at the rate $(1-\alpha)$. With a constant learning rate, we can update our estimate after observing x using the following rule:

$$\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x}). \tag{5.21}$$

The update rule above will appear again in later sections and is related to stochastic gradient descent. The magnitude of the update is proportional to the difference in the sample and the previous estimate. The difference between the sample and previous estimate is called the *temporal difference error*.

5.4.2 Q-Learning

One of the most popular model-free reinforcement learning algorithms is *Q-learning*. The idea is to apply incremental estimation to the Bellman equation

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \tag{5.22}$$

$$= R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} Q(s',a'). \tag{5.23}$$

Instead of using T and R, we use the observed next state s' and reward r to obtain the following incremental update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)). \tag{5.24}$$

Q-learning is outlined in Algorithm 5.3. As with the model-based methods, some exploration strategy is required to ensure that Q converges to the optimal state-action value function. We can initialize Q to values other than 0 to encode any prior knowledge we may have about the environment.

Algorithm 5.3 Q-learning

```
1: function QLearning
2:     t ← 0
3:     s_0 ← initial state
4:     Initialize Q
5:     loop
6:         Choose action a_t based on Q and some exploration strategy
7:         Observe new state s_{t+1} and reward r_t
8:         Q(s_t, a_t) ← Q(s_t, a_t) + α(r_t + γ max_a Q(s_{t+1}, a) − Q(s_t, a_t))
9:         t ← t + 1
```

5.4.3 Sarsa

An alternative to Q-learning is *Sarsa*, which derives its name from the fact that it uses $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ to update the Q function at each step. It uses the actual action taken to update Q instead of maximizing over all possible actions as done in Q-learning. The Sarsa algorithm is identical to Algorithm 5.3 except that Line 8 is replaced with

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)). \tag{5.25}$$

With a suitable exploration strategy, a_{t+1} will converge to the $\arg\max_a Q(s_{t+1}, a)$ that is used to update the Q function in the Q-learning algorithm. Although Q-learning and Sarsa both converge to the optimal strategy, the speed of the convergence depends on the application.

5.4.4 Eligibility Traces

One of the disadvantages of Q-learning and Sarsa is that learning can be very slow. For example, suppose the environment has a single goal state that provides a large reward. The reward is zero at all other states. After some amount of random exploration in the environment, we reach the goal state. Regardless of whether we use Q-learning or Sarsa, we only update the state-action value of the state immediately preceding the goal state. The values at all other states leading up to the goal remain at zero. Much more exploration is required to slowly propagate non-zero values to the remainder of the state space.

Q-learning and Sarsa can be modified to assign credit to achieving the goal to past states and actions using *eligibility traces*. The reward associated with reaching the goal is propagated backward to the states and actions leading up to the goal. The credit is decayed exponentially, so states closer to the goal are assigned larger state-action values. It is common to use λ as the exponential decay parameter, and so the versions of Q-learning and Sarsa with eligibility traces are often called Q(λ) and Sarsa(λ).

Algorithm 5.4 shows a version of Sarsa(λ). We keep track of an exponentially decaying visit count $N(s,a)$ for all the state-action pairs. When we take action a_t in state s_t, $N(s_t,a_t)$ is incremented by 1. We then update $Q(s,a)$ by adding $\alpha\delta N(s,a)$ at every state s and for every action a, where

$$\delta = r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t). \tag{5.26}$$

After performing the updates, we decay $N(s,a) \leftarrow \gamma\lambda N(s,a)$. Although the impact of eligibility traces is especially pronounced in environments with sparse reward, the algorithm can speed learning in general environments where reward is more evenly distributed.

Algorithm 5.4 Sarsa(λ)-learning

```
function SarsaLambdaLearning(λ)
    Initialize Q and N
    t ← 0
    s_0, a_0 ← initial state and action
    loop
        Observe reward r_t and new state s_{t+1}
        Choose action a_{t+1} based on some exploration strategy
        N(s_t, a_t) ← N(s_t, a_t) + 1
        δ ← r_t + γQ(s_{t+1}, a_{t+1}) − Q(s_t, a_t)
        for s ∈ S
            for a ∈ A
                Q(s, a) ← Q(s, a) + αδN(s, a)
                N(s, a) ← γλN(s, a)
        t ← t + 1
```

5.5 Generalization

Up to this point in this chapter, we have assumed that the state-action value function can be represented as a table, which is only useful for small discrete problems. The problem with larger state spaces is not just the size of the state-action table but also the amount of experience required to accurately estimate the values. The agent must *generalize* from limited experience to states that have not yet been visited. Many different approaches have been explored, many related to techniques for approximate dynamic programming Section 4.5.

5.5.1 Local Approximation

The assumption in local approximation methods is that states that are close together are likely to have similar state-action values. A common technique is to store estimates of $Q(s,a)$ at a limited number of states in set S and actions in set A. We denote the vector containing these estimates as θ, which has $|S| \times |A|$ elements. To denote the component associated with state s and action a, we use $\theta_{s,a}$. If we use a weighting function such that $\sum_{s'} \beta(s,s') = 1$ for all s, then we can approximate the state action value at arbitrary states as

$$Q(s,a) = \sum_{s'} \theta_{s',a} \beta(s,s'). \tag{5.27}$$

We can define a vectorized version of the weighting function as follows:

$$\beta(s) = (\beta(s,s_1), \ldots, \beta(s,s_{|S|})), \tag{5.28}$$

where $s_1, \ldots, s_{|S|}$ are the states in S. We can also define a two-argument version of the function β that takes as input both a state and an action and returns a vector with $|S| \times |A|$ elements. The vector $\beta(s,a)$ is identical to $\beta(s)$, except that the elements associated with actions other than a are set to 0. This notation allows us to rewrite Equation (5.27) as follows:

$$Q(s,a) = \theta^\top \beta(s,a). \tag{5.29}$$

The linear approximation of Equation (5.29) can be easily integrated into Q-learning. The state-action value estimates represented by θ are updated as follows based on observing a state transition from s_t to s_{t+1} by action a_t with reward r_t:

$$\theta \leftarrow \theta + \alpha(r_t + \gamma \max_a \theta^\top \beta(s_{t+1},a) - \theta^\top \beta(s_t,a_t))\beta(s_t,a_t). \tag{5.30}$$

The update rule above comes from substituting Equation (5.29) directly into the standard Q-learning update rule and multiplying the last term by $\beta(s_t,a_t)$ to provide greater updates at states that are closer to s_t.

Algorithm 5.5 shows this linear approximation Q-learning method. If we have prior knowledge about the state-action values, then we can initialize θ appropriately. This linear approximation method can easily be extended to other reinforcement learning methods, such as Sarsa.

The algorithm presented above assumes that the points in S remain fixed. However, for some problems, it may be beneficial to adjust the locations of the points in S to produce a better approximation. The locations can be adjusted based on the temporal difference error using a representation such as a *self-organizing map* (see references in Section 5.7). Various criteria have been explored for identifying when it is appropriate to add new points to S, such as when a new state has been observed that is outside some

Algorithm 5.5 Linear approximation Q-learning

```
function LinearApproximationQLearning
    t ← 0
    s_0 ← initial state
    Initialize θ
    loop
        Choose action a_t based on θ_a^⊤ β(s_t) and some exploration strategy
        Observe new state s_{t+1} and reward r_t
        θ ← θ + α(r_t + γ max_a θ^⊤ β(s_{t+1}, a) − θ^⊤ β(s_t, a_t)) β(s_t, a_t)
        t ← t + 1
```

threshold distance of the states in S. Although memory can become an issue, some methods simply store all observed states.

5.5.2 Global Approximation

Global approximation methods do not rely on a notion of distance. One such approximation method is a *perceptron*. Perceptrons have been widely used since at least the 1950s to mimic individual neurons for various learning tasks. A perceptron has a set of input nodes $x_{1:n}$, a set of weights $\theta_{1:n}$, and an output node q. The value of the output node is determined as follows:

$$q = \sum_{i=1}^{m} \theta_i x_i = \theta^\top \mathbf{x}. \tag{5.31}$$

The structure of a perceptron is shown in Figure 5.3a.

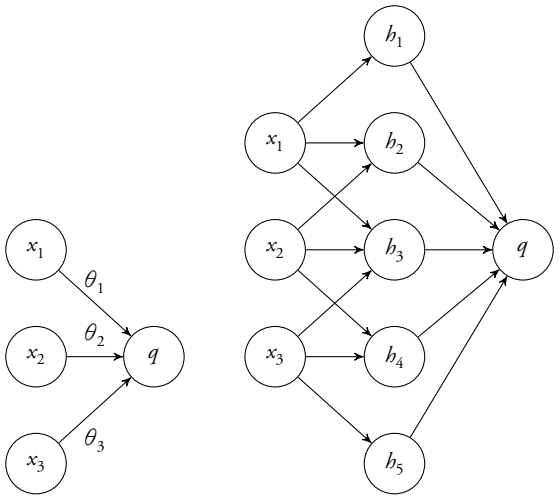
In *perceptron Q-learning*, we have a set of n perceptrons, one for each available action. The input is based on the state, and the output is the state-action value. We define a set of basis functions $\beta_1, \ldots, \beta_m$ over the state space, similar to the weighting functions in Section 5.5.1. The inputs to the perceptrons are $\beta_1(s), \ldots, \beta_m(s)$. If θ_a are the m weights associated with the perceptron for action a, then we have

$$Q(s, a) = \theta_a^\top \beta(s). \tag{5.32}$$

We can define a two-argument version of β as done in Section 5.5.1 and define θ to contain all the weights of all the perceptrons so that we can write

$$Q(s, a) = \theta^\top \beta(s, a). \tag{5.33}$$

Q-learning with a perceptron-based approximation follows Algorithm 5.5 exactly, but θ represents perceptron weights instead of value estimates and β represents basis functions instead of distance measures.



(a) Perceptron. (b) Neural network.

Figure 5.3 Approximation structures.

Perceptrons only represent linear functions, but *neural networks* can represent nonlinear functions. A neural network is a network of perceptrons. They are organized into an input layer, a hidden layer, and an output layer as shown in Figure 5.3b; the weights are omitted from the diagram. Adding hidden nodes generally increases the complexity of the state-action function the network can represent.

We can use an algorithm known as *backpropagation* to adjust the weights in the neural network to reduce the temporal difference error. The idea is to first adjust the weights associated with the edges leading from the hidden nodes to the output node in much the same way it is done for perceptron learning. We then compute the error associated with the hidden nodes and adjust the weights from the input nodes to the hidden nodes appropriately. Although convergence when using this form of function approximation is not guaranteed, it can result in satisfactory performance in a variety of domains.

5.5.3 Abstraction Methods

Abstraction methods involve partitioning the state space into discrete regions and estimating the state-action values for each of these regions. Abstraction methods tend to use model-based learning, which often results in faster convergence than model-free learning. Abstraction methods often use decision trees to partition the state space. Associated with the internal nodes of the tree are various tests along the various dimensions of the state space. The leaf nodes correspond to regions.

There are a variety of different abstraction methods, but one method is to start with a single region represented by a decision tree with a single node and then apply a series of acting, modeling, and planning phases. In the acting phase, we select an action based on the state-action value of the region associated with the current state s. After observing the transition from state s to s' by action a with reward r, the experience tuple (s, a, s', r) is stored in the leaf node associated with s.

In the modeling phase, we decide whether to split nodes. For each of the experience tuples at all of the leaf nodes, we compute

$$q(s, a) = r + \gamma U(s'), \tag{5.34}$$

where r is the observed reward and $U(s')$ is the value of the leaf node associated with the next state s'. We want to split a leaf node when the values of the experience tuples come from different distributions. One approach is to choose the split that minimizes the variance of the experience tuples at the resulting leaves. We stop splitting when some stopping criterion is met, such as when the variance at the leaf nodes is below some threshold.

In the planning phase, we use the experience tuples at the leaf nodes to estimate the state transition model and reward model. We then solve the resulting MDP using dynamic programming. The modeling and planning procedures require much more computation than what is required in the other generalization methods, but this approach can find better policies without as much interaction with the environment.

5.6 Summary

- Reinforcement learning is a computational approach to learning intelligent behavior from experience.
- Exploration must be carefully balanced with exploitation.
- In general, solving the exploration problem optimally is not feasible, but there are several Bayesian and heuristic approximation methods that can work well.
- It is important to determine how much actions in the past are responsible for later rewards.
- Model-based reinforcement learning involves building a model from experience and using this model to generate a plan.
- Model-free reinforcement learning involves directly estimating the values of states and actions without the use of transition and reward models.
- We must generalize from observations of rewards and state transitions because our interaction with the world is limited.
- Generalization can be done in a variety of ways, such as local and global approximations of the value function and state abstraction.

5.7 Further Reading

The classic book by Sutton and Barto, titled *Reinforcement Learning: An Introduction*, is the standard introductory text on classical reinforcement learning and provides a historical overview of the emergence of the field [2]. The volume edited by Wiering and Otterlo provides an up-to-date survey of much of the research that occurred since the publication of the Sutton and Barto book [3]. Kovacs and Egginton survey software for reinforcement learning [4].

Multi-armed bandit problems and their many variations have received considerable attention over the years [5]. Gittins developed the concept of an allocation index for solving multi-armed bandit problems [1]. Recent work has focused on improving the efficiency of computing allocation indices [6], [7].

Model-based reinforcement learning can be grouped into non-Bayesian and Bayesian methods [8]. The non-Bayesian methods generally rely on maximum likelihood estimation as discussed in Section 5.2. The Dyna approach was introduced by Sutton [9]. Prioritized sweeping was introduced by Moore and Atkeson [10].

Bayesian model-based methods have started to receive attention more recently [11]. Duff discusses the formulation of model-based reinforcement learning as a Bayes-adaptive Markov decision process [12]. In general, solving such a belief-state formulation exactly is intractable. Strens applies the concept of Thompson sampling [14] to model-based reinforcement learning [13]. Variants of the online planning algorithms presented in the previous chapter have been extended to Bayesian model-based reinforcement learning, including sparse sampling [15] and Monte Carlo tree search [16], [17].

Model-free reinforcement learning algorithms are often used for situations in which it is not feasible to build an explicit representation of the transition and reward models. Q-learning and Sarsa are two commonly used model-free techniques. Eligibility traces were proposed in the context of temporal difference learning by Sutton [18], and they were extended to Sarsa(λ) [19] and $Q(\lambda)$ [20], [21].

Much of the ongoing work in the field of reinforcement learning is concerned with generalizing from limited experience. The recent book *Reinforcement Learning and Dynamic Programming Using Function Approximators* by Busoniu et al. surveys a variety of different local and global function approximation methods [22]. Several different abstraction methods have been proposed over the years [23]–[26].

Although not discussed in this chapter, there has been some work on Bayesian approaches to model-free reinforcement learning. One approach is to maintain a distribution over state-action values [27], [28]. There are also Bayesian policy gradient methods that have been used with some success [29]. Multiagent reinforcement learning was also not discussed in this chapter, but Busoniu, Babuska, and De Schutter survey recent research in the area [30].

References

1. J.C. Gittins, "Bandit Processes and Dynamic Allocation Indices," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 41, no. 2, pp. 148–177, 1979.
2. R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.
3. M. Wiering and M. van Otterlo, eds., *Reinforcement Learning: State of the Art*. New York: Springer, 2012.
4. T. Kovacs and R. Egginton, "On the Analysis and Design of Software for Reinforcement Learning, with a Survey of Existing Systems," *Machine Learning*, vol. 84, no. 1-2, pp. 7–49, 2011. DOI: 10.1007/s10994-011-5237-8.
5. J. Gittins, K. Glazebrook, and R. Weber, *Multi-Armed Bandit Allocation Indices*, 2nd ed. Hoboken, NJ: Wiley, 2011.

6. J. Niño-Mora, "A $(2/3)^n$ Fast-Pivoting Algorithm for the Gittins Index and Optimal Stopping of a Markov Chain," *INFORMS Journal on Computing*, vol. 19, no. 4, pp. 596–606, 2007. DOI: 10.1287/ijoc.1060.0206.
7. I.M. Sonin, "A Generalized Gittins Index for a Markov Chain and Its Recursive Calculation," *Statistics and Probability Letters*, vol. 78, no. 12, pp. 1526–1533, 2008. DOI: 10.1016/j.spl.2008.01.049.
8. P.R. Kumar, "A Survey of Some Results in Stochastic Adaptive Control," *SIAM Journal on Control and Optimization*, vol. 23, no. 3, pp. 329–380, 1985. DOI: 10.1137/0323023.
9. R.S. Sutton, "Dyna, an Integrated Architecture for Learning, Planning, and Reacting," *SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991. DOI: 10.1145/122344.122377.
10. A.W. Moore and C.G. Atkeson, "Prioritized Sweeping: Reinforcement Learning With Less Data and Less Time," *Machine Learning*, vol. 13, no. 1, pp. 103–130, 1993. DOI: 10.1007/BF00993104.
11. P. Poupart, N.A. Vlassis, J. Hoey, and K. Regan, "An Analytic Solution to Discrete Bayesian Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2006.
12. M.O. Duff, "Optimal Learning: Computational Procedures for Bayes-Adaptive Markov Decision Processes," PhD thesis, University of Massachusetts at Amherst, 2002.
13. M.J.A. Strens, "A Bayesian Framework for Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2000.
14. W.R. Thompson, "On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933. DOI: 10.2307/2332286.
15. T. Wang, D.J. Lizotte, M.H. Bowling, and D. Schuurmans, "Bayesian Sparse Sampling for On-Line Reward Optimization," in *International Conference on Machine Learning (ICML)*, 2005.
16. J. Asmuth and M.L. Littman, "Learning Is Planning: Near Bayes-Optimal Reinforcement Learning via Monte-Carlo Tree Search," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011.
17. A. Guez, D. Silver, and P. Dayan, "Scalable and Efficient Bayes-Adaptive Reinforcement Learning Based on Monte-Carlo Tree Search," *Journal of Artificial Intelligence Research*, vol. 48, pp. 841–883, 2013. DOI: 10.1613/jair.4117.
18. R. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988. DOI: 10.1007/BF00115009.

19. G.A. Rummery, "Problem Solving with Reinforcement Learning," PhD thesis, University of Cambridge, 1995.
20. C.J. C.H. Watkins, "Learning from Delayed Rewards," PhD thesis, University of Cambridge, 1989.
21. J. Peng and R.J. Williams, "Incremental Multi-Step Q-Learning," *Machine Learning*, vol. 22, no. 1-3, pp. 283–290, 1996. DOI: 10.1023/A:1018076709321.
22. L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, FL: CRC Press, 2010.
23. D. Chapman and L.P. Kaelbling, "Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1991.
24. A.K. McCallum, "Reinforcement Learning with Selective Perception and Hidden State," PhD thesis, University of Rochester, 1995.
25. W.T.B. Uther and M.M. Veloso, "Tree Based Discretization for Continuous State Space Reinforcement Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, 1998.
26. M.J. Kochenderfer, "Adaptive Modelling and Planning for Learning Intelligent Behaviour," PhD thesis, University of Edinburgh, 2006.
27. R. Dearden, N. Friedman, and S.J. Russell, "Bayesian Q-Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, 1998.
28. Y. Engel, S. Mannor, and R. Meir, "Reinforcement Learning with Gaussian Processes," in *International Conference on Machine Learning (ICML)*, 2005.
29. M. Ghavamzadeh and Y. Engel, "Bayesian Policy Gradient Algorithms," in *Advances in Neural Information Processing Systems (NIPS)*, 2006.
30. L. Busoniu, R. Babuska, and B. De Schutter, "A Comprehensive Survey of Multiagent Reinforcement Learning," *IEEE Transactions on Systems Science and Cybernetics Part*, vol. 38, no. 2, pp. 156 –172, 2008. DOI: 10.1109/TSMCC.2007.913919.

6

State Uncertainty

Mykel J. Kochenderfer

The previous two chapters discussed sequential decision-making problems in which the current state is known by the agent. Because of sensor limitations or noise, the state might not be perfectly observable. This chapter discusses sequential decision problems with state uncertainty and methods for computing optimal and approximately optimal solutions.

6.1 Formulation

A sequential decision problem with state uncertainty can be modeled as a *partially observable Markov decision process* (POMDP). A POMDP is an extension to the MDP formulation introduced in Chapter 4. In a POMDP, a model specifies the probability of making a particular observation given the current state.

6.1.1 Example Problem

Suppose we are assigned the task of taking care of a baby. We decide when to feed the baby on the basis of whether the baby is crying. Crying is a noisy indication that the baby is hungry. There is a 10% chance the baby cries when not hungry, and there is a 80% chance the baby cries when hungry.

The dynamics are as follows. If we feed the baby, then the baby stops being hungry at the next time step. If the baby is not hungry and we do not feed the baby, then 10% of the time the baby may become hungry at the next time step. Once hungry, the baby continues being hungry until fed.

The cost of feeding the baby is 5 and the cost of the baby being hungry is 10. These costs are additive, and so if we feed the baby when the baby is hungry, then there is a cost of 15. We want to find the optimal strategy assuming an infinite horizon with a discount factor of 0.9. Figure 6.1 shows the structure of the crying-baby problem as a dynamic decision network.

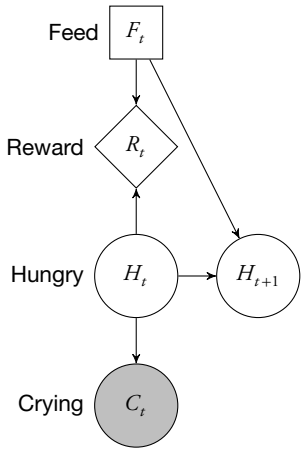


Figure 6.1 Crying-baby problem structure.

6.1.2 Partially Observable Markov Decision Processes

A POMDP is an MDP with an observation model. The probability of observing o given state s is written $O(o \mid s)$. In some formulations, the observation can also depend on the action a, and so we can write $O(o \mid s, a)$. The decisions in a POMDP at time t can only be based on the history of observations $o_{1:t}$. Instead of keeping track of arbitrarily long histories, it is common to keep track of the *belief state*. A belief state is a distribution over states. In belief state b, probability $b(s)$ is assigned to being in state s. A policy in a POMDP is a mapping from belief states to actions. The structure of a POMDP can be represented using the dynamic decision network in Figure 6.2.

6.1.3 Policy Execution

Algorithm 6.1 outlines how a POMDP policy is executed. We choose actions on the basis of the policy evaluated at the current belief state. Different ways to represent policies will be discussed in this chapter. When we receive a new observation and reward, we update our belief state. Belief-state updating is discussed in Section 6.2.

6.1.4 Belief-State Markov Decision Processes

A POMDP is really an MDP in which the states are belief states. We sometimes call the MDP over belief states a *belief-state MDP*. The state space of a belief-state MDP is simply the set of all possible beliefs, $\mathcal{B}$, in the POMDP. If there are n discrete states, then $\mathcal{B}$ is a subset of $\mathbb{R}^n$. The set of actions in the belief-state MDP is exactly the same

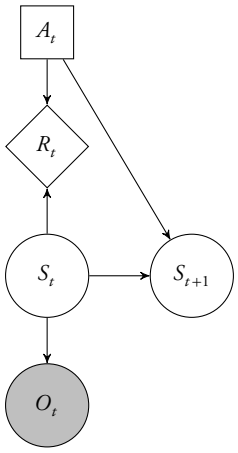


Figure 6.2 POMDP problem structure.

Algorithm 6.1 POMDP policy execution

```
function POMDPPolicyExecution(π)
    b ← initial belief state
    loop
        Execute action a = π(b)
        Observe o and reward r
        b ← UpdateBelief(b, a, o)
```

as for the POMDP. The state transition function $\tau(b' \mid b, a)$ is given by

$$\begin{aligned}
\tau(b' \mid b, a) &= P(b' \mid b, a) && (6.1)\\
&= \sum_o P(b' \mid b, a, o) P(o \mid b_t, a_t) && (6.2)\\
&= \sum_o P(b' \mid b, a, o) \sum_{s'} P(o \mid b, a, s') P(s' \mid b, a) && (6.3)\\
&= \sum_o P(b' \mid b, a, o) \sum_{s'} O(o \mid s') \sum_s P(s' \mid b, a, s) P(s \mid b, a) && (6.4)\\
&= \sum_o P(b' \mid b, a, o) \sum_{s'} O(o \mid s') \sum_s T(s' \mid s, a) b(s). && (6.5)
\end{aligned}$$

Here, $P(b' \mid b, a, o) = \delta_{b'}(\textsc{UpdateBelief}(b, a, o))$, with δ being the Kronecker delta function. The immediate reward in the belief-state MDP is

$$R(b, a) = \sum_s R(s, a) b(s). \tag{6.6}$$

Solving belief-state MDPs is challenging because the state space is continuous. We can use the approximate dynamic programming techniques presented in Section 4.5, but we can often do better by taking advantage of the structure of the belief-state MDP. This chapter will discuss such techniques after elaborating on how to update beliefs.

6.2 Belief Updating

Given an initial belief state, we can update our belief state using recursive Bayesian estimation based on the last observation and action executed. The update can be done exactly for problems with discrete states and for problems with linear-Gaussian dynamics and observations. For general problems with continuous state spaces, we often have to rely on approximation methods. This section presents methods for belief updating.

6.2.1 Discrete State Filter

In problems with a discrete state space, applying recursive Bayesian estimation is straightforward. Suppose our initial belief state is b, and we observe o after executing a. Our

new belief state b' is given by

$$b'(s') = P(s' \mid o, a, b) \tag{6.7}$$
$$\propto P(o \mid s', a, b) P(s' \mid a, b) \tag{6.8}$$
$$\propto O(o \mid s', a) P(s' \mid a, b) \tag{6.9}$$
$$\propto O(o \mid s', a) \sum_s P(s' \mid a, b, s) P(s \mid a, b) \tag{6.10}$$
$$\propto O(o \mid s', a) \sum_s T(s' \mid a, s) b(s). \tag{6.11}$$

The observation space can be continuous without posing any difficulty in computing the equation above exactly. The value $O(o \mid s', a)$ would represent a probability density rather than a probability mass.

To illustrate belief state updating, we will use the crying-baby problem. We simply apply Equation (6.11) to the model outlined in Section 6.1.1. Here are the first six steps of one potential scenario.

1. We begin with an initial belief state that assigns $b(h^0) = 0.5$ and $b(h^1) = 0.5$; in other words, uniform probability is assigned to whether the baby is hungry. If we had some prior belief that babies tend to be hungry more often than not, then we could have chosen a different initial belief. For compactness, we will represent our beliefs as tuples, $(b(h^0), b(h^1))$. In this case, our initial belief state is $(0.5, 0.5)$.
2. We do not feed the baby and the baby cries. According to Equation (6.11), the new belief state is $(0.0928, 0.9072)$. Although the baby is crying, it is only a noisy indication that the baby is actually hungry.
3. We feed the baby and the baby stops crying. Because we know that feeding the baby deterministically makes the baby not hungry, the result of our belief update is $(1, 0)$.
4. We do not feed the baby and the baby does not cry. In the prior step, we were certain that the baby was not hungry, and the dynamics specify that the baby becomes hungry at the next time step only 10% of the time. The fact that the baby is not crying further reduces our belief that the baby is hungry. Our new belief state is $(0.9759, 0.0241)$.
5. Again, we do not feed the baby and the baby does not cry. Our belief that the baby is hungry increases slightly. The new belief state is $(0.9701, 0.0299)$.
6. We do not feed the baby and the baby begins to cry. Our new belief is then $(0.4624, 0.5376)$. Because we were fairly certain that the baby was not hungry to begin with, our confidence that the baby is now hungry is significantly lower than when the baby started crying in the second step.

6.2.2 Linear-Gaussian Filter

If we generalize the linear-Gaussian dynamics in Section 4.4 to partial observability, we find that we can perform exact belief updates by using what is known as a *Kalman filter*. The dynamics and observations have the following form:

$$T(\mathbf{z} \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma_s) \tag{6.12}$$

$$O(\mathbf{o} \mid \mathbf{s}) = \mathcal{N}(\mathbf{o} \mid \mathbf{O}_s \mathbf{s}, \Sigma_o). \tag{6.13}$$

Hence, the continuous dynamics and observation models are specified using matrices $\mathbf{T}_s$, $\mathbf{T}_a$, Σ_s, $\mathbf{O}_s$, and Σ_o.

We assume that the initial belief state is represented by a Gaussian:

$$b(\mathbf{s}) = \mathcal{N}(\mathbf{s} \mid \mu_b, \Sigma_b). \tag{6.14}$$

Under the linear-Gaussian assumptions for the dynamics and observations, it can be shown that the belief state can be updated as follows:

$$\Sigma_b \leftarrow \mathbf{T}_s(\Sigma_b - \Sigma_b \mathbf{O}_s^\top (\mathbf{O}_s \Sigma_b \mathbf{O}_s^\top + \Sigma_o)^{-1} \mathbf{O}_s \Sigma_b)\mathbf{T}_s^\top + \Sigma_s \tag{6.15}$$

$$\mathbf{K} \leftarrow \mathbf{T}_s \Sigma_b \mathbf{O}_s^\top (\mathbf{O}_s \Sigma_b \mathbf{O}_s^\top + \Sigma_o)^{-1} \tag{6.16}$$

$$\mu_b \leftarrow \mathbf{T}_s \mu_b + \mathbf{T}_a \mathbf{a} + \mathbf{K}(\mathbf{o} - \mathbf{O}_s \mu_b). \tag{6.17}$$

The matrix $\mathbf{K}$ used for computing μ_b is called the *Kalman gain*. Kalman filters are often applied to systems that do not actually have linear-Gaussian dynamics. A variety of different modifications to the basic Kalman filter have been proposed to better accommodate nonlinear dynamics, as discussed in Section 6.7.

6.2.3 Particle Filter

If the state space is large or continuous and the dynamics are not well approximated by a linear-Gaussian model, then a sampling-based approach can be used to perform belief updates. The belief state is represented as a collection of *particles*, which are simply samples from the state space. The algorithm for adjusting these particles based on observations is known as a *particle filter*. There are many different variations of the particle filter, including versions in which the particles are assigned weights.

Our belief b is simply a set of samples from the state space. Updating b is based on a generative model G. We can draw a sample $(s', o') \sim G(s, a)$, which gives the next state s' and observation o' given the current state s and action a. The generative model can be implemented as a black-box simulator, without any explicit knowledge of the actual transition or observation probabilities.

Algorithm 6.2 returns the updated belief state b' based on the current belief state b, action a, and observation o. The process for generating a set of $|b|$ new particles is simple. Each sample is generated by randomly selecting a sample in b and then drawing samples $(s', o') \sim G(s, a)$ until the sampled o' matches the observed o. The sampled s' is then added to the new belief state b'.

Algorithm 6.2 Particle filter with rejection

```
function UpdateBelief(b, a, o)
    b' ← ∅
    for i ← 1 to |b|
        s ← random state in b
        repeat
            (s', o') ~ G(s, a)
        until o' = o
        Add s' to b'
    return b'
```

The problem with the version of the particle filter in Algorithm 6.2 is that many draws from the generative model may be required until the sampled observation is consistent with actual observation. The problem of rejecting many observation draws becomes especially apparent when the observation space is large or continuous. This issue was observed in Section 2.2.5 where we were trying to perform inference using direct sampling from a Bayesian network. The remedy presented in that section was to not sample the observed values but to weight the results using the likelihood of the observations.

Algorithm 6.3 is a version of a particle filter that does not involve rejecting samples. In this version, the generative model only returns states, instead of both states and observations. An observation model is required that specifies $O(o \mid s, a)$, which can be either a probability mass function or a probability density function depending on whether the observation space is continuous.

The algorithm is broken into two stages. The first stage involves generating $|b|$ new samples by randomly selecting samples in b and then propagating them forward using the generative model. For each of the new samples s'_i, we compute a corresponding weight w_i based on $O(o \mid s'_i, a)$, where o is the actual observation and a is the action taken. The second stage involves building the updated belief state b' by drawing $|b|$ samples from the set of new state samples with probability proportional to their weights.

In both versions of the particle filter presented here, it can be shown that as the number of particles increases, the distribution represented by the particles approaches the true posterior distribution. However, in practice, particle filters can fail. Because of the random nature of the particle filter, it is possible that a series of samples can lead

Algorithm 6.3 Particle filter without rejection

```
function UpdateBelief(b, a, o)
    b' ← ∅
    for i ← 1 to |b|
        s_i ← random state in b
        s'_i ~ G(s_i, a)
        w_i ← O(o | s'_i, a)
    for i ← 1 to |b|
        Randomly select k with probability proportional to w_k
        Add s'_k to b'
    return b'
```

to no particles near the true state. This problem, known as *particle deprivation*, can be mitigated to some extent by introducing additional noise to the particles.

6.3 Exact Solution Methods

As discussed earlier, a policy for a POMDP is a mapping from belief states to actions. This section explains how to compute and represent optimal policies.

6.3.1 Alpha Vectors

For now, let us assume that we are interested in computing the optimal policy for a discrete state POMDP with a one-step horizon. We know $U^*(s) = \max_a R(s,a)$, but because we do not know the state exactly in a POMDP, we have

$$U^*(b) = \max_a \sum_s b(s)R(s,a), \tag{6.18}$$

where b is our current belief state. If we let $\boldsymbol{\alpha}_a$ represent $R(\cdot, a)$ as a vector and $\mathbf{b}$ represent our belief state as a vector, then we can rewrite Equation (6.18) as

$$U^*(\mathbf{b}) = \max_a \boldsymbol{\alpha}_a^\top \mathbf{b}. \tag{6.19}$$

The α_a in the equation above is often referred to as an *alpha vector*. We have an alpha vector for each action in this single-step POMDP. These alpha vectors define hyperplanes in belief space. As can be seen in Equation (6.19), the optimal value function is piecewise-linear and convex.

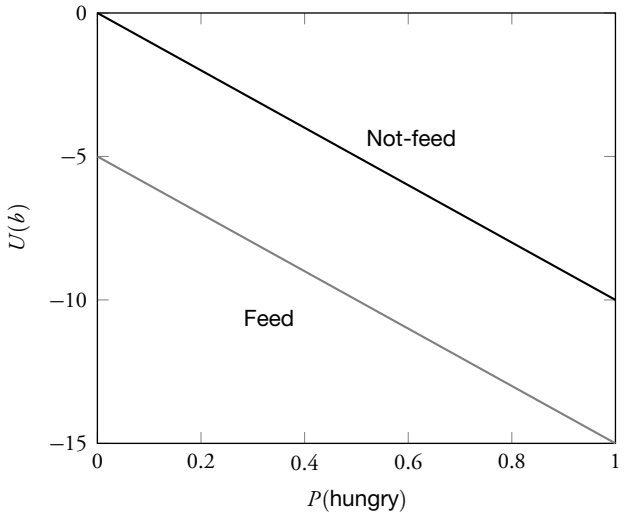


Figure 6.3 Alpha vectors for a one-step version of the crying-baby problem.

Figure 6.3 shows the alpha vectors associated with the crying-baby problem. If the belief vector is represented by the pair (b(not-hungry), b(hungry)), then the two alpha vectors are

$$\alpha_{\text{not-feed}} = (0, -10) \tag{6.20}$$

$$\alpha_{\text{feed}} = (-5, -15). \tag{6.21}$$

What is apparent from the plot is that regardless of our current beliefs, the one-step optimal policy is to not feed the baby. Given the dynamics of the problem, we do not see any potential benefit of feeding the baby until at least one step later.

6.3.2 Conditional Plans

The alpha vectors introduced in the computation of the optimal one-step policy can be generalized to an arbitrary horizon. In multistep POMDPs, we can think of a policy as a *conditional plan* represented as a tree. We start at the root node, which tells us what action to take at the first time step. After taking that action, we transition to one of the child nodes depending on what we observe. That child node tells us what action to take. We then proceed down the tree.

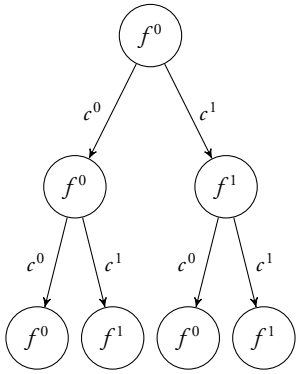


Figure 6.4 Example three-step conditional plan.

Figure 6.4 shows an example of a three-step plan for the crying-baby problem. The root node specifies that we do not feed the baby at the first step. For the second time step, as indicated by the directed edges, we feed the baby if there is crying; otherwise, we do not. At the third time step, we again feed the baby only if there is crying.

We can recursively compute $U^p(s)$, the expected utility associated with conditional plan p when starting in state s:

$$U^p(s) = R(s,a) + \sum_{s'} T(s' \mid s,a) \sum_o O(o \mid s',a) U^{p(o)}(s'), \tag{6.22}$$

where a is the action associated with the root node of p and $p(o)$ represents the subplan associated with observation o. We can compute the expected utility associated with a belief state as follows:

$$U^p(b) = \sum_s U^p(s) b(s). \tag{6.23}$$

We can use the alpha vector $\boldsymbol{\alpha}_p$ to represent the vectorized version of U^p. If $\mathbf{b}$ is the belief vector, then we can write

$$U^p(\mathbf{b}) = \boldsymbol{\alpha}_p^\top \mathbf{b}. \tag{6.24}$$

If we maximize over the space of all possible plans up to the planning horizon, then we can find

$$U^*(\mathbf{b}) = \max_p \boldsymbol{\alpha}_p^\top \mathbf{b}. \tag{6.25}$$

Hence, the finite horizon optimal value function is piecewise-linear and convex. We simply execute the action at the root node of the plan that maximizes $\boldsymbol{\alpha}_p^\top \mathbf{b}$.

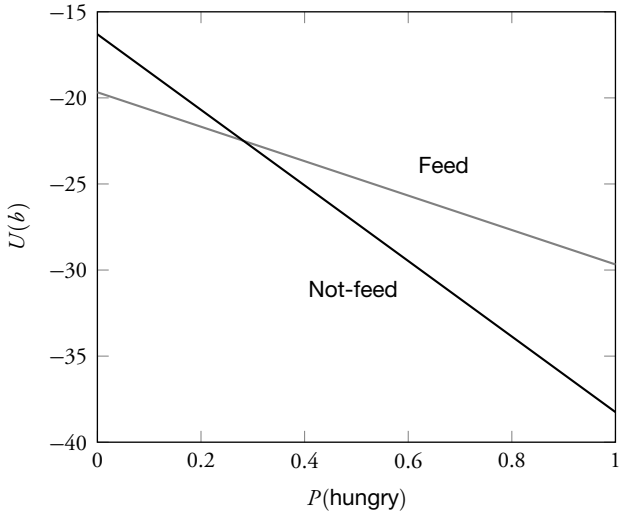


Figure 6.5 Optimal policy for the crying-baby problem.

6.3.3 Value Iteration

It is generally infeasible to enumerate every possible h-step plan to find the one that maximizes Equation (6.25) from the current belief state; the number of nodes in an h-step conditional plan is $(|O|^h - 1)/(|O| - 1)$. Associated with each node are $|A|$ actions. Hence, we have $|A|^{(|O|^h-1)/(|O|-1)}$ possible h-step plans. Even for our crying-baby problem with two actions and two observations, there are 2^{63} six-step conditional plans—too many to enumerate.

The idea in POMDP value iteration is to iterate over all the one-step plans and toss out the plans that are not optimal for any initial belief state. The remaining one-step plans are then used to generate potentially optimal two-step plans. Again, plans that are not optimal for any belief state are discarded. The process repeats until the desired horizon is reached. Identifying the plans that are *dominated* at certain belief states by other plans can be done using linear programming.

Figure 6.5 shows the two, nondominated, alpha vectors for the crying-baby problem with a discount factor of 0.9. The two alpha vectors intersect when $P(\text{hungry}) = 0.28206$. As indicated in the figure, we only want to feed the baby if $P(\text{hungry}) > 0.28206$. Of course, for this problem, it would have been easier to simply store this threshold instead of using alpha vectors, but for higher dimensional problems, alpha vectors often provide a compact representation of the policy.

Discarding dominated plans can significantly reduce the computation required to find the optimal set of alpha vectors. For many problems, the majority of the potential plans are dominated by at least one other plan. However, in the worst case, an exact solution for a general finite-horizon POMDP is *PSPACE-complete*, which is a complexity class that includes NP-complete problems and is suspected to include problems even more difficult. General infinite-horizon POMDPs have been shown to be *uncomputable*. Hence, there has been a tremendous amount of research recently on approximation methods, which will be discussed in the remainder of this chapter.

6.4 Offline Methods

Offline POMDP solution methods involve performing all or most of the computation prior to execution. In practice, we are generally restricted to finding only approximately optimal solutions. Some methods represent policies as alpha vectors, whereas others use finite-state controllers.

6.4.1 Fully Observable Value Approximation

A simple approximation technique is called *QMDP*. The idea is to create a set of alpha vectors, one for each action, based on the state-action value function $Q(s, a)$ under full observability. We can use value iteration to compute the alpha vectors. If we initialize $\alpha_a^{(0)}(s) = 0$ for all s, then we can iterate

$$\alpha_a^{(k+1)}(s) = R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) \max_{a'} \alpha_{a'}^{(k)}(s'). \tag{6.26}$$

Each iteration requires $O(|A|^2|S|^2)$ operations. As $k \to \infty$, the resulting set of $|A|$ alpha vectors can be used to estimate the value function. The value function at belief state $\mathbf{b}$ is given by $\max_a \boldsymbol{\alpha}_a^\top \mathbf{b}$, and the approximately optimal action is given by $\arg\max_a \boldsymbol{\alpha}_a^\top \mathbf{b}$.

The QMDP method assumes all state uncertainty disappears at the next time step. It can be shown that QMDP provides an upper bound on the value function. In other words, $\max_a \boldsymbol{\alpha}_a^\top \mathbf{b} \geq U^*(\mathbf{b})$ for all $\mathbf{b}$. QMDP tends to have difficulty with problems with information-gathering actions, such as "look over your right shoulder when changing lanes." However, the method performs extremely well in many real problems in which the particular choice of action has little impact on the reduction in state uncertainty.

6.4.2 Fast Informed Bound

Just as with the QMDP approximation, the *fast informed bound* (FIB) method computes a single alpha vector for each action. However, the fast informed bound takes into account, to some extent, partial observability. Instead of using the iteration of

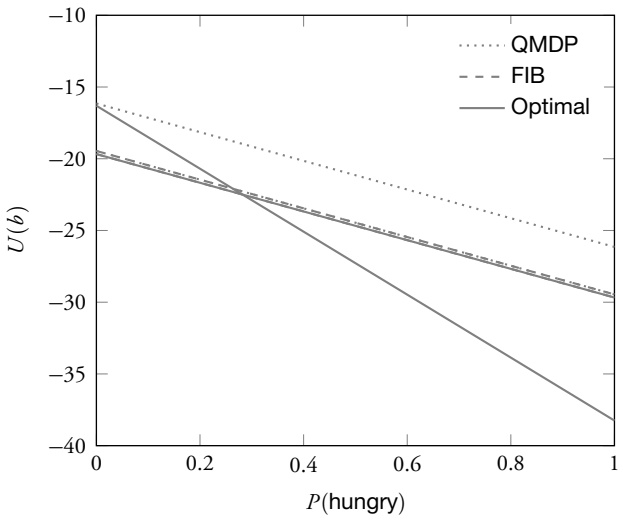


Figure 6.6 QMDP, FIB, and optimal alpha vectors for the crying-baby problem.

Equation (6.26), we use

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_o \max_{a'} \sum_{s'} O(o \mid s', a) T(s' \mid s, a) \alpha_{a'}^{(k)}(s'). \tag{6.27}$$

Each iteration requires $O(|A|^2|S|^2|O|)$ operations, only a factor of $|O|$ more than QMDP. At all belief states, the fast informed bound provides an upper bound on the optimal value function that is no higher than that of QMDP. Figure 6.6 compares the alpha vectors associated with QMDP, FIB, and the optimal policy for the crying baby problem.

6.4.3 Point-Based Value Iteration

There is a family of approximation methods that involves backing up alpha vectors associated with a limited number of points in the belief space. Let us denote the set of belief points $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ and their associated alpha vectors $\Gamma = \{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n\}$. Given these n alpha vectors, we can estimate the value function at any new point $\mathbf{b}$ as follows:

$$U^\Gamma(\mathbf{b}) = \max_{\boldsymbol{\alpha} \in \Gamma} \boldsymbol{\alpha}^\top \mathbf{b} = \max_{\alpha \in \Gamma} \sum_s \alpha(s) b(s). \tag{6.28}$$

For the moment, let us assume that these belief points are given to us; we will discuss how to choose these later in Section 6.4.5. We would like to initialize the alpha vectors in Γ such that $U^{\Gamma}(\mathbf{b}) \leq U^{*}(\mathbf{b})$ for all $\mathbf{b}$. One way to compute such a lower bound is to initialize all the components of all n alpha vectors to

$$\max_a \sum_{t=0}^{\infty} \gamma^t \min_s R(s,a) = \frac{1}{1-\gamma} \max_a \min_s R(s,a). \tag{6.29}$$

As we perform backups starting with this initial set of alpha vectors, we guarantee that $U(\mathbf{b})$ at each iteration never decreases for any $\mathbf{b}$.

We may update the value function at belief b based on the n alpha vectors

$$U(b) \leftarrow \max_a \left[R(b,a) + \gamma \sum_o P(o \mid a, b) U(b') \right], \tag{6.30}$$

where b' is as determined by UpdateBelief(b,a,o), $U(b')$ is as evaluated by Equation (6.28), and

$$P(o \mid b, a) = \sum_s O(o \mid s, a) b(s). \tag{6.31}$$

We know from Bayes' rule that

$$b'(s') = \frac{O(o \mid s', a)}{P(o \mid b, a)} \sum_s T(s' \mid s, a) b(s). \tag{6.32}$$

Combining Equations (6.28), (6.30), and (6.32) and simplifying, we get the update

$$U(b) \leftarrow \max_a \left[R(b,a) + \gamma \sum_o \max_{\alpha \in \Gamma} \sum_s b(s) \sum_{s'} O(o \mid s', a) T(s' \mid s, a) \alpha(s') \right]. \tag{6.33}$$

Besides simply updating the value at $\mathbf{b}$, we can compute an alpha vector at $\mathbf{b}$ by using Algorithm 6.4. Point-based value iteration approximation algorithms update the alpha vectors at the n belief states until convergence. These n alpha vectors can then be used to approximate the value function anywhere in the belief space.

6.4.4 Randomized Point-Based Value Iteration

The point-based value iteration approach discussed in Section 6.4.3 associates an alpha vector for each of the selected points in the belief space. To reduce the amount of computation required to perform an update of all the belief points, we can attempt to limit the number of alpha vectors representing the value function using the approach outlined in Algorithm 6.5.

Algorithm 6.4 Backup belief

```
function BackupBelief(Γ, b)
    for a ∈ A
        for o ∈ O
            b′ ← UpdateBelief(b, a, o)
            α_{a,o} ← arg max_{α∈Γ} α⊤b′
        for s ∈ S
            α_a(s) ← R(s,a) + γ Σ_{s′,o} O(o | s′, a) T(s′ | s, a) α_{a,o}(s′)
    α ← arg max_{α_a} α_a⊤ b
    return α
```

The algorithm begins by initializing Γ with a single alpha vector with all components set to Equation (6.29), which lower bounds the value function. Given this Γ and our belief points B, we call RandomizedPointBasedUpdate(B, Γ) to create a new set of alpha vectors that provides a tighter lower bound on the value function. These new alpha vectors can be improved on further through another call to this function. The process repeats until convergence.

Each update involves finding a set of alpha vectors Γ' that improves on the value function represented by Γ at the points in B. In other words, the update finds a set Γ' such that $U^{\Gamma'}(\mathbf{b}) \geq U^{\Gamma}(\mathbf{b})$ for all $\mathbf{b} \in B$. We begin by initializing Γ' to the empty set and the set B' to B. We then take a point $\mathbf{b}$ randomly from B' and call BackupBelief$(\mathbf{b}, \Gamma)$ to get a new alpha vector $\boldsymbol{\alpha}$. If this alpha vector improves the value at $\mathbf{b}$, then we add it to Γ'; otherwise, we find the alpha vector in Γ that dominates at $\mathbf{b}$ and add it to Γ'. The set B' then becomes the set of points that still have not been improved by Γ'. At each iteration, B' becomes smaller, and the process terminates when B' is empty.

6.4.5 Point Selection

Many point-based value iteration algorithms involve starting with B initialized to a set containing only the initial belief state b_0 and then iteratively expanding that set. One of the simplest ways to expand a set B is to select actions from each belief state B (based on some exploration strategy from Section 5.1.3) and then add the resulting belief states to B (Algorithm 6.6). This process requires sampling observations from a belief state given an action (Algorithm 6.7).

Other approaches attempt to disperse the points throughout the reachable state space. For example, Algorithm 6.8 iterates through B, tries each available action, and adds the new belief states that are furthest from any of the points already in the set. There are many ways to measure distance between two belief states; the algorithm shown uses the L_1 distance metric, where the distance between b and b' is given by $\sum_s |b(s) - b'(s)|$.

Algorithm 6.5 Randomized point-based backup

```
function RandomizedPointBasedUpdate(B, Γ)
    Γ′ ← ∅
    B′ ← B
    repeat
        b ← belief point sampled uniformly at random from set B′
        α ← BackupBelief(b, Γ)
        if α⊤b ≥ U^Γ(b)
            Add α to Γ′
        else
            Add α′ = arg max_{α∈Γ} α⊤b to Γ′
        B′ ← {b ∈ B | U^{Γ′}(b) < U^Γ(b)}
    until B′ = ∅
    return Γ′
```

Algorithm 6.6 Expand belief points with random actions

```
function ExpandBeliefPoints(B)
    B′ ← B
    for b ∈ B
        a ← random action selected uniformly from action space A
        o ← SampleObservation(b, a)
        b′ ← UpdateBelief(b, a, o)
        Add b′ to B′
    return B′
```

Algorithm 6.7 Sample observation

```
function SampleObservation(b, a)
    s ∼ b
    s′ ← random state selected with probability T(s′ | s, a)
    o ← random observation selected with probability O(o | a, s′)
    return o
```

Algorithm 6.8 Expand belief points with exploratory actions

```
function ExpandBeliefPoints(B)
    B' ← B
    for b ∈ B
        for a ∈ A
            o ← SampleObservation(b, a)
            b_a ← UpdateBelief(b, a, o)
        a ← arg max_a min_{b'∈B'} Σ_s |b'(s) − b_a(s)|
        Add b_a to B'
    return B'
```

6.4.6 Linear Policies

As discussed in Section 6.2.2, the belief state in a problem with linear-Gaussian dynamics can be represented by a Gaussian distribution $\mathcal{N}(\mu_b, \Sigma_b)$. If the reward function is quadratic, as assumed in Section 4.4, then it can be shown that the optimal policy can be computed exactly offline. In fact, the solution is identical to the perfect observability case outlined in Section 4.4, but the μ_b computed by the Kalman filter is used in place of the true state. With each observation, we simply use the Kalman filter to update our μ_b, and then we matrix multiply μ_b with the policy matrix given in Section 4.4 to determine the optimal action.

6.5 Online Methods

Online methods determine the optimal policy by planning from the current belief state. The belief states reachable from the current state are typically small compared with the full belief space. Many online methods use a depth-first tree-based search up to some horizon. The time complexity of these online algorithms is generally exponential in the horizon. Although online methods require more computation per decision step during execution than offline approaches, online methods are sometimes easier to apply to high-dimensional problems.

6.5.1 Lookahead with Approximate Value Function

We can use the one-step lookahead strategy online to improve on a policy that has been computed offline. If b is the current belief state, then the one-step lookahead policy is given by

$$\pi(b) = \arg\max_a \left[R(b,a) + \gamma \sum_o P(o \mid b, a) U(\textsc{UpdateBelief}(b, a, o)) \right], \tag{6.34}$$

where U is an approximate value function. This approximate value function may be represented by alpha vectors computed offline using strategies such as QMDP, fast informed bound, or point-based value iteration discussed earlier. Experiments have shown that for many problems, a one-step lookahead can significantly improve performance over the base offline strategy.

The approximate value function may also be estimated by sampling from a rollout policy as introduced in Section 4.6.4 but with modifications to handle partial observability as shown in Algorithm 6.9. The generative model G returns a sampled next state s' and reward r given state s and action a. This algorithm uses a single rollout policy π_0, but we can use a set of rollout policies and evaluate them in parallel. The policy that results in the largest value is the one that is used to estimate the value at that particular belief state.

Algorithm 6.9 Rollout evaluation

```
function ROLLOUT(b, d, π_0)
    if d = 0
        return 0
    a ~ π_0(b)
    s ~ b
    (s', o, r) ~ G(s, a)
    b' ← UPDATEBELIEF(b, a, o)
    return r + γROLLOUT(b', d − 1, π_0)
```

An alternative to summing over all possible observations in Equation (6.34) is to use sampling. We can generate n observations for each action through independent calls to SAMPLEOBSERVATION(b, a) and then compute

$$\pi(b) = \arg\max_a \left[R(b,a) + \gamma \frac{1}{n} \sum_{i=1}^{n} U(\text{UPDATEBELIEF}(b, a, o_{a,i})) \right]. \tag{6.35}$$

This strategy is particularly useful when the observation space is large.

6.5.2 Forward Search

The one-step lookahead strategy can be extended to look an arbitrary depth into the future. Algorithm 6.10 defines the function SELECTACTION(b, d, U), which returns the pair (a^*, u^*) defining the best action and the expected utility, given the current belief b, depth d, and approximate value function U.

When $d = 0$, no action is able to be selected, and so a^* is NIL and the utility is $U(b)$. When $d > 0$, we compute the value for every available action and return the best action and its associated value. To compute the value for an action a, we evaluate

$$R(b,a) + \gamma \sum_o P(o \mid b,a) U_{d-1}(\text{UpdateBelief}(b,a,o)). \tag{6.36}$$

In the equation above, $U_{d-1}(b')$ is the expected utility returned by the recursive call SelectAction$(b', d-1)$. The complexity is given by $O(|A|^d |O|^d)$. Algorithm 6.10 can be modified to sample n observations instead of enumerating all $|O|$ of them, similar to what is done in Equation (6.35). The complexity then becomes $O(|A|^d n^d)$.

Algorithm 6.10 Forward search online policy

```
function SelectAction(b, d)
    if d = 0
        return (NIL, U(b))
    (a*, u*) ← (NIL, −∞)
    for a ∈ A
        u ← R(b, a)
        for o ∈ O
            b' ← UpdateBelief(b, a, o)
            (a', u') ← SelectAction(b', d − 1)
            u ← u + γ P(o | b, a) u'
        if u > u*
            (a*, u*) ← (a, u)
    return (a*, u*)
```

6.5.3 Branch and Bound

The branch and bound technique originally introduced in Section 4.6.2 in the context of MDPs can be easily extended to POMDPs. As with the POMDP version of forward search, we have to iterate over the observations and update beliefs—otherwise the algorithm is nearly identical to the MDP version. Again, the ordering of the actions in the for loop is important. To prune as much of the search space as possible, the actions should be enumerated such that their upper bounds decrease in value. In other words, action a_i comes before a_j if $\overline{U}(b,a_i) \geq \overline{U}(b,a_j)$.

We can use QMDP or the fast informed bound as the upper bound function $\overline{U}$. For the lower bound function $\underline{U}$, we can use the value function associated with a *blind policy*, which selects the same action regardless of the current belief state. This value

function can be represented by a set of $|A|$ alpha vectors, which can be computed as follows:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a)\alpha_a^{(k)}(s'), \tag{6.37}$$

where $\alpha_a^{(0)} = \min_s R(s,a)/(1-\gamma)$. Equation (6.37) is similar to the QMDP equation in Equation (6.26) except that it does not have a maximization over the alpha vectors on the right-hand side.

So long as $\underline{U}$ and $\overline{U}$ are true lower and upper bounds, the result of the branch and bound algorithm will be the same as the forward search algorithm with $\underline{U}$ as the approximate value function. In practice, branch and bound can significantly reduce the amount of computation required to select an action. The tighter the upper and lower bounds, the more of the search space branch and bound can prune. However, in the worst case, the complexity of branch and bound is no better than that of forward search.

Algorithm 6.11 Branch and bound online policy

```
function SelectAction(b, d)
    if d = 0
        return (nil, U(b))
    (a*, u) ← (nil, −∞)
    for a ∈ A
        if Ū(b, a) ≤ u
            return (a*, u)
        u ← R(b, a)
        for o ∈ O
            b′ ← UpdateBelief(b, a, o)
            (a′, u′) ← SelectAction(b′, d − 1)
            u ← u + γ P(o | b, a) u′
        if u > u
            (a*, u) ← (a, u)
    return (a*, u)
```

6.5.4 Monte Carlo Tree Search

The Monte Carlo tree search approach for MDPs can be extended to POMDPs, as outlined in Algorithm 6.12. The input to the algorithm is a belief state b, depth d, and rollout policy π_0. The main difference between the POMDP algorithm and the MDP algorithm in Section 4.6.4 is that the counts and values are associated with *histories* instead of states. A history is a sequence of past observations and actions. For example,

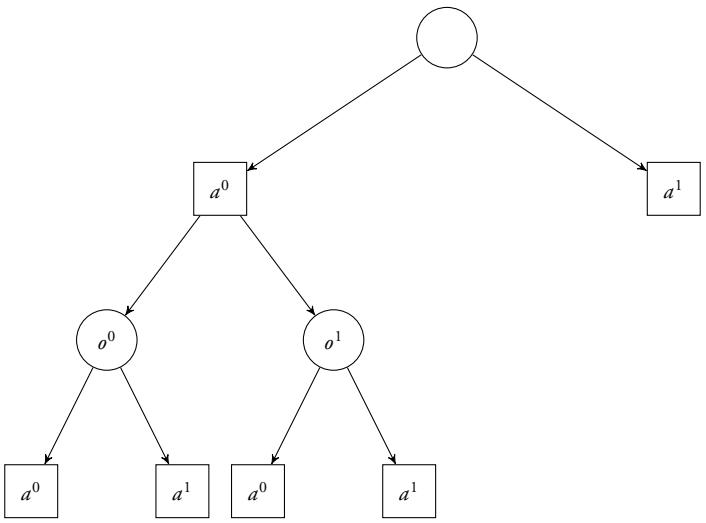


Figure 6.7 Example Monte Carlo tree search history tree.

if we have two actions a^0 and a^1 and two observations o^0 and o^1, then a possible history could be the sequence $h = a^0 o^1 a^1 o^1 a^0 o^0$. During the execution of the algorithm, we update the value estimates $Q(h, a)$ and counts $N(h, a)$ for a set of history-action pairs.

The histories associated with Q and N may be organized in a tree like the one in Figure 6.7. The root node represents the empty history starting from the initial belief state b. During the execution of the algorithm, the tree structure expands. The layers of the tree alternate between action nodes and observation nodes. Associated with each action node are values $Q(h, a)$ and $N(h, a)$, where the history is determined by the path to the root node. In the algorithm, $N(h) = \sum_a N(h, a)$.

As with the MDP version, the Monte Carlo tree search algorithm is an anytime algorithm. The loop in SelectAction(b, d) can be terminated at any time, and some solution will be returned. It has been shown that, with a sufficient number of iterations, the algorithm converges to the optimal action.

Prior knowledge can be incorporated into this algorithm through the choice of initialization parameters N_0 and Q_0 as well as the rollout policy. The algorithm does not need to be reinitialized with each decision. The history tree and associated counts and value estimates can be maintained between calls. The observation node associated with the selected action and actual observation becomes the root node at the next time step.

Algorithm 6.12 Monte Carlo tree search

```
function SelectAction(b, d)
    h ← ∅
    loop
        s ~ b
        Simulate(s, h, d)
    return arg max_a Q(h, a)
function Simulate(s, h, d)
    if d = 0
        return 0
    if h ∉ T
        for a ∈ A(s)
            (N(h, a), Q(h, a)) ← (N_0(h, a), Q_0(h, a))
        T = T ∪ {h}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(h, a) + c √(log N(h) / N(h, a))
    (s′, o, r) ~ G(s, a)
    q ← r + γ Simulate(s′, hao, d − 1)
    N(h, a) ← N(h, a) + 1
    Q(h, a) ← Q(h, a) + (q − Q(h, a)) / N(h, a)
    return q
```

6.6 Summary

- POMDPs are MDPs over belief states.
- POMDPs are difficult to solve exactly in general but can often be approximated well.
- Policies can be represented as alpha vectors.
- Large problems can often be solved online.

6.7 Further Reading

It was observed in the 1960s that POMDPs can be transformed into MDPs over belief states [1]. Belief state updating in discrete state spaces is a straightforward application of Bayes' rule. A thorough introduction to the Kalman filter and its variants is provided in *Estimation with Applications to Tracking and Navigation* by Bar-Shalom, Li, and Kirubarajan [2]. Arulampalam et al. provide a tutorial on particle filters [3]. *Probabilistic Robotics* by Thrun, Burgard, and Fox discusses different methods for belief updating in the context of robotic applications [4].

Exact solution methods for POMDPs were originally proposed by Smallwood and Sondik [5] and Sondik [6] in the 1970s. There are several surveys of early work on POMDPs [7]–[9]. Kaelbling, Littman, and Cassandra present techniques for identifying dominated plans to improve the efficiency of exact solution methods [10]. Computing exact solutions to POMDPs is intractable in general [11], [12].

Approximation methods for POMDPs have been the focus of considerable research recently. Hauskrecht discusses the relationship between QMDP and the fast informed bound and presents empirical results [13]. Offline approximate POMDP solution algorithms have focused on point-based approximation techniques, as surveyed by Shani, Pineau, and Kaplow [14]. The point-based value iteration (PBVI) algorithm was proposed by Pineau, Gordon, and Thrun [15]. There are other, more involved, point-based value iteration algorithms. Two of the best algorithms, Heuristic Search Value Iteration (HSVI) [16], [17] and Successive Approximations of the Reachable Space under Optimal Policies (SARSOP) [18], involve building search trees through belief space and maintaining upper and lower bounds on the value function. The randomized point-based value iteration algorithm discussed in Section 6.4.4 is based on the Perseus algorithm presented by Spaan and Vlassis [19]. Controller-based solutions have also been explored for concisely representing policies for infinite-horizon problems and removing the need for belief updating during execution [20], [21]. Several online solution methods are surveyed by Ross et al. [22]. Silver and Veness present a Monte Carlo tree search algorithm for POMDPs called Partially Observable Monte Carlo Planning (POMCP) [23].

References

1. K.J. Åström, "Optimal Control of Markov Processes with Incomplete State Information," *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174–205, 1965. DOI: 10.1016/0022-247X(65)90154-X.
2. Y. Bar-Shalom, X.R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*. New York: Wiley, 2001.
3. M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear / Non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002. DOI: 10.1109/78.978374.
4. S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Cambridge, MA: MIT Press, 2006.
5. R.D. Smallwood and E.J. Sondik, "The Optimal Control of Partially Observable Markov Processes Over a Finite Horizon," *Operations Research*, vol. 21, no. 5, pp. 1071–1088, 1973.
6. E.J. Sondik, "The Optimal Control of Partially Observable Markov Processes Over the Infinite Horizon: Discounted Costs," *Operations Research*, vol. 26, no. 2, pp. 282–304, 1978.
7. C.C. White III, "A Survey of Solution Techniques for the Partially Observed Markov Decision Process," *Annals of Operations Research*, vol. 32, no. 1, pp. 215–230, 1991. DOI: 10.1007/BF02204836.
8. W.S. Lovejoy, "A Survey of Algorithmic Methods for Partially Observed Markov Decision Processes," *Annals of Operations Research*, vol. 28, no. 1, pp. 47–65, 1991. DOI: 10.1007/BF02055574.
9. G.E. Monahan, "A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms," *Management Science*, vol. 28, no. 1, pp. 1–16, 1982.
10. L.P. Kaelbling, M.L. Littman, and A.R. Cassandra, "Planning and Acting in Partially Observable Stochastic Domains," *Artificial Intelligence*, vol. 101, no. 1–2, pp. 99–134, 1998. DOI: 10.1016/S0004-3702(98)00023-X.
11. C. Papadimitriou and J. Tsitsiklis, "The Complexity of Markov Decision Processes," *Mathematics of Operation Research*, vol. 12, no. 3, pp. 441–450, 1987. DOI: 10.1287/moor.12.3.441.
12. O. Madani, S. Hanks, and A. Condon, "On the Undecidability of Probabilistic Planning and Related Stochastic Optimization Problems," *Artificial Intelligence*, vol. 147, no. 1-2, pp. 5–34, 2003. DOI: 10.1016/S0004-3702(02)00378-8.

13. M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000. DOI: 10.1613/jair.678.

14. G. Shani, J. Pineau, and R. Kaplow, "A Survey of Point-Based POMDP Solvers," *Autonomous Agents and Multi-Agent Systems*, pp. 1–51, 2012. DOI: 10.1007/s10458-012-9200-2.

15. J. Pineau, G.J. Gordon, and S. Thrun, "Anytime Point-Based Approximations for Large POMDPs," *Journal of Artificial Intelligence Research*, vol. 27, pp. 335–380, 2006. DOI: 10.1613/jair.2078.

16. T. Smith and R.G. Simmons, "Heuristic Search Value Iteration for POMDPs," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2004.

17. ——, "Point-Based POMDP Algorithms: Improved Analysis and Implementation," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.

18. H. Kurniawati, D. Hsu, and W.S. Lee, "SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces," in *Robotics: Science and Systems*, 2008.

19. M.T.J. Spaan and N.A. Vlassis, "Perseus: Randomized Point-Based Value Iteration for POMDPs," *Journal of Artificial Intelligence Research*, vol. 24, pp. 195–220, 2005. DOI: 10.1613/jair.1659.

20. P. Poupart and C. Boutilier, "Bounded Finite State Controllers," in *Advances in Neural Information Processing Systems (NIPS)*, 2003.

21. C. Amato, D.S. Bernstein, and S. Zilberstein, "Solving POMDPs Using Quadratically Constrained Linear Programs," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

22. S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, "Online Planning Algorithms for POMDPs," *Journal of Artificial Intelligence Research*, vol. 32, pp. 663–704, 2008. DOI: 10.1613/jair.2567.

23. D. Silver and J. Veness, "Monte-Carlo Planning in Large POMDPs," in *Advances in Neural Information Processing Systems (NIPS)*, 2010.

7

Cooperative Decision Making

Christopher Amato

Solving problems in which a group of agents must operate collaboratively in sequential environments is an important challenge. As the construction of agents (e.g., robots, sensors, software agents) becomes less costly, more agents can be deployed, but for a team to achieve its full potential, each agent must reason about the others. In this chapter, we discuss models in which agents may have uncertainty about both the state of the environment and the choices of the other agents. Agents seek to optimize a shared objective function but must develop plans of action based on a partial view of the environment. We describe modeling this problem as a decentralized partially observable Markov decision process (Dec-POMDP) and discuss the complexity and salient properties of this model. We also provide an overview of exact and approximate solution methods for Dec-POMDPs, discuss the use of communication between agents in this model, and describe notable subclasses with additional modeling assumptions and reduced complexity.

7.1 Formulation

Multiagent systems can be modeled in a centralized way using MDPs and POMDPs by requiring all agent information and decision making to be centralized at each step. However, many problems require decentralized execution. The Dec-POMDP model is an extension of the MDP and POMDP models that provides decentralized policies for each agent. In a Dec-POMDP, the dynamics of the system and the objective function depend on the actions of all agents, but each agent must make decisions based on local information. We first present the model, discuss an example problem, and outline two forms of solution representations.

7.1.1 Decentralized POMDPs

A Dec-POMDP is defined by the following:

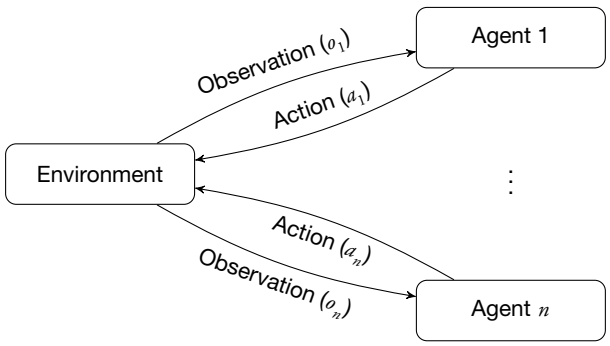


Figure 7.1 Dec-POMDP structure with n agents.

- I, a finite set of agents,
- S, a finite set of states with designated initial state distribution b_0,
- A_i, a finite set of actions for each agent i,
- T, transition probability function $T(s' \mid s, \mathbf{a})$ that specifies the probability of transitioning from state s to s' when action $\mathbf{a}$ is taken by the agents,
- R, a reward function $R(s, \mathbf{a})$ that specifies the immediate reward for being in state s and taking the actions $\mathbf{a}$,
- Ω_i, a finite set of observations for each agent i, and
- O, observation model $O(\mathbf{o} \mid s', \mathbf{a})$ that specifies the probability of observing $\mathbf{o}$ in state s' given action $\mathbf{a}$.

As shown in Figure 7.1, a Dec-POMDP involves multiple agents that operate under uncertainty on the basis of different streams of observations. Like an MDP or a POMDP, a Dec-POMDP unfolds over a finite or an infinite sequence of steps. At each step, every agent chooses an action based purely on its local observations, resulting in an immediate reward for the set of agents and an observation for each individual agent. Because the state is not directly observed, it may be beneficial for each agent to remember its observation history. Unlike in POMDPs, in Dec-POMDPs, it is not always possible to calculate an estimate of the system state (a belief state) from the observation history of a single agent (as we will discuss in Section 7.2.1).

A *joint policy* is a set of policies, one for each agent in the problem. A *local policy* for an agent is a mapping from local observation histories to actions. The objective in a Dec-POMDP is to find a joint policy that maximizes expected utility. As with MDPs and POMDPs, we can define utility in different ways, such as the sum of rewards over a finite horizon or the discounted sum of rewards over an infinite horizon (Section 4.1.2).

A generalization of the Dec-POMDP formulation involves specifying independent reward functions for the various agents. If the agents are to maximize their own accumulation of reward, then the problem becomes a partially observable stochastic game

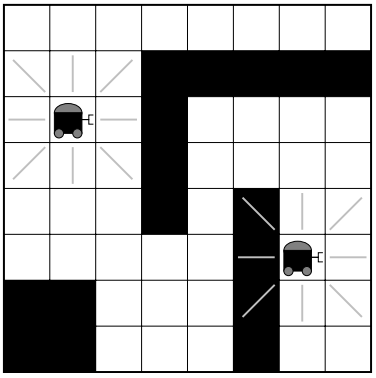


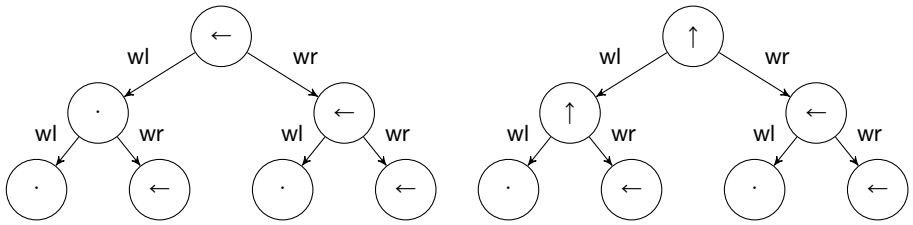
Figure 7.2 Robot navigation problem with two agents.

(POSG). Such problems require a game-theoretic treatment (similar to what was introduced in Section 3.3 for single-shot decisions) and are significantly more difficult to analyze.

7.1.2 Example Problem

One set of domains that can be modeled as Dec-POMDPs is robot navigation and exploration problems. A simple grid-based robot navigation problem is shown in Figure 7.2. The states in this problem correspond to the positions of both robots. The actions are up, down, left, right, and stay in place. The movement actions move the agent one grid cell in the desired direction with probability 0.6 or in one of the other directions or current cell with probability 0.1 each. Movements into a wall result in the robot's staying in place. Choosing to stay in place always keeps the agent in the current location. We assume perfect observation of the grid cells immediately surrounding the agent (as indicated by the gray lines in the figure). As a result, each robot can observe the wall configurations in surrounding squares but not its own actual location. The objective is for the agents to meet as quickly as possible. When both agents occupy the same square, a reward of one is received; otherwise, no reward is received. The initial state is the one shown in the figure.

There are three types of uncertainty in this problem: uncertainty about action outcomes (as in an MDP), uncertainty about sensor information (as in a POMDP), and uncertainty about the information of the other agents. Although the agents observe the surrounding grid squares and can narrow down their own possible locations, the observations usually provide no information about the choices or location of the other agent. As a result, optimal algorithms will often consider all the possible choices and locations for the other agents when generating a solution. As discussed later, centralized



(a) Agent 1. (b) Agent 2.

Figure 7.3 Policy tree representation for a two-agent Dec-POMDP.

belief states as used by a POMDP are no longer possible, and solving a Dec-POMDP becomes much more difficult. A solution to this problem would be for each agent to move toward a central location and, if the other agent is not seen, continue to some locations the other agent could be. If the other agent is still not seen, then the agent could move to a location that was agreed on in the solution process and wait for the other agent to arrive. The solution method would produce the policy of which movements an agent should make after seeing different observation histories, optimizing these choices over the uncertainty.

7.1.3 Solution Representations

For finite-horizon problems, local policies can be represented by policy trees. An example is shown in Figure 7.3. Such trees are similar to the policy trees for POMDPs, but now each agent possesses its own tree that is independent of the other agents' trees. To make this more concrete, we consider a simplified version of the example problem above in a 2×2 grid without obstacles. Agent 1 starts in the top right, and agent 2 starts in the bottom left. Actions are represented by arrows or the stop symbol, $\cdot$, (each agent can move in the given direction or stay where it is). Observations are labeled "wl" and "wr" for seeing a wall on the left or the right, respectively. In this representation, an agent takes the action defined at the root node and then, after seeing an observation, chooses the next action that is defined by the respective branch. This sequence continues until the action at a leaf node is executed. For example, agent 1 would first move left, and if a wall is seen on the right, then the agent would move left again. If a wall is now seen on the left, then the agent does not move on the final step. A policy tree is a record of the entire local history for an agent up to some fixed horizon. Because each tree is independent of the others, it can be executed in a decentralized manner. The resulting policies allow the agents to meet in the top left square quickly and with high probability.

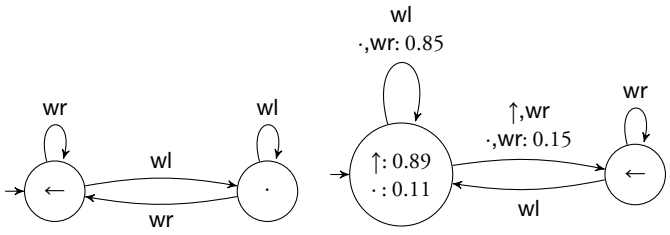


Figure 7.4 Stochastic controller representation for a two-agent Dec-POMDP.

These trees can be evaluated by summing the rewards at each step weighted by the likelihood of transitioning to a given state and observing a given set of observations. For a set of agents, the value of trees $\mathbf{q}$ while starting at state s is given recursively by

$$U(\mathbf{q}, s) = R(\mathbf{a_q}, s) + \sum_{s', \mathbf{o}} P(s' \mid \mathbf{a_q}, s) P(\mathbf{o} \mid \mathbf{a_q}, s') U(\mathbf{q_o}, s'), \tag{7.1}$$

where $\mathbf{a_q}$ are the actions defined at the root of trees $\mathbf{q}$, and $\mathbf{q_o}$ are the subtrees of $\mathbf{q}$ that are visited after $\mathbf{o}$ have been seen.

Although this representation is useful for finite-horizon problems, infinite-horizon problems would require trees of infinite height. Another option is to condition action selection on some internal memory state. These solutions can be represented as a set of local finite-state controllers (seen in Figure 7.4). Again, these controllers are similar to those used by POMDPs except each agent possesses its own independent controller.

The controllers operate in a similar way to the policy trees. There is a designated initial node, and following the action selection at that node, the controller transitions to the next node depending on the observation. This process continues for the infinite steps of the problem. We can also consider stochastic controllers, which choose actions and transitions stochastically, as they are able to produce higher quality solutions than deterministic controllers with the same number of nodes. Throughout this chapter, controller states will be referred to as nodes to help distinguish them from system states.

An example of two-node stochastic controllers for the 2×2 version of the example problem can be seen Figure 7.4. Agent 2 begins at node 1, moving up with probability 0.89 and staying in place with probability 0.11. If the agent stayed in place and a wall is then seen on the left (observation "wl"), on the next step, then the controller would transition to node 1, and the agent would use the same distribution of actions again. If a wall was seen on the right instead (observation "wr"), then there is a 0.85 probability that the controller will transition back to node 1 and a 0.15 probability that the controller will transition to node 2 for the next step. The resulting policy again allows the agents to

meet quickly and with high probability in the top left square. The finite-state controller allows an infinite-horizon policy to be represented compactly by remembering some aspects of the agent's history without representing the entire local history.

We can evaluate the joint policy by beginning at the initial node and transitioning through the controller according to the actions taken and observations seen. We define the probability an action a_i will be taken in node q_i by agent i to be $P(a_i \mid q_i)$. We also have $P(q_i' \mid q_i, a_i, o_i)$ represent the probability that the controller transitions to node q_i' given that the controller is currently in q_i, takes action a_i, and observes o_i. The value for starting in nodes $\mathbf{q}$ and at state s with action selection and node transition probabilities for each agent, i, is given by the following Bellman equation:

$$U(\mathbf{q}, s) = \sum_{\mathbf{a}} \left(\prod_i P(a_i \mid q_i) \left[R(s, \mathbf{a}) + \gamma \sum_{s', \mathbf{o}, \mathbf{q}'} P(s' \mid \mathbf{a}, s) O(\mathbf{o} \mid s', \mathbf{a}) \prod_j P(q_j' \mid q_j, a_j, o_j) U(\mathbf{q}', s') \right] \right). \quad (7.2)$$

Note that the values (for either the trees or controllers) can be calculated offline in order to determine a policy for each agent that can then be executed online in a decentralized manner. In fact, as we will discuss below, many algorithms consider this offline planning scenario in which the solution (trees or controllers) is generated offline in a centralized manner and then the policy is executed online in a decentralized manner.

7.2 Properties

The decentralized nature of Dec-POMDPs makes them fundamentally different from POMDPs. We explain some of these differences, discuss the complexity of the general Dec-POMDP model, and describe an extension of the concept of belief states to multiagent problems.

7.2.1 Differences with POMDPs

In a Dec-POMDP, the decisions of each agent affect all the agents in the domain, but because of the decentralized nature of the model, each agent must choose actions based solely on local information. Because each agent receives a separate observation that does not usually provide sufficient information to efficiently reason about the other agents, solving a Dec-POMDP optimally becomes difficult. Each agent may receive a different piece of information that does not allow a common state estimate or any estimate of the other agents' decisions to be calculated. For example, in the robot navigation example problem in Figure 7.2, even though each agent knows the initial location of the other, agent 1's observations (until it becomes adjacent) give it no information about agent 2's

choice of action or location. Therefore, while it may be possible to limit the possible locations the other agent may be in (such as those not in surrounding grid cells), it is usually not possible to generate an estimate of the system state. Exceptions to this are when observations provide this information (e.g., when the other agent is seen) or the policies of the other agents are known (as discussed later).

State estimates are crucial in single-agent problems because they allow the agent's history to be summarized concisely (as belief states), but they are not generally available in Dec-POMDPs. The lack of state estimates (and thus the lack of a concise sufficient statistic) requires agents to remember whole action and observation histories in order to act optimally; therefore, Dec-POMDPs cannot be transformed into belief state MDPs, and we must use a different set of tools to solve them.

7.2.2 Dec-POMDP Complexity

The difference between Dec-POMDPs and POMDPs is seen in the complexity of the finite-horizon problem. A Dec-POMDP with at least two agents is *NEXP-complete*, a category of problem that may require doubly exponential time in practice. This complexity is in contrast to the MDP (P-complete) and the POMDP (PSPACE-complete). As in solving an infinite-horizon POMDP, optimally solving an infinite-horizon Dec-POMDP is *undecidable* because it may require infinite resources (infinitely sized controllers), but ϵ-optimal solutions can be found with finite time and memory. These complexity differences show that introducing multiple decentralized agents causes Dec-POMDPs to be significantly more difficult to solve than POMDPs. The intuition behind this complexity is that agents must consider the possible choices of all other agents in addition to the state and action uncertainty present in order to produce an optimal policy.

7.2.3 Generalized Belief States

As mentioned above, from an agent's perspective, not only is there uncertainty about the state, but there may also be uncertainty about the policies of the other agents. If we consider the possible policies of the other agents as part of the state of the system, then we can form a *generalized belief state* (sometimes called a multiagent belief state). Agents can also consider the generalized belief space that includes all possible distributions over states of the system and policies of the other agents. In a two-agent situation, the value of an agent's policy p at a given generalized belief state b_G is given by

$$U(p, b_G) = \sum_{q,s} b_G(s, q) U(p, q, s), \tag{7.3}$$

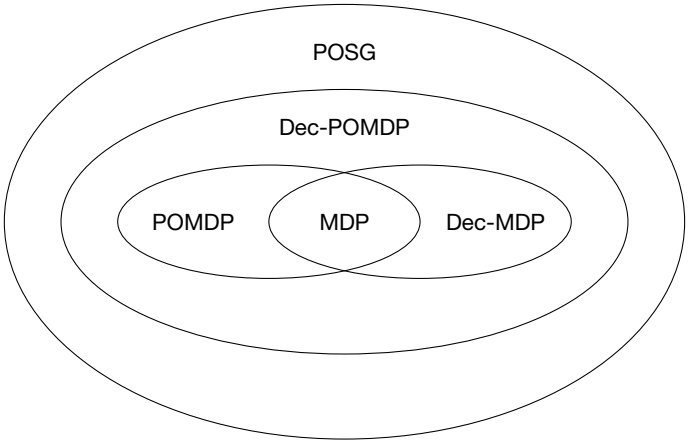


Figure 7.5 Subclasses and superclasses of Dec-POMDPs.

where q represents a policy of the other agent. If all other agents have known policies, then the generalized belief state is the same as a POMDP belief state, and we can solve the Dec-POMDP for the remaining agent as a POMDP (the other agents can be thought of as part of the environment). Unfortunately, when we solve a Dec-POMDP, probability distributions for other agent policies are generally not known. As a result, the generalized belief state cannot usually be calculated. Nevertheless, the idea of the possible policies of the other agents and the generalized belief space can be used in the decision-making process.

7.3 Notable Subclasses

Because of the high worst-case complexity of Dec-POMDPs, a number of Dec-POMDP subclasses have been explored that may be more tractable theoretically or in practice. This section discusses a few of these, including the Dec-MDP, network distributed POMDP (ND-POMDP), and multiagent MDP (MMDP). The relationships among Dec-POMDPs, POSGs, POMDPs, MDPs, and Dec-MDPs are shown in Figure 7.5.

7.3.1 Dec-MDPs

A *decentralized Markov decision process* (Dec-MDP) is a Dec-POMDP with joint full observability. In other words, if the observations of all the agents are combined, then the state of the environment is known exactly. A common example of a Dec-MDP is a problem in which the state consists of the locations of a set of robots and each agent observes its own location perfectly. Therefore, if all these observations are combined, then the locations of all robots would be known. Note that (perhaps counterintuitively)

Dec-MDPs do include observations for each agent that are often noisy indicators of the state. The complexity of Dec-MDPs is the same as Dec-POMDPs; although the true state may be known if observations are shared, this sharing does not take place.

We now discuss factorization in the context of Dec-MDPs, but similar factorization can be done in full Dec-POMDPs. A *factored n-agent Dec-MDP* is a Dec-MDP in which the world state can be factored into $n+1$ components, $S = S_0 \times S_1 \times \ldots \times S_n$. The states in S_i are the *local states* associated with agent i. The S_0 component is a property of the environment and is not affected by any agent actions (and is sometimes omitted). For example, S_0 may be the location of a target in a target-tracking scenario. Similarly, an agent's local state, S_i, might consist of its location in a grid. A factored, n-agent Dec-MDP is said to be *locally fully observable* if each agent fully observes its own state component.

A factored, n-agent Dec-MDP is said to be *transition independent* if the state transition probabilities factorize as follows:

$$T(s' \mid s, \mathbf{a}) = T_0(s'_0 \mid s_0) \prod_i T_i(s'_i \mid s_i, a_i). \tag{7.4}$$

Here, $T_i(s'_i \mid s_i, a_i)$ represents the probability that the local state of agent i transitions from s_i to s'_i after executing action a_i. The unaffected state transition probability is denoted $T_0(s'_0 \mid s_0)$. The robot navigation problem is transition independent if the robots never affect each other (i.e., they do not bump into each other when moving and can share the same grid cell).

A factored, n-agent Dec-MDP is said to be *observation independent* if the observation probabilities factorize as follows:

$$O(\mathbf{o} \mid s, \mathbf{a}) = \prod_i O_i(o_i \mid s_i, a_i). \tag{7.5}$$

In the equation above, $O_i(o_i \mid s_i, a_i)$ represents the probability that agent i receives observation o_i in state s_i after executing action a_i. If the robots in the navigation problem cannot observe each other (due to working in different locations or lack of sensors), then the problem becomes observation independent.

A factored, n-agent Dec-MDP is said to be *reward independent* if

$$R(s, \mathbf{a}) = f\left(R_1(s_i, a_i), \ldots, R_n(s_i, a_i)\right), \tag{7.6}$$

with the constraint that f is some monotonically non-decreasing function. Such a function has the property that $x_i \leq x'_i$ if and only if

$$f(x_1, \ldots, x_i, \ldots, x_n) \leq f(x_1, \ldots, x'_i, \ldots, x_n). \tag{7.7}$$

Table 7.1 Complexity of finite-horizon Dec-MDP subclasses.

Independence	Complexity
Transitions, observations and rewards	P-complete
Transitions and observations	NP-complete
Any other subset	NEXP-complete

Under this assumption, the global reward is maximized by maximizing local rewards. Additive local rewards are often used in reward independent models, where

$$R(s, \mathbf{a}) = R_0(s_0) + \sum_i R_i(s_i, a_i). \tag{7.8}$$

Table 7.1 shows the complexity of different subclasses of Dec-MDPs. The simplest case results from having independent transitions, observations, and rewards. It is straightforward to see that, in this case, the problem can be decomposed into n separate MDPs, and their solution can then be combined. When only the transitions and observations are independent, the problem becomes NP-complete. Intuitively, the NP-completeness is because the other agents' policies do not affect an agent's state (only the reward attained at the set of local states). Because independent transitions and observations imply local full observability, an agent's observation history does not provide any additional information about its own state—it is already known. Similarly, an agent's observation history does not provide any additional information about the other agents' states because they are independent. As a result, optimal policies become mappings from local states to actions instead of mappings from observation histories (or local state histories as local states are locally fully observable in this case) to actions. All other combinations of independent transitions, observations, and rewards do not reduce the complexity of the problem, leaving it NEXP-complete in the worst case.

7.3.2 ND-POMDPs

A *networked distributed POMDP* (ND-POMDP) is a Dec-POMDP with transition and observation independence and a special reward structure. The reward structure is represented by a coordination graph or *hypergraph*. A hypergraph is a generalization of a graph in which an edge can connect any number of nodes. The nodes in the ND-POMDP hypergraph correspond to the various agents. The edges relate to interactions between the agents in the reward function. An ND-POMDP associates with each edge j in the hypergraph a reward component R_j that depends on the state and action components to which the edge connects. The reward function in an ND-POMDP is simply the sum of the reward components associated with the edges. This fact allows the value

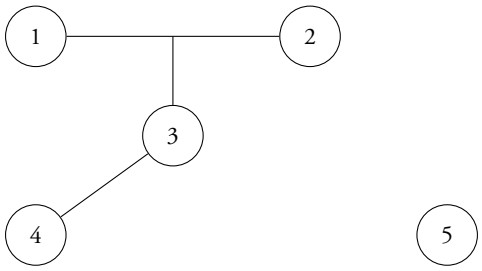


Figure 7.6 Example ND-POMDP structure.

function to be factorable in the same way, and solution methods can take advantage of this additional structure.

Figure 7.6 shows an example ND-POMDP structure with five agents. There are three hyper edges: one involving agents 1, 2, and 3; another involving agents 3 and 4; and another involving agent 5 on its own. The reward function decomposes as follows:

$$R_{123}(s_1, s_2, s_3, a_1, a_2, a_3) + R_{34}(s_3, s_4, a_3, a_4) + R_5(s_5, a_5). \tag{7.9}$$

Motivating domains for ND-POMDPs are typically sensor network and target tracking problems.

The ND-POMDP model is similar to the transition and observation independent Dec-MDP model, but it does not make the joint full observability assumption. Even if all observations are shared, the true state of the world may not be known. Furthermore, even with factored transitions and observations, a policy in an ND-POMDP is a mapping from observation histories to actions, unlike the transition and observation Dec-MDP case in which policies are mappings from local states to actions. The worst-case complexity remains the same as a full Dec-POMDP (NEXP-complete), but algorithms for ND-POMDPs are typically much more scalable in the number of agents. Scalability can increase as the hypergraph becomes less connected.

7.3.3 MMDPs

Another notable subclass is the *multiagent Markov decision process* (MMDP). In an MMDP, each agent is able to observe the true state, making the problem fully observable. Because each agent is able to observe the true state, an MMDP can be solved as an MDP (using some coordination mechanisms to ensure policies are consistent with each other) in polynomial time. An example based on the robot navigation problem above for the MMDP case would assume that each robot knows the location of the other robots at each step of the problem. The relationships among MMDPs, Dec-POMDPs, and single-agent models are shown in Figure 7.7.

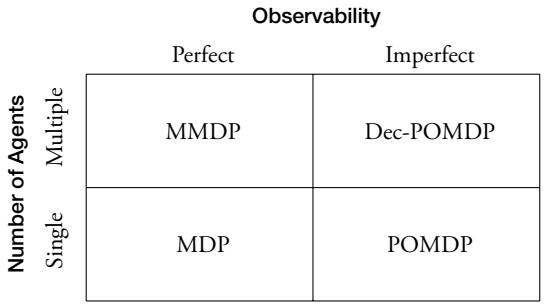


Figure 7.7 The relationship between Dec-POMDPs and other models.

7.4 Exact Solution Methods

This section discusses optimal algorithms for finite-horizon problems and ϵ-optimal algorithms for infinite-horizon problems. Finite-horizon methods can be used to solve infinite-horizon problems within any ϵ by using a horizon that is large enough to cause further action selection to contribute little to the overall value. POMDP methods are often scalable enough to produce such solutions, but Dec-POMDP methods are not. As a result, specific infinite-horizon methods for Dec-POMDPs, which use finite-state controllers as solution representations, are discussed. Most solution methods assume the problem is solved centrally offline to produce policies that can be executed online in a decentralized manner. These methods can also produce solutions in a decentralized manner with proper coordination mechanisms for the given algorithm.

7.4.1 Dynamic Programming

Algorithm 7.1 is a dynamic programming algorithm for optimally solving a finite-horizon Dec-POMDP. This approach constructs a set of trees for each agent (represented as π) from the last step until the first. At each step, the algorithm exhaustively generates all next step policies, evaluating them, and then pruning policies that are provably suboptimal for each agent i in turn until no more trees can be removed. This generation, evaluation, and pruning continues until the desired horizon T is reached. Dynamic programming in Dec-POMDPs is similar to the value iteration approach for POMDPs, except generalized belief states are used, resulting in a more complicated pruning step.

In the dynamic programming algorithm, a set of T-step policy trees, one for each agent, is generated from the bottom up. More precisely, on the last step of the problem, each agent will perform just a single action, which can be represented as a one-step policy tree. All possible actions for each agent are considered, and each combination of

Algorithm 7.1 Dynamic programming for Dec-POMDPs

```
1:  function DecDynamicProgramming(T)
2:      t ← 0
3:      π_t ← ∅
4:      repeat
5:          π_{t+1} ←ExhaustiveBackup(π_t)
6:          Compute V^{π_{t+1}}
7:          repeat
8:              π̂_{t+1} ← π_{t+1}
9:              for i ∈ I
10:                 π̂_{t+1} ←Prune(π̂_{t+1}, i)
11:                 Compute V^{π̂_{t+1}}
12:         until |π_t| = |π̂_t|
13:         t ← t + 1
14:     until t = T
15:     return π_t
```

these one-step trees is evaluated at each state of the problem by using Equation (7.1). Any action that has lower value than some other action for all states and possible actions of the other agents (the generalized belief space) is then pruned. All two-step policies are then generated for each agent by an *exhaustive backup* of the current trees. That is, for each action and each resulting observation, some one-step tree is chosen. If an agent has $|Q_i|$ one-step trees, $|A_i|$ actions, and $|\Omega_i|$ observations, then there will be $|A_i||Q_i|^{|\Omega_i|}$ two-step trees. After this exhaustive backup of next step trees is completed for each agent, pruning is again used to reduce their number. This process of backing up trees and pruning continues until horizon T is reached.

The resulting set of trees will contain an optimal solution for horizon T and any initial state of the system. This is because we considered all possible trees at each step and removed only those that were not useful no matter what policies the other agents chose at that step. This conservative pruning of trees ensures that we can safely remove trees that will be suboptimal at a given step.

The linear program used to determine whether a tree can be pruned can be represented as follows. For agent i's given tree q_i and variables ϵ and $x(\mathbf{q}_{-i}, s)$, maximize ϵ, given

$$\sum_{\mathbf{q}_{-i},s} x(\mathbf{q}_{-i},s)U(\hat{q}_i,\mathbf{q}_{-i},s)+\epsilon \leq \sum_{\mathbf{q}_{-i},s} x(\mathbf{q}_{-i},s)U(q_i,\mathbf{q}_{-i},s) \quad \forall\ \hat{q}_i \tag{7.10}$$

$$\sum_{\mathbf{q}_{-i},s} x(\mathbf{q}_{-i},s) = 1 \text{ and } x(\mathbf{q}_{-i},s) \geq 0 \quad \forall \mathbf{q}_{-i}, s. \tag{7.11}$$

This linear program determines whether agent i tree, q_i is dominated by comparing its value to other trees for that agent, $\hat{q}_i$. The variable $x(\mathbf{q}_{-i}, s)$ is a distribution over trees of the other agents and system states (the generalized belief state). We maximize ϵ while ensuring the variable x that represents the generalized belief state remains a proper probability distribution and test to see whether there is some distribution of trees that has higher or equal value for all states and policies of the other agents. Because there is always an alternative with at least equal value, regardless of system state and other agent policies, a tree q_i can be pruned if ϵ is non-positive.

The test for dominance is used for trees of a given horizon, ensuring that we consider all possible policies for a given horizon and remove those that are not useful no matter what policies are chosen by the other agents. If policies are removed, then the generalized belief space becomes smaller for the other agents (due to the removal of a possible policy to consider), and more policies may be able to be pruned. Thus, we can keep testing for dominated policies for each agent until no agent is able to prune any further policies. Lines 7–11 in Algorithm 7.1 show that pruning continues for all agents while any agent is able to remove any tree.

Unlike value iteration for POMDPs, the policy trees must be retained because it is no longer possible to recover the policy from the value function. Even with one-step lookahead in the POMDP case, we must calculate the belief state after an action is chosen. Because it is not possible to calculate a belief state in the Dec-POMDP case, and because the actions must be chosen based on local information, the optimal value function is not sufficient to generate a Dec-POMDP policy.

7.4.2 Heuristic Search

Instead of computing the policy trees from the bottom up, as is done by dynamic programming methods, we can construct the trees from the top down starting from a known initial state. This is the approach of multiagent A* (MAA*), which is an optimal algorithm built on heuristic search techniques. The search is conducted by using an upper bound on the value of partially defined policies (using POMDP or MDP solutions) and then choosing the partial joint policy to expand in a best-first ordering. Actions are then added, and an upper bound on this new partial joint policy is found. Again, the highest-valued partial joint policy is chosen, and another action choice is fixed. This process continues until a fully defined joint policy is found that has value higher than any of the partial joint policies. Algorithm 7.2 outlines the approach.

We can grow a joint policy in a top-down fashion by first considering the possible actions each agent can take to begin its policies. MAA* considers all possible combinations of actions that could be taken at that step and constructs a search tree that has these combinations as separate search nodes. In the worst case, a search tree could be constructed that contains all possible policies for each agent by considering all possible

Algorithm 7.2 Multiagent A*

```
1:  function MultiagentA*(T, b_0)
2:      V ← −∞
3:      L ← ×_i A_i
4:      repeat
5:          δ ← select(L)
6:          Δ' ← expand(δ)
7:          Δ^T ← fullPols(Δ')
8:          π ← bestFullPol(Δ^T)
9:          v ← valueOf(π)
10:         if v > V
11:             π* ← π
12:             V ← v
13:             prune(L, V)
14:         Δ' ← Δ' \ Δ^T
15:         L ← L \ δ ∪ Δ'
16:     until L is empty
17:     return π*
```

combinations of actions at the first step, followed by all possible combinations of actions for each observation at the second step, and so on. Because many of these policies may be suboptimal, a more intelligent approach for determining which choices to make is desired.

To assist in choosing better actions, MAA* incorporates a heuristic value for continuing execution for a given number of steps after a partial policy has been executed. That is, using a search based on the A* heuristic search method, we can estimate the value of a set of horizon T policies from a set of horizon t trees using the value of those trees up to horizon t calculated with Equation (7.1) and then some heuristic value for the value of continuing to horizon T.

More formally, we will call some set of horizon $t < T$ policy trees $\mathbf{q}$ a *partial policy* and a set of policies Δ^{T-t} a *completion policy*. A completion policy consists of appending $T - t$ policies to each leaf (last action) in the partial policy. Given a partial policy and a completion policy, we could evaluate them at a state s as follows:

$$U(\{\mathbf{q}^t, \Delta^{T-t}\}, s) = U(\mathbf{q}^t, s) + U(\Delta^{T-t} \mid \mathbf{q}^t, s). \tag{7.12}$$

Rather than explicitly considering completion policies to append, we can instead estimate the value a completion policy would produce. Therefore, we can produce an estimated value $\hat{U}$ that a partial policy has for the full horizon T as

$$\hat{U}(\mathbf{q}^t, s) = U(\mathbf{q}^t, s) + \hat{U}^{T-t}_{\mathbf{q}^t, s}, \tag{7.13}$$

where $\hat{V}_{\mathbf{q}^t,s}^{T-t}$ is the estimate for the value of continuing until horizon T after executing $\mathbf{q}$ starting at s.

There are many ways we can calculate $V_{\mathbf{q}^t,s}^{T-t}$, but to ensure an optimal policy is produced, we require the estimated value to be at least as high as the optimal value of continuing ($\hat{V} \geq V^*$). MAA* considers relaxing problem assumptions by using MDP or POMDP policies to produce the heuristic values. $V^*_{\text{POMDP}}(b)$ can be defined as the optimal value of a POMDP policy starting at belief b (i.e., a centralized solution in which all observations for all agents are known and actions can be chosen based on this centralized information). Similarly, $V^*_{\text{MDP}}(s)$ can be defined as the optimal value of an MDP policy starting at state s (i.e., a centralized policy assuming the state of the problem can be seen by all agents for the remainder of the problem). It can be shown that $V^*_{\text{Dec-POMDP}} \leq V^*_{\text{POMDP}} \leq V^*_{\text{MDP}}$, which intuitively holds because policies are less constrained as more information is available to the agents. MAA* then evaluates a partial policy by evaluating the policy until its given horizon t and then assumes either the belief state (in the V^*_{POMDP} case) or the state (in the V^*_{MDP} case) is known to the agents thereafter starting from the leaf nodes of $\mathbf{q}^t$.

The search in MAA* then chooses to grow the partial policy with the highest heuristic value. Growing a policy appends actions for each possible observation after the leaves of the current policy. The expansion of this search node adds an exponential number of nodes to the search tree at each step. Each of these new policies is then evaluated to produce an updated heuristic value.

A lower bound on the value of an optimal joint policy is also maintained. If growing a policy results in a horizon T policy, then the value of this policy is compared with the lower bound, updating it if the value of this policy is greater. Subsequently, any partial policy with estimated value less than the lower bound can then be pruned. This pruning can occur because the estimated value of a policy is an upper bound on its true value, indicating that it will never have value higher than the policy that produced the lower bound. The search completes when there are no more partial policies to expand. In this case, a set of $\mathbf{q}^T$ policies has been generated with higher value than the heuristic values of any partial policy.

We now describe Algorithm 7.2 in more detail. A lower bound value, $\underline{V}$, which represents the value of the best known joint policy, is initialized to negative infinity. An open list L, which represents the partial policies that are available to be expanded, is initialized to all joint actions for the agents. At each step, the partial joint policy (node in the search tree) with the highest estimated value $\hat{V}$ is selected. This partial policy is then expanded, generating all next step policies for that joint policy (all children in the search tree). This set is called Δ'. The set of full horizon T joint policies is now gathered into Δ^T, each of which is evaluated, and the one with the highest value v is chosen. If this value is higher than the value of the best known full policy, then the

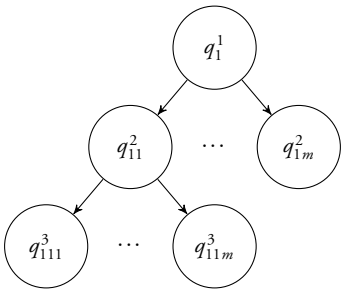


Figure 7.8 An example search tree in MAA*.

lower bound value and pointer to best policy are updated. Also, any partial policy in the open list that has estimated upper bound value (using the Q_{MDP} or Q_{POMDP} heuristics) lower than $\underline{V}$ is pruned. The full trees are removed from the set of expanded nodes, and the selected node is removed from the open list. The remaining expanded nodes are then added to the open list. This algorithm continues until the open list is empty and returns an optimal joint policy π^*.

An example of this search is shown in Figure 7.8. Each search node represents a joint policy for a given horizon. Subscripts represent the indices of the search node and each of the ancestor nodes in the search tree, whereas superscripts represent the horizon of the joint policy. Ancestors are indexed to signify that partial policies are shared for the appropriate horizon. Note that m represents the number of possible next-step joint policies, which is $|A_{max}|^{n|\Omega_{max}|}$, given the largest action and observation sets among the n agents A_{max}, Ω_{max}. The available partial policies (the open list) are represented by the leaves of the tree, whereas the nodes that have already been expanded are shown as internal nodes.

7.4.3 Policy Iteration

Because the infinite-horizon problem is undecidable, it may not be possible to generate a solution with the exact optimal value. As a consequence, methods focus on producing solutions within ϵ of the optimal value.

The policy iteration approach for Dec-POMDPs is similar to the finite-horizon dynamic programming algorithm, except finite-state controllers are used as policy representations (like the policy iteration approach for POMDPs). Starting from an initial controller for each agent, nodes are added at each step by using an exhaustive backup to produce any possible next-step policy for each agent. Pruning can then be completed to remove a node in an agent's controller if it has lower value than beginning in another node for all states of the system and all possible controllers of the other agents.

Algorithm 7.3 Policy iteration for Dec-POMDPs

```
1: function DecPolicyIteration(π_0, ε)
2:     t ← 0
3:     repeat
4:         π_{t+1} ← ExhaustiveBackup(π_t)
5:         Compute V^{π_{t+1}}
6:         repeat
7:             π̂_{t+1} ← π_{t+1}
8:             for i ∈ I
9:                 π̂_{t+1} ← Prune(π̂_{t+1}, i)
10:                UpdateController(π̂_{t+1}, i)
11:                Compute V^{π̂_{t+1}}
12:        until |π_t| = |π̂_t|
13:        t ← t + 1
14:    until γ^{t+1}|R_max| / (1−γ) ≤ ε
15:    return π_t
```

These exhaustive backups and pruning steps continue until the solution is provably within ϵ of an optimal solution. This algorithm can produce an ϵ-optimal policy in a finite number of steps. The details of policy iteration follow.

The policy iteration algorithm is shown in Algorithm 7.3. The input is an initial joint controller, π_0, and a parameter, ϵ. At each step, evaluation, backup, and pruning occur. The controller is evaluated using Equation (7.2). Next, an exhaustive backup is performed to add nodes to the local controllers. An exhaustive backup adds nodes to the local controllers for all agents at once. Similar to the finite-horizon case, for each agent i, $|A_i||Q_i|^{|\Omega_i|}$ nodes are added to the local controller, one for each one-step policy. Note that repeated application of exhaustive backups amounts to a brute force search in the space of deterministic policies. Continuing these exhaustive backups converges to optimality but is obviously quite inefficient.

To increase the efficiency of the algorithm, pruning takes place. Recall that planning takes place offline, so the controllers for each agent are known at each step, but agents will not know which node of their controller any of the other agents will be in during execution. As a result, pruning must be completed over the generalized belief space (using a linear program that is similar to that described for finite-horizon dynamic programming). That is, a node for an agent's controller can only be pruned if there is some combination of nodes that has higher value for all states of the system and at all nodes of the other agents' controllers. If this condition holds, then edges to the removed node are redirected to the dominating nodes. Because a node may be dominated by a distribution of other nodes, the resulting transitions may be stochastic rather than

deterministic. The updated controller is evaluated, and pruning continues until no agent can remove any further nodes.

In contrast to the single-agent case, there is no Bellman residual for testing convergence to ϵ-optimality. We resort to a simpler test 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. The algorithm terminates after iteration t if $\gamma^{t+1}|R_{\max}|/(1-\gamma) \leq \epsilon$. At this point, due to discounting, the value of any policy after step t is less than ϵ.

7.5 Approximate Solution Methods

In this section, we discuss approximate dynamic programming methods for finite-horizon problems and fixed-size controller-based methods for infinite-horizon problems. The algorithms in this section do not possess error bounds.

7.5.1 Memory-Bounded Dynamic Programming

The major limitation of dynamic programming approaches is the explosion of memory and time requirements as the horizon grows. This explosion occurs because each step requires generating and evaluating all joint policy trees (sets of policy trees for each agent) before performing the pruning step. Approximate dynamic programming techniques can mitigate this problem by keeping a fixed number of policy trees for each agent at each step, using a parameter called *MaxTrees*. This approach, called *memory-bounded dynamic programming* (MBDP), is outlined in Algorithm 7.4.

MBDP merges top-down (heuristic search) and bottom-up (dynamic programming) approaches by using heuristics (referred to as H in the algorithm) to choose top-down policies for each agent up to a given horizon. That is, dynamic programming proceeds as before, but after each backup, *MaxTrees* belief states are generated using heuristics to perform top-down sampling until the current step of dynamic programming is reached ($T - t$ if dynamic programming is at step t). It is then assumed that the resulting belief states are revealed to the agents, and only the trees that have the highest value at these belief states are retained. During execution, the belief state will not truly be revealed to the agents, but the hope is that high-valued decentralized policies can still be produced using this strategy. MBDP is conducted in an iterative fashion similar to traditional dynamic programming. For example, in a problem of horizon T, heuristic policies can be used for the first $T-1$ steps, and dynamic programming can find the best one-step trees (actions) for the resulting beliefs. Then, heuristic policies can be used for the first $T-2$ steps, and the *MaxTrees* one-step trees can be built up to two steps by dynamic programming. This process continues until *MaxTrees* horizon T trees have been constructed. Heuristics that have been used include MDP and random policies.

Algorithm 7.4 Memory-bounded dynamic programming (MBDP)

```
1:  function MBDP(MaxTrees, T, H)
2:      t ← 0
3:      π_t ← ∅
4:      repeat
5:          π_{t+1} ←ExhaustiveBackup(π_t)
6:          Compute V^{π_{t+1}}
7:          π̂_{t+1} ← ∅
8:          for k ∈ MaxTrees
9:              b_k ← GenerateBelief(H, T − t − 1)
10:             π̂_{t+1} ← π̂_{t+1} ∪ arg max_{π_{t+1}} V^{π_{t+1}}(b_k)
11:         t ← t + 1
12:         π_{t+1} ← π̂_{t+1}
13:     until t = T
14:     return π_t
```

Because only a fixed number of trees is retained at each step, the result is a suboptimal but much more scalable algorithm. In fact, because the number of policies retained at each step is bounded by *MaxTrees*, MBDP has time and space complexity linear in the horizon.

7.5.2 Joint Equilibrium Search

As an alternative to MBDP-based approaches, a method called joint equilibrium search for policies (JESP) utilizes alternating best response. JESP is shown in Algorithm 7.5. Initial policies are generated for all agents and then all but one are held fixed. The remaining agent can then calculate a best response (local optimum) to the fixed policies. This agent's policy then becomes fixed, and the next agent calculates a best response. This process continues until no agents change their policies. The result is a policy that is only locally optimal, but it may be high valued. JESP can be made more efficient by incorporating dynamic programming in the policy generation. Note that JESP can be thought of as finding a Nash equilibrium (as discussed in Section 3.3) in the cooperative game represented as the Dec-POMDP.

7.6 Communication

Communication can be implicitly represented using observations, but more explicit representations of communication have also been developed. Free and instantaneous communication is equivalent to centralization because all agents can have access to all observations at each step. When communication is delayed or has a cost, agents

Algorithm 7.5 Joint equilibrium search for strategies (JESP) without DP

```
1:  function JESP(π)
2:      k = 0
3:      π^k ← π
4:      repeat
5:          π^{k+1} ← π^k
6:          k ← k + 1
7:          for i ∈ I
8:              Compute V^{π^k}
9:              π^k(i) ← BestResponse(π^k, −i)
10:     until π^k = π^{k−1}
11:     return π^k
```

must reason about what and when to communicate. The complexity classes of the Dec-POMDP model with different types of observability and communication are shown in Table 7.2.

One extension of the general Dec-POMDP model to explicitly include communication is the decentralized partially observable Markov decision process with communication (Dec-POMDP-Com). A Dec-POMDP-Com is a Dec-POMDP, where each agent can send a message at each step. The reward at each step is a function of the joint state, joint action, and set of messages sent by the agents.

Producing an optimal solution of a Dec-POMDP-Com has the same complexity as solving the general Dec-POMDP model (NEXP-complete), and similar algorithms can be used to solve it as well. It can be shown that the optimal value of communicating an agent's history since the last communication will have value at least as high as any other set of possible messages, assuming fixed communication costs. As a result, many communication approaches assume that observation histories are used during communication. Many methods assume communication decisions are made by each agent separately, but some models assume agents can force all others to send their observation histories (allowing a POMDP belief state to be calculated).

A natural approach to solving a Dec-POMDP with communication is to produce a centralized (POMDP) plan and communicate when the observations received by an agent would cause it to choose a different action from the one prescribed by the centralized plan. This approach can be thought of as the agents agreeing on a policy before execution, and when it is noticed (by an agent's local observations) that this policy could be improved, communication takes place. This approach does not explicitly consider communication cost or delay but can limit the number of times communication takes place.

Table 7.2 The effect of communication on model complexity with different observability.

Observability	General Communication	Free Communication
Full	MMDP (P)	MMDP (P)
Joint Full	Dec-MDP (NEXP)	MMDP (P)
Partial	Dec-POMDP (NEXP)	MPOMDP (PSPACE)

7.7 Summary

- Dec-POMDPs can represent cooperative multiagent problems with action outcome uncertainty, observation uncertainty, and costly, lossy, or no communication.
- Like MDPs and POMDPs, Dec-POMDPs model sequential problems using a decision-theoretic approach that utilizes probabilities for uncertainties and values for outcomes.
- Unlike the single-agent models, each agent must make choices based solely on its own observation history.
- Solving finite-horizon Dec-POMDPs is NEXP-complete.
- Many subclasses of Dec-POMDPs are more efficient in theory or practice.
- Algorithms have been developed that can produce optimal or ϵ-optimal solutions for finite and infinite-horizon Dec-POMDPs.
- More scalable approximate algorithms have also been developed.
- Communication can also be explicitly modeled to assist in determining when and what to communicate to improve performance.

7.8 Further Reading

General overviews of Dec-POMDPs with additional algorithms and models are provided by Seuken and Zilberstein [1], Oliehoek [2], and Goldman and Zilberstein [3]. The complexity of the general Dec-POMDP was proven by Bernstein et al. [4]. A model similar to a Dec-POMDP is the multiagent team decision problem (MTDP) [5]. Finite-horizon dynamic programming is presented by Hansen, Bernstein, and Zilberstein [6], and infinite-horizon policy iteration is described by Bernstein et al. [7]. Multiagent A* is presented by Szer, Charpillet, and Zilberstein [8].

Dec-MDPs with independent transitions and observations were first discussed and solved by Becker et al. [9]. Dec-MDPs with event-driven interactions, considering limited transition dependence, were also discussed [10]. Allen and Zilberstein discussed a number of different modeling assumptions and resulting complexity [11]. ND-POMDPs were first proposed by Nair et al. [12], and MMDPs are presented by Boutilier [13]. Approximate finite-horizon algorithms MBDP and JESP are de-

scribed by Seuken and Zilberstein [14] and Nair et al. [15], respectively. Approximate infinite-horizon algorithms can be found in the literature [16]–[18].

The Dec-POMDP-Com model was presented by Goldman and Zilberstein [3]. The idea of using a centralized policy as a basis for communication was described by Roth, Simmons, and Veloso [19]. Forced synchronizing communication was discussed by Nair and Tambe [20].

Optimal finite-horizon algorithms have been improved in a few directions. Pruning can be used while conducting the backup to limit the subtrees that need to be considered [21]. Other work has also compressed policies rather than agent histories, improving the efficiency of the linear program used for pruning [22]. MAA* has been improved in several ways, including the incorporation of new heuristics and improved search [23]–[25]. Recently, two approaches have been developed for generating more concise sufficient statistics for offline planning, proving the sufficiency of distributions of states and joint histories [26] and converting Dec-POMDPs into POMDPs using decentralizable policies [27].

Approximate algorithms have also seen additional improvements. A number of approaches have improved on MBDP by compressing observations [28], replacing the exhaustive backup with a branch-and-bound search in the space of joint policy trees [29], as well as constraint optimization [30] and linear programming [31] to scale up the selection of the trees at each step. Additional fixed-size controller approaches have been developed that utilize an alternative representation of the controller as a Mealy machine to increase performance [32], employ expectation maximization for parameter optimization [33], or make use of more structured periodic controllers and an improved search technique [34].

Algorithms for subclasses have also been developed. The algorithm for solving transition and observation independent Dec-MDPs was the coverage set method [9]. Additional methods have also been developed to solve independent transition and observation Dec-MDPs more efficiently. These methods include a bilinear programming algorithm [35], a mixture of heuristic search and constraint optimization [36], and recasting the problem as a continuous MDP [37]. Optimal and approximate methods for solving ND-POMDPs have been proposed [12]. Other ND-POMDP methods have also been developed, such as those producing quality bounded solutions [38] and using finite-state controllers for agent policies [39].

Other models of communication are discussed in the literature [3], [5]. Communication has been studied in the context of locally fully observable Dec-MDPs with independent transitions and observations. This problem can be modeled with a cost of communication to receive the local states of the other agents and solved with a set of heuristic approaches [40]. Myopic communication, where an agent decides whether to communicate based on the assumption that communication can take place on this step or never again, has also been shown to work well in many cases [41]. Other types

of communication explored include stochastically delayed communication [42] and communication for online planning in Dec-POMDPs [43].

Factored models have been studied in the context of Dec-POMDPs. General factored models have been described and solved [44]. Witwicki and Durfee summarized different types of factored models and their complexity [45], and improved algorithms have been developed [46], [47].

In general, the research community has focused on planning methods for models that are similar to Dec-POMDPs, but a few learning methods have been explored. These include model-free reinforcement learning methods using gradient-based methods to improve the policies [48], [49] and using communication to learn solutions in ND-POMDPs [50].

Additional solution methods for general Dec-POMDPs have also been proposed. These include a mixed integer linear programming approach for Dec-POMDPs [51] and sampling methods [52], [53].

A number of applications have been studied, including multi-robot coordination in the form of space exploration rovers [54], helicopter flights [5] and navigation [55]–[57], load balancing for decentralized queues [58], network congestion control [59], multiaccess broadcast channels [60], network routing [61], sensor network management [12], target tracking [12], [62], and weather phenomena [63].

References

1. S. Seuken and S. Zilberstein, "Formal Models and Algorithms for Decentralized Control of Multiple Agents," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 2, pp. 190–250, 2008. DOI: 10.1007/s10458-007-9026-5.
2. F.A. Oliehoek, "Decentralized POMDPs," in *Reinforcement Learning: State of the Art*, M. Wiering and M. van Otterlo, eds., vol. 12, Berlin: Springer, 2012.
3. C.V. Goldman and S. Zilberstein, "Decentralized Control of Cooperative Systems: Categorization and Complexity Analysis," *Journal of Artificial Intelligence Research*, vol. 22, pp. 143–174, 2004. DOI: 10.1613/jair.1427.
4. D.S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein, "The Complexity of Decentralized Control of Markov Decision Processes," *Mathematics of Operations Research*, vol. 27, no. 4, pp. 819–840, 2002. DOI: 10.1287/moor.27.4.819.297.
5. D.V. Pynadath and M. Tambe, "The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models," *Journal of Artificial Intelligence Research*, vol. 16, pp. 389–423, 2002. DOI: 10.1613/jair.1024.
6. E.A. Hansen, D.S. Bernstein, and S. Zilberstein, "Dynamic Programming for Partially Observable Stochastic Games," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2004.

7. D.S. Bernstein, C. Amato, E.A. Hansen, and S. Zilberstein, "Policy Iteration for Decentralized Control of Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 34, pp. 89–132, 2009. DOI: 10.1613/jair.2667.
8. D. Szer, F. Charpillet, and S. Zilberstein, "MAA*: A Heuristic Search Algorithm for Solving Decentralized POMDPs," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.
9. R. Becker, S. Zilberstein, V. Lesser, and C.V. Goldman, "Solving Transition-Independent Decentralized Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 22, pp. 423–455, 2004. DOI: 10.1613/jair.1497.
10. R. Becker, V. Lesser, and S. Zilberstein, "Decentralized Markov Decision Processes with Event-Driven Interactions," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2004.
11. M. Allen and S. Zilberstein, "Complexity of Decentralized Control: Special Cases," in *Advances in Neural Information Processing Systems (NIPS)*, 2009.
12. R. Nair, P. Varakantham, M. Tambe, and M. Yokoo, "Networked Distributed POMDPS: A Synthesis of Distributed Constraint Optimization and POMDPs," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2005.
13. C. Boutilier, "Sequential Optimality and Coordination in Multiagent Systems," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.
14. S. Seuken and S. Zilberstein, "Memory-Bounded Dynamic Programming for DEC-POMDPs," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.
15. R. Nair, D. Pynadath, M. Yokoo, M. Tambe, and S. Marsella, "Taming Decentralized POMDPS: Towards Efficient Policy Computation for Multiagent Settings," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.
16. D. Szer and F. Charpillet, "An Optimal Best-First Search Algorithm for Solving Infinite Horizon DEC-POMDPs," in *European Conference on Machine Learning (ECML)*, 2005.
17. D.S. Bernstein, E.A. Hansen, and S. Zilberstein, "Bounded Policy Iteration for Decentralized POMDPs," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.
18. C. Amato, D.S. Bernstein, and S. Zilberstein, "Optimizing Fixed-Size Stochastic Controllers for POMDPs and Decentralized POMDPs," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 21, no. 3, pp. 293–320, 2010. DOI: 10.1007/s10458-009-9103-z.
19. M. Roth, R. Simmons, and M.M. Veloso, "Reasoning About Joint Beliefs for Execution-Time Communication Decisions," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2005.

20. R. Nair and M. Tambe, "Communication for Improving Policy Computation in Distributed POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2004.
21. C. Amato, J.S. Dibangoye, and S. Zilberstein, "Incremental Policy Generation for Finite-Horizon DEC-POMDPs," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.
22. A. Boularias and B. Chaib-draa, "Exact Dynamic Programming for Decentralized POMDPs with Lossless Policy Compression," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.
23. F.A. Oliehoek, M.T.J. Spaan, and N. Vlassis, "Optimal and Approximate Q-Value Functions for Decentralized POMDPs," *Journal of Artificial Intelligence Research*, vol. 32, pp. 289–353, 2008.
24. F.A. Oliehoek, M.T.J. Spaan, and C. Amato, "Scaling Up Optimal Heuristic Search in Dec-POMDPs via Incremental Expansion," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.
25. F.A. Oliehoek, M.T.J. Spaan, C. Amato, and S. Whiteson, "Incremental Clustering and Expansion for Faster Optimal Planning in Dec-POMDPs," *Journal of Artificial Intelligence Research*, vol. 46, pp. 449–509, 2013. DOI: 10.1613/jair.3804.
26. F.A. Oliehoek, "Sufficient Plan-Time Statistics for Decentralized POMDPs," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.
27. J.S. Dibangoye, C. Amato, O. Buffet, and F. Charpillet, "Optimally Solving Dec-POMDPs as Continuous-State MDPs," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.
28. A. Carlin and S. Zilberstein, "Value-Based Observation Compression for DEC-POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008.
29. J.S. Dibangoye, A.-I. Mouaddib, and B. Chaib-draa, "Point-Based Incremental Pruning Heuristic for Solving Finite-Horizon DEC-POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2009.
30. A. Kumar and S. Zilberstein, "Point-Based Backup for Decentralized POMDPs: Complexity and New Algorithms," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2010.
31. F. Wu, S. Zilberstein, and X. Chen, "Point-Based Policy Generation for Decentralized POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2010.
32. C. Amato, B. Bonet, and S. Zilberstein, "Finite-State Controllers Based on Mealy Machines for Centralized and Decentralized POMDPs," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

33. A. Kumar and S. Zilberstein, "Anytime Planning for Decentralized POMDPs Using Expectation Maximization," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

34. J.K. Pajarinen and J. Peltonen, "Periodic Finite State Controllers for Efficient POMDP and DEC-POMDP Planning," in *Advances in Neural Information Processing Systems (NIPS)*, 2011.

35. M. Petrik and S. Zilberstein, "A Bilinear Programming Approach for Multiagent Planning," *Journal of Artificial Intelligence Research*, vol. 35, pp. 235–274, 2009. DOI: 10.1613/jair.2673.

36. J.S. Dibangoye, C. Amato, and A. Doniec, "Scaling Up Decentralized MDPs Through Heuristic Search," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2012.

37. J.S. Dibangoye, C. Amato, A. Doniec, and F. Charpillet, "Producing Efficient Error-Bounded Solutions for Transition Independent Decentralized MDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2013.

38. P. Varakantham, J. Marecki, Y. Yabu, M. Tambe, and M. Yokoo, "Letting Loose a SPIDER on a Network of POMDPS: Generating Quality Guaranteed Policies," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2007.

39. J. Marecki, T. Gupta, P. Varakantham, M. Tambe, and M. Yokoo, "Not All Agents Are Equal: Scaling up Distributed POMDPs for Agent Networks," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008.

40. P. Xuan and V. Lesser, "Multi-Agent Policies: From Centralized Ones to Decentralized Ones," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2002.

41. R. Becker, A. Carlin, V. Lesser, and S. Zilberstein, "Analyzing Myopic Approaches for Multi-Agent Communication," *Computational Intelligence*, vol. 25, no. 1, pp. 31–50, 2009. DOI: 10.1111/j.1467-8640.2008.01329.x.

42. M.T.J. Spaan, F.A. Oliehoek, and N. Vlassis, "Multiagent Planning Under Uncertainty with Stochastic Communication Delays," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

43. F. Wu, S. Zilberstein, and X. Chen, "Multi-Agent Online Planning with Communication," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

44. F.A. Oliehoek, M.T.J. Spaan, S. Whiteson, and N. Vlassis, "Exploiting Locality of Interaction in Factored Dec-POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008.

45. S.J. Witwicki and E.H. Durfee, "Towards a Unifying Characterization for Quantifying Weak Coupling in Dec-POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2011.

46. S.J. Witwicki, F.A. Oliehoek, and L.P. Kaelbling, "Heuristic Search of Multiagent Influence Space," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2012.

47. F.A. Oliehoek, S. Whiteson, and M.T.J. Spaan, "Approximate Solutions for Factored Dec-POMDPs with Many Agents," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2013.

48. A. Dutech, O. Buffet, and F. Charpillet, "Multi-Agent Systems by Incremental Gradient Reinforcement Learning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

49. L. Peshkin, K.-E. Kim, N. Meuleau, and L.P. Kaelbling, "Learning to Cooperate via Policy Search," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2000.

50. C. Zhang and V.R. Lesser, "Coordinated Multi-Agent Reinforcement Learning in Networked Distributed POMDPs," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2011.

51. R. Aras, A. Dutech, and F. Charpillet, "Mixed Integer Linear Programming for Exact Finite-Horizon Planning in Decentralized POMDPs," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.

52. C. Amato and S. Zilberstein, "Achieving Goals in Decentralized POMDPs," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2009.

53. F.A. Oliehoek, J.F. Kooi, and N. Vlassis, "The Cross-Entropy Method for Policy Search in Decentralized POMDPs," *Informatica*, vol. 32, no. 4, pp. 341–357, 2008.

54. D.S. Bernstein, S. Zilberstein, R. Washington, and J.L. Bresina, "Planetary Rover Control as a Markov Decision Process," in *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2001.

55. R. Emery-Montemerlo, G. Gordon, J. Schneider, and S. Thrun, "Game Theoretic Control for Robot Teams," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

56. M.T.J. Spaan and F.S. Melo, "Interaction-Driven Markov Games for Decentralized Multiagent Planning Under Uncertainty," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008.

57. L. Matignon, L. Jeanpierre, and A.-I. Mouaddib, "Coordinated Multi-Robot Exploration Under Communication Constraints Using Decentralized Markov Decision Processes," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2012.

58. R. Cogill, M. Rotkowitz, B. Van Roy, and S. Lall, "An Approximate Dynamic Programming Approach to Decentralized Control of Stochastic Systems," in *Allerton Conference on Communication, Control, and Computing*, 2004.

59. K. Winstein and H. Balakrishnan, "TCP Ex Machina: Computer-Generated Congestion Control," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

60. J.M. Ooi and G.W. Wornell, "Decentralized Control of a Multiple Access Broadcast Channel: Performance Bounds," in *IEEE Conference on Decision and Control (CDC)*, 1996.

61. L. Peshkin and V. Savova, "Reinforcement Learning for Adaptive Routing," in *International Joint Conference on Neural Networks (IJCNN)*, 2002.

62. A. Kumar and S. Zilberstein, "Constraint-Based Dynamic Programming for Decentralized POMDPs with Structured Interactions," in *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2009.

63. ——, "Event-Detecting Multi-Agent MDPS: Complexity and Constant-Factor Approximation," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.