Chapter 10

Sequential Decision Theory

Chapter 9 essentially took a break from planning by indicating how to make a single decision in the presence of uncertainty. In this chapter, we return to planning by formulating a sequence of decision problems. This is achieved by extending the discrete planning concepts from Chapter 2 to incorporate the effects of multiple decision makers. The most important new decision maker is nature, which causes unpredictable outcomes when actions are applied during the execution of a plan. State spaces and state transition equations reappear in this chapter; however, in contrast to Chapter 2, additional decision makers interfere with the state transitions. As a result of this effect, a plan needs to incorporate state feedback, which enables it to choose an action based on the current state. When the plan is determined, it is not known what future states will arise. Therefore, feedback is required, as opposed to specifying a plan as a sequence of actions, which sufficed in Chapter 2. This was only possible because actions were predictable.

Keep in mind throughout this chapter that the current state is always known. The only uncertainty that exists is with respect to predicting future states. Chapters 11 and 12 will address the important and challenging case in which the current state is not known. This requires defining sensing models that attempt to measure the state. The main result is that planning occurs in an *information space*, as opposed to the state space. Most of the ideas of this chapter extend into information spaces when uncertainties in prediction and in the current state exist together.

The problems considered in this chapter have a wide range of applicability. Most of the ideas were developed in the context of stochastic control theory [93, 564, 567]. The concepts can be useful for modeling problems in mobile robotics because future states are usually unpredictable and can sometimes be modeled probabilistically [1004] or using worst-case analysis [590]. Many other applications exist throughout engineering, operations research, and economics. Examples include process scheduling, gambling strategies, and investment planning.

As usual, the focus here is mainly on arriving in a goal state. Both nondeterministic and probabilistic forms of uncertainty will be considered. In the nondeterministic case, the task is to find plans that are guaranteed to work in spite of nature. In some cases, a plan can be computed that has optimal worst-

case performance while achieving the goal. In the probabilistic case, the task is to find a plan that yields optimal expected-case performance. Even though the outcome is not predictable in a single-plan execution, the idea is to reduce the average cost, if the plan is executed numerous times on the same problem.

10.1 Introducing Sequential Games Against Nature

This section extends many ideas from Chapter 2 to the case in which nature interferes with the outcome of actions. Section 10.1.1 defines the planning problem in this context, which is a direct extension of Section 2.1. Due to unpredictability, *forward projections* and *backprojections* are introduced in Section 10.1.2 to characterize possible future and past states, respectively. Forward projections characterize the future states that will be obtained under the application of a plan or a sequence of actions. In Chapter 2 this concept was not needed because the sequence of future states could always be derived from a plan and initial state. Section 10.1.3 defines the notion of a plan and uses forward projections to indicate how its execution may differ every time the plan is applied.

10.1.1 Model Definition

The formulation presented in this section is an extension of Formulation 2.3 that incorporates the effects of nature at every stage. Let X denote a discrete state space, and let $U(x)$ denote the set of actions available to the decision maker (or robot) from state $x \in X$. At each stage k it is assumed that a *nature action* θ_k is chosen from a set $\Theta(x_k, u_k)$. This can be considered as a multi-stage generalization of Formulation 9.4, which introduced $\Theta(u)$. Now Θ may depend on the state in addition to the action because both x_k and u_k are available in the current setting. This implies that nature acts with the knowledge of the action selected by the decision maker. It is always assumed that during stage k, the decision maker does not know the particular nature action that will be chosen. It does, however, know the set $\Theta(x_k, u_k)$ for all $x_k \in X$ and $u_k \in U(x_k)$.

As in Section 9.2, there are two alternative nature models: nondeterministic or probabilistic. If the nondeterministic model is used, then it is only known that nature will make a choice from $\Theta(x_k, u_k)$. In this case, making decisions using worst-case analysis is appropriate.

If the probabilistic model is used, then a probability distribution over $\Theta(x_k, u_k)$ is specified as part of the model. The most important assumption to keep in mind for this case is that nature is *Markovian*. In general, this means that the probability depends only on local information. In most applications, this locality is with respect to time. In our formulation, it means that the distribution over $\Theta(x_k, u_k)$ depends only on information obtained at the current stage. In other settings, Markovian could mean a dependency on a small number of stages, or

even a local dependency in terms of spatial relationships, as in a *Markov random field* [231, 377].

To make the Markov assumption more precise, the state and action histories as defined in Section 8.2.1 will be used again here. Let

$$\tilde{x}_k = (x_1, x_2, \ldots, x_k) \tag{10.1}$$

and

$$\tilde{u}_k = (u_1, u_2, \ldots, u_k). \tag{10.2}$$

These represent all information that is available up to stage k. Without the Markov assumption, it could be possible that the probability distribution for nature is conditioned on all of $\tilde{x}_k$ and $\tilde{u}_k$, to obtain $P(\theta_k|\tilde{x}_k, \tilde{u}_k)$. The Markov assumption declares that for all $\theta_k \in \Theta(x_k, u_k)$,

$$P(\theta_k|\tilde{x}_k, \tilde{u}_k) = P(\theta_k|x_k, u_k), \tag{10.3}$$

which drops all history except the current state and action. Once these two are known, there is no extra information regarding the nature action that could be gained from any portion of the histories.

The effect of nature is defined in the state transition equation, which produces a new state, x_{k+1}, once x_k, u_k, and θ_k are given:

$$x_{k+1} = f(x_k, u_k, \theta_k). \tag{10.4}$$

From the perspective of the decision maker, θ_k is not given. Therefore, it can only infer that a particular set of states will result from applying u_k and x_k:

$$X_{k+1}(x_k, u_k) = \{x_{k+1} \in X \mid \exists \theta_k \in \Theta(x_k, u_k) \text{ such that } x_{k+1} = f(x_k, u_k, \theta_k)\}. \tag{10.5}$$

In (10.5), the notation $X_{k+1}(x_k, u_k)$ indicates a set of possible values for x_{k+1}, given x_k and u_k. The notation $X_k(\cdot)$ will generally be used to indicate the possible values for x_k that can be derived using the information that appears in the argument.

In the probabilistic case, a probability distribution over X can be derived for stage $k+1$, under the application of u_k from x_k. As part of the problem, $P(\theta_k|x_k, u_k)$ is given. Using the state transition equation, $x_{k+1} = f(x_k, u_k, \theta_k)$,

$$P(x_{k+1}|x_k, u_k) = \sum_{\theta_k \in \Theta'} P(\theta_k|x_k, u_k) \tag{10.6}$$

can be derived, in which

$$\Theta' = \{\theta_k \in \Theta(x_k, u_k) \mid x_{k+1} = f(x_k, u_k, \theta_k)\}. \tag{10.7}$$

The calculation of $P(x_{k+1}|x_k, u_k)$ simply involves accumulating all of the probability mass that could lead to x_{k+1} from the application of various nature actions.

Putting these parts of the model together and adding some of the components from Formulation 2.3, leads to the following formulation:

Formulation 10.1 (Discrete Planning with Nature)

1. A nonempty *state space* X which is a finite or countably infinite set of *states*.
2. For each state, $x \in X$, a finite, nonempty *action space* $U(x)$. It is assumed that U contains a special *termination action*, which has the same effect as the one defined in Formulation 2.3.
3. A finite, nonempty *nature action space* $\Theta(x, u)$ for each $x \in X$ and $u \in U(x)$.
4. A *state transition function* f that produces a state, $f(x, u, \theta)$, for every $x \in X$, $u \in U$, and $\theta \in \Theta(x, u)$.
5. A set of *stages*, each denoted by k, that begins at $k = 1$ and continues indefinitely. Alternatively, there may be a fixed, maximum stage $k = K+1 = F$.
6. An *initial state* $x_I \in X$. For some problems, this may not be specified, in which case a solution plan must be found from all initial states.
7. A *goal set* $X_G \subset X$.
8. A stage-additive cost functional L. Let $\tilde{\theta}_K$ denote the history of nature actions up to stage K. The cost functional may be applied to any combination of state, action, and nature histories to yield

$$L(\tilde{x}_F, \tilde{u}_K, \tilde{\theta}_K) = \sum_{k=1}^{K} l(x_k, u_k, \theta_k) + l_F(x_F), \tag{10.8}$$

in which $F = K + 1$. If the termination action u_T is applied at some stage k, then for all $i \geq k$, $u_i = u_T$, $x_i = x_k$, and $l(x_i, u_T, \theta_i) = 0$.

Using Formulation 10.1, either a feasible or optimal planning problem can be defined. To obtain a feasible planning problem, let $l(x_k, u_k, \theta_k) = 0$ for all $x_k \in X$, $u_k \in U$, and $\theta_k \in \Theta_k(u_k)$. Furthermore, let

$$l_F(x_F) = \begin{cases} 0 & \text{if } x_F \in X_G \\ \infty & \text{otherwise.} \end{cases} \tag{10.9}$$

To obtain an optimal planning problem, in general $l(x_k, u_k, \theta_k)$ may assume any nonnegative, finite value if $x_k \notin X_G$. For problems that involve probabilistic uncertainty, it is sometimes appropriate to assign a high, finite value for $l_F(x_F)$ if $x_F \notin X_G$, as opposed to assigning an infinite cost for failing to achieve the goal.

Note that in each stage, the cost term is generally allowed to depend on the nature action θ_k. If probabilistic uncertainty is used, then Formulation 10.1 is often referred to as a *controlled Markov process* or *Markov decision process* (MDP). If the actions are removed from the formulation, then it is simply referred to as a *Markov process*. In most statistical literature, the name *Markov chain* is used instead of

Markov process when there are discrete stages (as opposed to continuous-time Markov processes). Thus, the terms *controlled Markov chain* and *Markov decision chain* may be preferable.

In some applications, it may be convenient to avoid the explicit characterization of nature. Suppose that $l(x_k, u_k, \theta_k) = l(x_k, u_k)$. If nondeterministic uncertainty is used, then $X_{k+1}(x_k, u_k)$ can be specified for all $x_k \in X$ and $u_k \in U(x_k)$ as a substitute for the state transition equation; this avoids having to refer to nature. The application of an action u_k from a state x_k directly leads to a specified subset of X. If probabilistic uncertainty is used, then $P(x_{k+1}|x_k, u_k)$ can be directly defined as the alternative to the state transition equation. This yields a probability distribution over X, if u_k is applied from some x_k, once again avoiding explicit reference to nature. Most of the time we will use a state transition equation that refers to nature; however, it is important to keep these alternatives in mind. They arise in many related books and research articles.

As used throughout Chapter 2, a directed state transition graph is sometimes convenient for expressing the planning problem. The same idea can be applied in the current setting. As in Section 2.1, X is the vertex set; however, the edge definition must change to reflect nature. A directed edge exists from state x to x' if there exists some $u \in U(x)$ and $\theta \in \Theta(x, u)$ such that $x' = f(x, u, \theta)$. A weighted graph can be made by associating the cost term $l(x_k, u_k, \theta_k)$ with each edge. In the case of a probabilistic model, the probability of the transition occurring may also be associated with each edge.

Note that both the decision maker and nature are needed to determine which vertex will be reached. As the decision maker contemplates applying an action u from the state x, it sees that there may be several outgoing edges due to nature. If a different action is contemplated, then this set of possible outgoing edges changes. Once nature applies its action, then the particular edge is traversed to arrive at the new state; however, this is not completely controlled by the decision maker.

Example 10.1 (Traversing the Number Line) Let $X = \mathbb{Z}$, $U = \{-2, 2, u_T\}$, and $\Theta = \{-1, 0, 1\}$. The action sets of the decision maker and nature are the same for all states. For the state transition equation, $x_{k+1} = f(x_k, u_k, \theta_k) = x_k + u_k + \theta_k$. For each stage, unit cost is received. Hence $l(x, u, \theta) = 1$ for all x, θ, and $u \neq u_T$. The initial state is $x_I = 100$, and the goal set is $X_G = \{-1, 0, 1\}$.

Consider executing a sequence of actions, $(-2, -2, \ldots, -2)$, under the nondeterministic uncertainty model. This means that we attempt to move left two units in each stage. After the first -2 is applied, the set of possible next states is $\{97, 98, 99\}$. Nature may slow down the progress to be only one unit per stage, or it may speed up the progress so that X_G is three units closer per stage. Note that after 100 stages, the goal is guaranteed to be achieved, in spite of any possible actions of nature. Once X_G is reached, u_T should be applied. If the problem is changed so that $X_G = \{0\}$, it becomes impossible to guarantee that the goal will be reached because nature may cause the goal to be overshot.

Now let $U = \{-1, 1, u_T\}$ and $\Theta = \{-2, -1, 0, 1, 2\}$. Under nondeterministic uncertainty, the problem can no longer be solved because nature is now powerful

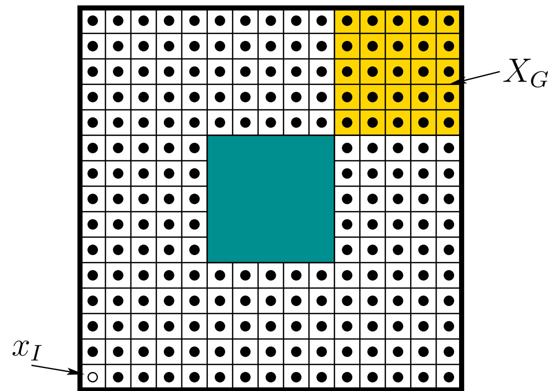


Figure 10.1: A grid-based shortest path problem with interference from nature.

enough to move the state completely in the wrong direction in the worst case. A reasonable probabilistic version of the problem can, however, be defined and solved. Suppose that $P(\theta) = 1/5$ for each $\theta \in \Theta$. The transition probabilities can be defined from $P(\theta)$. For example, if $x_k = 100$ and $u_k = -1$, then $P(x_{k+1}|x_k, u_k) = 1/5$ if $97 \leq x_k \leq 101$, and $P(x_{k+1}|x_k, u_k) = 0$ otherwise. With the probabilistic formulation, there is a nonzero probability that the goal, $X_G = \{-1, 0, 1\}$, will be reached, even though in the worst-case reaching the goal is not guaranteed. ■

Example 10.2 (Moving on a Grid) A grid-based robot planning model can be made. A simple example is shown in Figure 10.1. The state space is a subset of a 15×15 integer grid in the plane. A state is represented as (i, j), in which $1 \leq i, j \leq 15$; however, the points in the center region (shown in Figure 10.1) are not included in X.

Let $A = \{0, 1, 2, 3, 4\}$ be a set of actions, which denote "stay," "right," "up," "left," and "down," respectively. Let $U = A \cup u_T$. For each $x \in X$, let $U(x)$ contain u_T and whichever actions are applicable from x (some are not applicable along the boundaries).

Let $\Theta(x, u)$ represent the set of all actions in A that are applicable after performing the move implied by u. For example, if $x = (2, 2)$ and $u = 3$, then the robot is attempting to move to $(1, 2)$. From this state, there are three neighboring states, each of which corresponds to an action of nature. Thus, $\Theta(x, u)$ in this case is $\{0, 1, 2, 4\}$. The action $\theta = 3$ does not appear because there is no state to the left of $(1, 2)$. Suppose that the probabilistic model is used, and that every nature action is equally likely.

The state transition function f is formed by adding the effect of both u_k and θ_k. For example, if $x_k = (i, j)$, $u_k = 1$, and $\theta_k = 2$, then $x_{k+1} = (i + 1, j + 1)$. If θ_k had been 3, then the two actions would cancel and $x_{k+1} = (i, j)$. Without nature, it would have been assumed that $\theta_k = 0$. As always, the state never changes once u_T is applied, regardless of nature's actions.

For the cost functional, let $l(x_k, u_k) = 1$ (unless $u_k = u_T$; in this case, $l(x_k, u_T) = 0$). For the final stage, let $l_F(x_F) = 0$ if $x_F \in X_G$; otherwise, let $l_F(x_F) = \infty$. A reasonable task is to get the robot to terminate in X_G in the minimum expected number of stages. A feedback plan is needed, which will be introduced in Section 10.1.3, and the optimal plan for this problem can be efficiently computed using the methods of Section 10.2.1.

This example can be easily generalized to moving through a complicated labyrinth in two or more dimensions. If the grid resolution is high, then an approximation to motion planning is obtained. Rather than forcing motions in only four directions, it may be preferable to allow any direction. This case is covered in Section 10.6, which addresses planning in continuous state spaces. ■

10.1.2 Forward Projections and Backprojections

A *forward projection* is a useful concept for characterizing the behavior of plans during execution. Before uncertainties were considered, a plan was executed exactly as expected. When a sequence of actions was applied to an initial state, the resulting sequence of states could be computed using the state transition equation. Now that the state transitions are unpredictable, we would like to imagine what states are possible several stages into the future. In the case of nondeterministic uncertainty, this involves computing a set of possible future states, given a current state and plan. In the probabilistic case, a probability distribution over states is computed instead.

Nondeterministic forward projections To facilitate the notation, suppose in this section that $U(x) = U$ for all $x \in X$. In Section 10.1.3 this will be lifted.

Suppose that the initial state, $x_1 = x_I$, is known. If the action $u_1 \in U$ is applied, then the set of possible next states is

$$X_2(x_1, u_1) = \{x_2 \in X \mid \exists \theta_1 \in \Theta(x_1, u_1) \text{ such that } x_2 = f(x_1, u_1, \theta_1)\}, \quad (10.10)$$

which is just a special version of (10.5). Now suppose that an action $u_2 \in U$ will be applied. The forward projection must determine which states could be reached from x_1 by applying u_1 followed by u_2. This can be expressed as

$$X_3(x_1, u_1, u_2) = \{x_3 \in X \mid \exists \theta_1 \in \Theta(x_1, u_1) \text{ and } \exists \theta_2 \in \Theta(x_2, u_2) \\ \text{such that } x_2 = f(x_1, u_1, \theta_1) \text{ and } x_3 = f(x_2, u_2, \theta_2)\}. \quad (10.11)$$

This idea can be repeated for any number of iterations but becomes quite cumbersome in the current notation. It is helpful to formulate the forward projection recursively. Suppose that an action history $\tilde{u}_k$ is fixed. Let $X_{k+1}(X_k, u_k)$ denote the forward projection at stage $k + 1$, given that X_k is the forward projection at

stage k. This can be computed as

$$X_{k+1}(X_k, u_k) = \{x_{k+1} \in X \mid \exists x_k \in X_k \text{ and } \exists \theta_k \in \Theta(x_k, u_k) \text{ such that } x_{k+1} = f(x_k, u_k, \theta_k)\}. \tag{10.12}$$

This may be applied any number of times to compute X_{k+1} from an initial condition $X_1 = \{x_1\}$.

Example 10.3 (Nondeterministic Forward Projections) Recall the first model given in Example 10.1, in which $U = \{-2, 2, u_T\}$ and $\Theta = \{-1, 0, 1\}$. Suppose that $x_1 = 0$, and $u = 2$ is applied. The one-stage forward projection is $X_2(0, 2) = \{1, 2, 3\}$. If $u = 2$ is applied again, the two-stage forward projection is $X_3(0, 2, 2) = \{2, 3, 4, 5, 6\}$. Repeating this process, the k-stage forward projection is $\{k, \ldots, 3k\}$. ■

Probabilistic forward projections The probabilistic forward projection can be considered as a Markov process because the "decision" part is removed once the actions are given. Suppose that x_k is given and u_k is applied. What is the probability distribution over x_{k+1}? This was already specified in (10.6) and is the one-stage forward projection. Now consider the two-stage probabilistic forward projection, $P(x_{k+2}|x_k, u_k, u_{k+1})$. This can be computed by marginalization as

$$P(x_{k+2}|x_k, u_k, u_{k+1}) = \sum_{x_{k+1} \in X} P(x_{k+2}|x_{k+1}, u_{k+1}) P(x_{k+1}|x_k, u_k). \tag{10.13}$$

Computing further forward projections requires nested summations, which marginalize all of the intermediate states. For example, the three-stage forward projection is

$$P(x_{k+3}|x_k, u_k, u_{k+1}, u_{k+2}) = \sum_{x_{k+1} \in X} \sum_{x_{k+2} \in X} P(x_{k+3}|x_{k+2}, u_{k+2}) P(x_{k+2}|x_{k+1}, u_{k+1}) P(x_{k+1}|x_k, u_k). \tag{10.14}$$

A convenient expression of the probabilistic forward projections can be obtained by borrowing nice algebraic properties from linear algebra. For each action $u \in U$, let its *state transition matrix* M_u be an $n \times n$ matrix, for $n = |X|$, of probabilities. The matrix is defined as

$$M_u = \begin{pmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,n} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,n} \\ \vdots & \vdots & & \vdots \\ m_{n,1} & m_{n,2} & \cdots & m_{n,n} \end{pmatrix}, \tag{10.15}$$

in which

$$m_{i,j} = P(x_{k+1} = i \mid x_k = j,\ u). \tag{10.16}$$

For each j, the jth column of M_u must sum to one and can be interpreted as the probability distribution over X that is obtained if u_k is applied from state $x_k = j$.

Let v denote an n-dimensional column vector that represents any probability distribution over X. The product $M_u v$ yields a column vector that represents the probability distribution over X that is obtained after starting with v and applying u. The matrix multiplication performs n inner products, each of which is a marginalization as shown in (10.13). The forward projection at any stage, k, can now be expressed using a product of $k-1$ state transition matrices. Suppose that $\tilde{u}_{k-1}$ is fixed. Let $v = [0\ 0\ \cdots 0\ 1\ 0\ \cdots\ 0]$, which indicates that x_1 is known (with probability one). The forward projection can be computed as

$$v' = M_{u_{k-1}} M_{u_{k-2}} \cdots M_{u_2} M_{u_1} v. \tag{10.17}$$

The ith element of v' is $P(x_k = i \mid x_1, \tilde{u}_{k-1})$.

Example 10.4 (Probabilistic Forward Projections) Once again, use the first model from Example 10.1; however, now assign probability 1/3 to each nature action. Assume that, initially, $x_1 = 0$, and $u = 2$ is applied in every stage. The one-stage forward projection yields probabilities

$$[1/3\ \ 1/3\ \ 1/3] \tag{10.18}$$

over the sequence of states $(1, 2, 3)$. The two-stage forward projection yields

$$[1/9\ \ 2/9\ \ 3/9\ \ 2/9\ \ 1/9] \tag{10.19}$$

over $(2, 3, 4, 5, 6)$. ■

Backprojections Sometimes it is helpful to define the set of possible previous states from which one or more current states could be obtained. For example, they will become useful in defining graph-based planning algorithms in Section 10.2.3. This involves maintaining a *backprojection*, which is a counterpart to the forward projection that runs in the opposite direction. Backprojections were considered in Section 8.5.2 to keep track of the active states in a Dijkstra-like algorithm over continuous state spaces. In the current setting, backprojections need to address uncertainty.

Consider the case of nondeterministic uncertainty. Let a state $x \in X$ be given. Under a fixed action u, what previous states, $x' \in X$, could possibly lead to x? This depends only on the possible choices of nature and is expressed as

$$\mathrm{WB}(x, u) = \{x' \in X \mid \exists \theta \in \Theta(x', u) \text{ such that } x = f(x', u, \theta)\}. \tag{10.20}$$

The notation $\mathrm{WB}(x, u)$ refers to the *weak backprojection of x under u*, and gives the set of all states from which x may possibly be reached in one stage.

The backprojection is called "weak" because it does not guarantee that x is reached, which is a stronger condition. By guaranteeing that x is reached, a *strong backprojection of x under u* is defined as

$$\mathrm{SB}(x, u) = \{x' \in X \mid \forall \theta \in \Theta(x', u), \; x = f(x', u, \theta)\}. \tag{10.21}$$

The difference between (10.20) and (10.21) is either *there exists* an action of nature that enables x to be reached, or x is reached *for all* actions of nature. Note that $\mathrm{SB}(x, u) \subseteq \mathrm{WB}(x, u)$. In many cases, $\mathrm{SB}(x, u) = \emptyset$, and $\mathrm{WB}(x, u)$ is rarely empty. The backprojection that was introduced in (8.66) of Section 8.5.2 did not involve uncertainty; hence, the distinction between weak and strong backprojections did not arise.

Two useful generalizations will now be made: 1) A backprojection can be defined from a set of states; 2) the action does not need to be fixed. Instead of a fixed state, x, consider a set $S \subseteq X$ of states. What are the states from which an element of S could possibly be reached in one stage under the application of u? This is the *weak backprojection of S under u:*

$$\mathrm{WB}(S, u) = \{x' \in X \mid \exists \theta \in \Theta(x', u) \text{ such that } f(x', u, \theta) \in S\}, \tag{10.22}$$

which can also be expressed as

$$\mathrm{WB}(S, u) = \bigcup_{x \in S} \mathrm{WB}(x, u). \tag{10.23}$$

Similarly, the *strong backprojection of S under u* is defined as

$$\mathrm{SB}(S, u) = \{x' \in X \mid \forall \theta \in \Theta(x', u), \; f(x', u, \theta) \in S\}. \tag{10.24}$$

Note that $\mathrm{SB}(S, u)$ cannot be formed by the union of $SB(x, u)$ over all $x \in S$. Another observation is that for each $x_k \in \mathrm{SB}(S, u_k)$, we have $X_{k+1}(x_k, u_k) \subseteq S$.

Now the dependency on u will be removed. This yields a backprojection of a set S. These are states from which there exists an action that possibly reaches S. The *weak backprojection of S* is

$$\mathrm{WB}(S) = \{x' \in X \mid \exists u \in U(x) \text{ such that } x \in \mathrm{WB}(S, u)\}, \tag{10.25}$$

and the *strong backprojection of S* is

$$\mathrm{SB}(S) = \{x' \in X \mid \exists u \in U(x) \text{ such that } x \in \mathrm{SB}(S, u)\}. \tag{10.26}$$

Note that $\mathrm{SB}(S) \subseteq \mathrm{WB}(S)$.

Example 10.5 (Backprojections) Once again, consider the model from the first part of Example 10.1. The backprojection $\mathrm{WB}(0, 2)$ represents the set of

all states from which $u = 2$ can be applied and $x = 0$ is possibly reached; the result is $\mathrm{WB}(0, 2) = \{-3, -2, -1\}$. The state 0 cannot be reached with certainty from any state in $\mathrm{WB}(0, 2)$. Therefore, $\mathrm{SB}(0, 2) = \emptyset$.

Now consider backprojections from the goal, $X_G = \{-1, 0, 1\}$, under the action $u = 2$. The weak backprojection is

$$\mathrm{WB}(X_G, 2) = \mathrm{WB}(-1, 2) \cup \mathrm{WB}(0, 2) \cup \mathrm{WB}(1, 2) = \{-4, -3, -2, -1, 0\}. \quad (10.27)$$

The strong backprojection is $\mathrm{SB}(X_G, 2) = \{-2\}$. From any of the other states in $\mathrm{WB}(X_G, 2)$, nature could cause the goal to be missed. Note that $\mathrm{SB}(X_G, 2)$ cannot be constructed by taking the union of $\mathrm{SB}(x, 2)$ over every $x \in X_G$.

Finally, consider backprojections that do not depend on an action. These are $\mathrm{WB}(X_G) = \{-4, -3, \ldots, 4\}$ and $\mathrm{SB}(X_G) = X_G$. In the latter case, all states in X_G lie in $\mathrm{SB}(X_G)$ because u_T can be applied. Without allowing u_T, we would obtain $\mathrm{SB}(X_G) = \{-2, 2\}$. ■

Other kinds of backprojections are possible, but we will not define them. One possibility is to make backprojections over multiple stages, as was done for forward projections. Another possibility is to define them for the probabilistic case. This is considerably more complicated. An example of a probabilistic backprojection is to find the set of all states from which a state in S will be reached with at least probability p.

10.1.3 A Plan and Its Execution

In Chapter 2, a plan was specified by a sequence of actions. This was possible because the effect of actions was completely predictable. Throughout most of Part II, a plan was specified as a path, which is a continuous-stage version of the action sequence. Section 8.2.1 introduced plans that are expressed as a function on the state space. This was optional because uncertainty was not explicitly modeled (except perhaps in the initial state).

As a result of unpredictability caused by nature, it is now important to separate the definition of a plan from its execution. The same plan may be executed many times from the same initial state; however, because of nature, different future states will be obtained. This requires the use of feedback in the form of a plan that maps states to actions.

Defining a plan Let a *(feedback) plan* for Formulation 10.1 be defined as a function $\pi : X \to U$ that produces an action $\pi(x) \in U(x)$, for each $x \in X$. Although the future state may not be known due to nature, if π is given, then it will at least be known what action will be taken from any future state. In other works, π has been called a *feedback policy*, *feedback control law*, *reactive plan* [340], and *conditional plan*.

For some problems, particularly when K is fixed at some finite value, a *stage-dependent plan* may be necessary. This enables a different action to be chosen for

every stage, even from the same state. Let $\mathcal{K}$ denote the set $\{1, \ldots, K\}$ of stages. A stage-dependent plan is defined as $\pi : X \times \mathcal{K} \rightarrow U$. Thus, an action is given by $u = \pi(x, k)$. Note that the definition of a K-step plan, which was given Section 2.3, is a special case of the current definition. In that setting, the action depended only on the stage because future states were always predictable. Here they are no longer predictable and must be included in the domain of π. Unless otherwise mentioned, it will be assumed by default that π is *not* stage-dependent.

Note that once π is formulated, the state transitions appear to be a function of only the current state and nature. The next state is given by $f(x, \pi(x), \theta)$. The same is true for the cost term, $l(x, \pi(x), \theta)$.

Forward projections under a fixed plan Forward projections can now be defined under the constraint that a particular plan is executed. The specific expression of actions is replaced by π. Each time an action is needed from a state $x \in X$, it is obtained as $\pi(x)$. In this formulation, a different $U(x)$ may be used for each $x \in X$, assuming that π is correctly defined to use whatever actions are actually available in $U(x)$ for each $x \in X$.

First we will consider the nondeterministic case. Suppose that the initial state x_1 and a plan π are known. This means that $u_1 = \pi(x_1)$, which can be substituted into (10.10) to compute the one-stage forward projection. To compute the two-stage forward projection, u_2 is determined from $\pi(x_2)$ for use in (10.11). A recursive formulation of the nondeterministic forward projection under a fixed plan is

$$\begin{split} X_{k+1}(x_1, \pi) = \{x_{k+1} \in X \mid \exists \theta_k \in \Theta(x_k, \pi(x_k)) \text{ such that} \\ x_k \in X_k(x_1, \pi) \text{ and } x_{k+1} = f(x_k, \pi(x_k), \theta_k)\}. \end{split} \tag{10.28}$$

The probabilistic forward projection in (10.10) can be adapted to use π, which results in

$$P(x_{k+2}|x_k, \pi) = \sum_{x_{k+1} \in X} P(x_{k+2}|x_{k+1}, \pi(x_{k+1})) P(x_{k+1}|x_k, \pi(x_k)). \tag{10.29}$$

The basic idea can be applied $k-1$ times to compute $P(x_k|x_1, \pi)$.

A state transition matrix can be used once again to express the probabilistic forward projection. In (10.15), all columns correspond to the application of the action u. Let M_π, be the forward projection due to a fixed plan π. Each column of M_π may represent a different action because each column represents a different state x_k. Each entry of M_π is

$$m_{i,j} = P(x_{k+1} = i \mid x_k = j,\ \pi(x_k)). \tag{10.30}$$

The resulting M_π defines a Markov process that is induced under the application of the plan π.

Graph representations of a plan The game against nature involves two decision makers: nature and the robot. Once the plan is formulated, the decisions of the robot become fixed, which leaves nature as the only remaining decision maker. Using this interpretation, a directed graph can be defined in the same way as in Section 2.1, except nature actions are used instead of the robot's actions. It can even be imagined that nature itself faces a discrete feasible planning problem as in Formulation 2.1, in which $\Theta(x, \pi(x))$ replaces $U(x)$, and there is no goal set. Let $\mathcal{G}_\pi$ denote a *plan-based state transition graph*, which arises under the constraint that π is executed. The vertex set of $\mathcal{G}_\pi$ is X. A directed edge in $\mathcal{G}_\pi$ exists from x to x' if there exists some $\theta \in \Theta(x, \pi(x))$ such that $x' = f(x, \pi(x), \theta)$. Thus, from each vertex in $\mathcal{G}_\pi$, the set of outgoing edges represents all possible transitions to next states that are possible, given that the action is applied according to π. In the case of probabilistic uncertainty, $\mathcal{G}_\pi$ becomes a weighted graph in which each edge is assigned the probability $P(x'|x, \pi(x), \theta)$. In this case, $\mathcal{G}_\pi$ corresponds to the graph representation commonly used to depict a Markov chain.

A nondeterministic forward projection can easily be derived from $\mathcal{G}_\pi$ by following the edges outward from the current state. The outward edges lead to the states of the one-stage forward projection. The outward edges of these states lead to the two-stage forward projection, and so on. The probabilistic forward projection can also be derived from $\mathcal{G}_\pi$.

The cost of a feedback plan Consider the cost-to-go of executing a plan π from a state $x_1 \in X$. The resulting cost depends on the sequences of states that are visited, actions that are applied by the plan, and the applied nature actions. In Chapter 2 this was obtained by adding the cost terms, but now there is a dependency on nature. Both worst-case and expected-case analyses are possible, which generalize the treatment of Section 9.2 to state spaces and multiple stages.

Let $\mathcal{H}(\pi, x_1)$ denote the set of state-action-nature histories that could arise from π when applied using x_1 as the initial state. The cost-to-go, $G_\pi(x_1)$, under a given plan π from x_1 can be measured using *worst-case analysis* as

$$G_\pi(x_1) = \max_{(\tilde{x}, \tilde{u}, \tilde{\theta}) \in \mathcal{H}(\pi, x_1)} \Big\{ L(\tilde{x}, \tilde{u}, \tilde{\theta}) \Big\}, \tag{10.31}$$

which is the maximum cost over all possible trajectories from x_1 under the plan π. If any of these fail to terminate in the goal, then the cost becomes infinity. In (10.31), $\tilde{x}$, $\tilde{u}$, and $\tilde{\theta}$ are infinite histories, although their influence on the cost is expected to terminate early due to the application of u_T.

An optimal plan using worst-case analysis is any plan for which $G_\pi(x_1)$ is minimized over all possible plans (all ways to assign actions to the states). In the case of feasible planning, there are usually numerous equivalent alternatives. Sometimes the task may be only to find a feasible plan, which means that all trajectories must reach the goal, but the cost does not need to be optimized.

Using probabilistic uncertainty, the cost of a plan can be measured using

expected-case analysis as

$$G_\pi(x_1) = E_{\mathcal{H}(\pi,x_1)}\Big[L(\tilde{x}, \tilde{u}, \tilde{\theta})\Big], \tag{10.32}$$

in which E denotes the mathematical expectation taken over $\mathcal{H}(\pi, x_1)$ (i.e., the plan is evaluated in terms of a weighted sum, in which each term has a weight for the probability of a state-action-nature history and its associated cost, $L(\tilde{x}, \tilde{u}, \tilde{\theta})$). This can also be interpreted as the expected cost over trajectories from x_1. If any of these have nonzero probability and fail to terminate in the goal, then $G_\pi(x_1) = \infty$. In the probabilistic setting, the task is usually to find a plan that minimizes $G_\pi(x_1)$.

An interesting question now emerges: Can the same plan, π, be optimal from every initial state $x_1 \in X$, or do we need to potentially find a different optimal plan for each initial state? Fortunately, a single plan will suffice to be optimal over all initial states. Why? This behavior was also observed in Section 8.2.1. If π is optimal from some x_1, then it must also be optimal from every other state that is potentially visited by executing π from x_1. Let x denote some visited state. If π was not optimal from x, then a better plan would exist, and the goal could be reached from x with lower cost. This contradicts the optimality of π because solutions must travel through x. Let π^* denote a plan that is optimal from every initial state.

10.2 Algorithms for Computing Feedback Plans

10.2.1 Value Iteration

Fortunately, the value iteration method of Section 2.3.1.1 extends nicely to handle uncertainty in prediction. This was the main reason why value iteration was introduced in Chapter 2. Value iteration was easier to describe in Section 2.3.1.1 because the complications of nature were avoided. In the current setting, value iteration retains most of its efficiency and can easily solve problems that involve thousands or even millions of states.

The state space, X, is assumed to be finite throughout Section 10.2.1. An extension to the case of a countably infinite state space can be developed if cost-to-go values over the entire space do not need to be computed incrementally.

Only backward value iteration is considered here. Forward versions can be defined alternatively.

Nondeterministic case Suppose that the nondeterministic model of nature is used. A dynamic programming recurrence, (10.39), will be derived. This directly yields an iterative approach that computes a plan that minimizes the worst-case cost. The following presentation shadows that of Section 2.3.1.1; therefore, it may be helpful to refer back to this periodically.

An optimal plan π^* will be found by computing optimal cost-to-go functions. For $1 \leq k \leq F$, let G_k^* denote the worst-case cost that could accumulate from stage k to F under the execution of the optimal plan (compare to (2.5))

$$G_k^*(x_k) = \min_{u_k} \max_{\theta_k} \min_{u_{k+1}} \max_{\theta_{k+1}} \cdots \min_{u_K} \max_{\theta_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i, \theta_i) + l_F(x_F) \right\}. \quad (10.33)$$

Inside of the min's and max's of (10.33) are the last $F - k$ terms of the cost functional, (10.8). For simplicity, the ranges of each u_i and θ_i in the min's and max's of (10.33) have not been indicated. The optimal cost-to-go for $k = F$ is

$$G_F^*(x_F) = l_F(x_F), \quad (10.34)$$

which is the same as (2.6) for the predictable case.

Now consider making K passes over X, each time computing G_k^* from G_{k+1}^*, as k ranges from F down to 1. In the first iteration, G_F^* is copied from l_F. In the second iteration, G_K^* is computed for each $x_K \in X$ as (compare to (2.7))

$$G_K^*(x_K) = \min_{u_K} \max_{\theta_K} \Big\{ l(x_K, u_K, \theta_K) + l_F(x_F) \Big\}, \quad (10.35)$$

in which $u_K \in U(x_K)$ and $\theta_K \in \Theta(x_K, u_K)$. Since $l_F = G_F^*$ and $x_F = f(x_K, u_K, \theta_K)$, substitutions are made into (10.35) to obtain (compare to (2.8))

$$G_K^*(x_K) = \min_{u_K} \max_{\theta_K} \Big\{ l(x_K, u_K, \theta_K) + G_F^*(f(x_K, u_K, \theta_K)) \Big\}, \quad (10.36)$$

which computes the costs of all optimal one-step plans from stage K to stage $F = K + 1$.

More generally, G_k^* can be computed once G_{k+1}^* is given. Carefully study (10.33), and note that it can be written as (compare to (2.9))

$$G_k^*(x_k) = \min_{u_k} \max_{\theta_k} \left\{ \min_{u_{k+1}} \max_{\theta_{k+1}} \cdots \min_{u_K} \max_{\theta_K} \left\{ l(x_k, u_k, \theta_k) + \sum_{i=k+1}^{K} l(x_i, u_i, \theta_i) + l_F(x_F) \right\} \right\} \quad (10.37)$$

by pulling the first cost term out of the sum and by separating the minimization over u_k from the rest, which range from u_{k+1} to u_K. The second min and max do not affect the $l(x_k, u_k, \theta_k)$ term; thus, $l(x_k, u_k, \theta_k)$ can be pulled outside to obtain (compare to (2.10))

$$G_k^*(x_k) = \min_{u_k} \max_{\theta_k} \left\{ l(x_k, u_k, \theta_k) + \min_{u_{k+1}} \max_{\theta_{k+1}} \cdots \min_{u_K} \max_{\theta_K} \left\{ \sum_{i=k+1}^{K} l(x_i, u_i, \theta_i) + l(x_F) \right\} \right\}. \quad (10.38)$$

The inner min's and max's represent G^*_{k+1}, which yields the recurrence (compare to (2.11))

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \Big\{ \max_{\theta_k} \Big\{ l(x_k, u_k, \theta_k) + G^*_{k+1}(x_{k+1}) \Big\} \Big\}. \tag{10.39}$$

Probabilistic case Now consider the probabilistic case. A value iteration method can be obtained by once again shadowing the presentation in Section 2.3.1.1. For k from 1 to F, let G^*_k denote the expected cost from stage k to F under the execution of the optimal plan (compare to (2.5)):

$$G^*_k(x_k) = \min_{u_k,\ldots,u_K} \left\{ E_{\theta_k,\ldots,\theta_K} \left[\sum_{i=k}^{K} l(x_i, u_i, \theta_i) + l_F(x_F) \right] \right\}. \tag{10.40}$$

The optimal cost-to-go for the boundary condition of $k = F$ again reduces to (10.34).

Once again, the algorithm makes K passes over X, each time computing G^*_k from G^*_{k+1}, as k ranges from F down to 1. As before, G^*_F is copied from l_F. In the second iteration, G^*_K is computed for each $x_K \in X$ as (compare to (2.7))

$$G^*_K(x_K) = \min_{u_K} \Big\{ E_{\theta_K} \Big[l(x_K, u_K, \theta_K) + l_F(x_F) \Big] \Big\}, \tag{10.41}$$

in which $u_K \in U(x_K)$ and the expectation occurs over θ_K. Substitutions are made into (10.41) to obtain (compare to (2.8))

$$G^*_K(x_K) = \min_{u_K} \Big\{ E_{\theta_K} \Big[l(x_K, u_K, \theta_K) + G^*_F(f(x_K, u_K, \theta_K)) \Big] \Big\}. \tag{10.42}$$

The general iteration is

$$\begin{aligned} G^*_k(x_k) = \min_{u_k} \Bigg\{ E_{\theta_k} \Bigg[\min_{u_{k+1},\ldots,u_K} \Bigg\{ E_{\theta_{k+1},\ldots,\theta_K} \Bigg[l(x_k, u_k, \theta_k) + \\ \sum_{i=k+1}^{K} l(x_i, u_i, \theta_i) + l_F(x_F) \Bigg] \Bigg\} \Bigg] \Bigg\}, \end{aligned} \tag{10.43}$$

which is obtained once again by pulling the first cost term out of the sum and by separating the minimization over u_k from the rest. The second min and expectation do not affect the $l(x_k, u_k, \theta_k)$ term, which is pulled outside to obtain (compare to (2.10))

$$\begin{aligned} G^*_k(x_k) = \min_{u_k} \Bigg\{ E_{\theta_k} \Bigg[l(x_k, u_k, \theta_k) + \\ \min_{u_{k+1},\ldots,u_K} \Bigg\{ E_{\theta_{k+1},\ldots,\theta_K} \Bigg[\sum_{i=k+1}^{K} l(x_i, u_i, \theta_i) + l(x_F) \Bigg] \Bigg\} \Bigg] \Bigg\}. \end{aligned} \tag{10.44}$$

The inner min and expectation define G^*_{k+1}, yielding the recurrence (compare to (2.11) and (10.39))

$$\begin{aligned} G^*_k(x_k) &= \min_{u_k \in U(x_k)} \Big\{ E_{\theta_k}\Big[l(x_k,u_k,\theta_k) + G^*_{k+1}(x_{k+1}) \Big] \Big\} \\ &= \min_{u_k \in U(x_k)} \Big\{ \sum_{\theta_k \in \Theta(x_k,u_k)} \Big(l(x_k,u_k,\theta_k) + G^*_{k+1}(f(x_k,u_k,\theta_k)) \Big) P(\theta_k|x_k,u_k) \Big\}. \end{aligned} \tag{10.45}$$

If the cost term does not depend on θ_k, it can be written as $l(x_k, u_k)$, and (10.45) simplifies to

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \Big\{ l(x_k,u_k) + \sum_{x_{k+1} \in X} G^*_{k+1}(x_{k+1}) P(x_{k+1}|x_k,u_k) \Big\}. \tag{10.46}$$

The dependency of state transitions on θ_k is implicit through the expression of $P(x_{k+1}|x_k, u_k)$, for which the definition uses $P(\theta_k|x_k, u_k)$ and the state transition equation f. The form given in (10.46) may be more convenient than (10.45) in implementations.

Convergence issues If the maximum number of stages is fixed in the problem definition, then convergence is assured. Suppose, however, that there is no limit on the number of stages. Recall from Section 2.3.2 that each value iteration increases the total path length by one. The actual stage indices were not important in backward dynamic programming because arbitrary shifting of indices does not affect the values. Eventually, the algorithm terminated because optimal cost-to-go values had been computed for all reachable states from the goal. This resulted in a *stationary cost-to-go function* because the values no longer changed. States that are reachable from the goal converged to finite values, and the rest remained at infinity. The only problem that prevents the existence of a stationary cost-to-go function, as mentioned in Section 2.3.2, is negative cycles in the graph. In this case, the best plan would be to loop around the cycle forever, which would reduce the cost to $-\infty$.

In the current setting, a stationary cost-to-go function once again arises, but cycles once again cause difficulty in convergence. The situation is, however, more complicated due to the influence of nature. It is helpful to consider a plan-based state transition graph, $\mathcal{G}_\pi$. First consider the nondeterministic case. If there exists a plan π from some state x_1 for which all possible actions of nature cause the traversal of cycles that accumulate negative cost, then the optimal cost-to-go at x_1 converges to $-\infty$, which prevents the value iterations from terminating. These cases can be detected in advance, and each such initial state can be avoided (some may even be in a different connected component of the state space).

It is also possible that there are unavoidable positive cycles. In Section 2.3.2, the cost-to-go function behaved differently depending on whether the goal set was

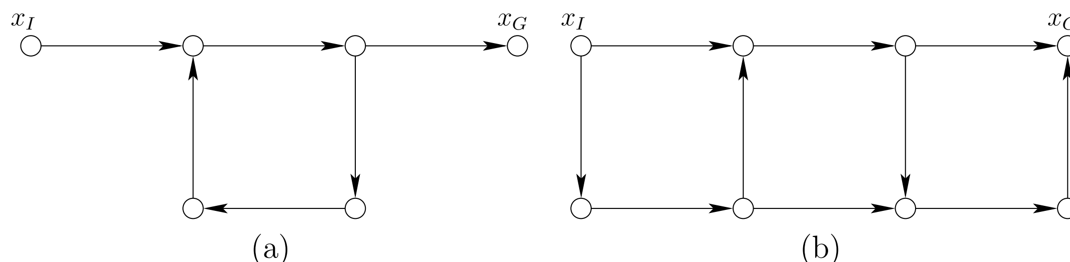


Figure 10.2: Plan-based state transition graphs. (a) The goal is possibly reachable, but not guaranteed reachable because an infinite cycle could occur. (b) The goal is guaranteed reachable because all flows lead to the goal.

reachable. Due to nature, the goal set may be possibly reachable or guaranteed reachable, as illustrated in Figure 10.2. To be *possibly reachable* from some initial state, there must exist a plan, π, for which there exists a sequence of nature actions that will lead the state into the goal set. To be *guaranteed reachable*, the goal must be reached in spite of *all* possible sequences of nature actions. If the goal is possibly reachable, but not guaranteed reachable, from some state x_1 and all edges have positive cost, then the cost-to-go value of x_1 tends to infinity as the value iterations are repeated. For example, every plan-based state transition graph may contain a cycle of positive cost, and in the worst case, nature may cause the state to cycle indefinitely. If convergence of the value iterations is only evaluated at states from which the goal set is guaranteed to be reachable, and if there are no negative cycles, then the algorithm should terminate when all cost-to-go values remain unchanged.

For the probabilistic case, there are three situations:

1. The value iterations arrive at a stationary cost-to-go function after a finite number of iterations.

2. The value iterations do not converge in any sense.

3. The value iterations converge only asymptotically to a stationary cost-to-go function. The number of iterations tends to infinity as the values converge.

The first two situations have already occurred. The first one occurs if there exists a plan, π, for which $\mathcal{G}_\pi$ has no cycles. The second situation occurs if there are negative or positive cycles for which all edges in the cycle have probability one. This situation is essentially equivalent to that for the nondeterministic case. Worst-case analysis assumes that the worst possible nature actions will be applied. For the probabilistic case, the nature actions are forced by setting their probabilities to one.

The third situation is unique to the probabilistic setting. This is caused by positive or negative cycles in $\mathcal{G}_\pi$ for which the edges have probabilities in $(0, 1)$. The optimal plan may even have such cycles. As the value iterations consider

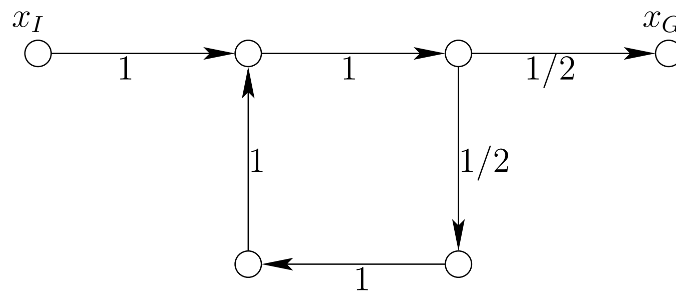


Figure 10.3: A plan-based state transition graph that causes asymptotic convergence. The probabilities of the transitions are shown on the edges. Longer and longer paths exist by traversing the cycle, but the probabilities become smaller.

longer and longer paths, a cycle may be traversed more times. However, each time the cycle is traversed, the probability diminishes. The probabilities diminish exponentially in terms of the number of stages, whereas the costs only accumulate linearly. The changes in the cost-to-go function gradually decrease and converge only to stationary values as the number of iterations tends to infinity. If some approximation error is acceptable, then the iterations can be terminated once the maximum change over all of X is within some ϵ threshold. The required number of value iterations to obtain a solution of the desired quality depends on the probabilities of following the cycles and on their costs. If the probabilities are lower, then the algorithm converges sooner.

Example 10.6 (A Cycle in the Transition Graph) Suppose that a plan, π, is chosen that yields the plan-based state transition graph shown in Figure 10.3. A probabilistic model is used, and the probabilities are shown on each edge. For simplicity, assume that each transition results in unit cost, $l(x, u, \theta) = 1$, over all x, u, and θ.

The expected cost from x_I is straightforward to compute. With probability 1/2, the cost to reach x_G is 3. With probability 1/4, the cost is 7. With probability 1/8, the cost is 11. Each time another cycle is taken, the cost increases by 4, but the probability is cut in half. This leads to the infinite series

$$G_\pi(x_I) = 3 + 4 \sum_{i=1}^{\infty} \frac{1}{2^i} = 7. \tag{10.47}$$

The infinite sum is the standard geometric series and converges to 1; hence (10.47) converges to 7.

Even though the cost converges to a finite value, this only occurs in the limit. An infinite number of value iterations would theoretically be required to obtain this result. For most applications, an approximate solution suffices, and very high precision can be obtained with a small number of iterations (e.g., after 20 iterations, the change is on the order of one-billionth). Thus, in general, it is sensible to terminate the value iterations after the maximum cost-to-go change is less than a threshold based directly on precision.

Note that if nondeterministic uncertainty is used, then the value iterations do not converge because, in the worst case, nature will cause the state to cycle forever. Even though the goal is not guaranteed reachable, the probabilistic uncertainty model allows reasonable solutions. ■

Using the plan Assume that there is no limit on the number of stages. After the value iterations terminate, cost-to-go functions are determined over X. This is not exactly a plan, because an *action* is required for each $x \in X$. The actions can be obtained by recording the $u \in U(x)$ that produced the minimum cost value in (10.45) or (10.39).

Assume that the value iterations have converged to a stationary cost-to-go function. Before uncertainty was introduced, the optimal actions were determined by (2.19). The nondeterministic and probabilistic versions of (2.19) are

$$\pi^*(x) = \operatorname*{argmin}_{u \in U(x)} \Big\{ \max_{\theta \in \Theta(x,u)} \Big\{ l(x,u,\theta) + G^*(f(x,u,\theta)) \Big\} \Big\} \tag{10.48}$$

and

$$\pi^*(x) = \operatorname*{argmin}_{u \in U(x)} \Big\{ E_\theta \Big[l(x,u,\theta) + G^*(f(x,u,\theta)) \Big] \Big\}, \tag{10.49}$$

respectively. For each $x \in X$ at which the optimal cost-to-go value is known, one evaluation of (10.45) yields the best action.

Conveniently, the optimal action can be recovered directly during execution of the plan, rather than storing actions. Each time a state x_k is obtained during execution, the appropriate action $u_k = \pi^*(x_k)$ is selected by evaluating (10.48) or (10.49) at x_k. This means that the cost-to-go function itself can be interpreted as a representation of the optimal plan, once it is understood that a local operator is required to recover the action. It may seem strange that such a local computation yields the global optimum; however, this works because the cost-to-go function already encodes the global costs. This behavior was also observed for continuous state spaces in Section 8.4.1, in which a navigation function served to define a feedback motion plan. In that context, a gradient operator was needed to recover the direction of motion. In the current setting, (10.48) and (10.49) serve the same purpose.

10.2.2 Policy Iteration

The value iterations of Section 10.2.1 work by iteratively updating cost-to-go values on the state space. The optimal plan can alternatively be obtained by iteratively searching in the space of plans. This leads to a method called *policy iteration* [84]; the term *policy* is synonymous with *plan*. The method will be explained for the case of probabilistic uncertainty, and it is assumed that X is finite. With minor adaptations, a version for nondeterministic uncertainty can also be developed.

Policy iteration repeatedly requires computing the cost-to-go for a given plan, π. Recall the definition of G_π from (10.32). First suppose that there are no uncertainties, and that the state transition equation is $x' = f(x,u)$. The dynamic programming equation (2.18) from Section 2.3.2 can be used to derive the cost-to-go for each state $x \in X$ under the application of π. Make a copy of (2.18) for each $x \in X$, and instead of the min, use the given action $u = \pi(x)$, to yield

$$G_\pi(x) = l(x,\pi(x)) + G_\pi(f(x,\pi(x))). \tag{10.50}$$

In (10.50), the G^* has been replaced by G_π because there are no variables to optimize (it is simply the cost of applying π). Equation (10.50) can be thought of as a trivial form of dynamic programming in which the choice of possible plans has been restricted to a single plan, π. If the dynamic programming recurrence (2.18) holds over the space of all plans, it must certainly hold over a space that consists of a single plan; this is reflected in (10.50).

If there are n states, (10.50) yields n equations, each of which gives an expression of $G_\pi(x)$ for a different state. For the states in which $x \in X_G$, it is known that $G_\pi(x) = 0$. Now that this is known, the cost-to-go for all states from which X_G can be reached in one stage can be computed using (10.50) with $G_\pi(f(x,\pi(x))) = 0$. Once these cost-to-go values are computed, another wave of values can be computed from states that can reach these in one stage. This process continues until the cost-to-go values are computed for all states. This is similar to the behavior of Dijkstra's algorithm.

This process of determining the cost-to-go should not seem too mysterious. Equation (10.50) indicates how the costs differ between neighboring states in the state transition graph. Since all of the differences are specified and an initial condition is given for X_G, all others can be derived by adding up the differences expressed in (10.50). Similar ideas appear in the Hamilton-Jacobi-Bellman equation and Pontryagin's minimum principle, which are covered in Section 15.2.

Now we turn to the case in which there are probabilistic uncertainties. The probabilistic analog of (2.18) is (10.49). For simplicity, consider the special case in which $l(x,u,\theta)$ does not depend on θ, which results in

$$\pi^*(x) = \operatorname*{argmin}_{u \in U(x)} \left\{ l(x,u) + \sum_{x' \in X} G^*(x') P(x'|x,u) \right\}, \tag{10.51}$$

in which $x' = f(x,u)$. The cost-to-go function, G^*, satisfies the dynamic programming recurrence

$$G^*(x) = \min_{u \in U(x)} \left\{ l(x,u) + \sum_{x' \in X} G^*(x') P(x'|x,u) \right\}. \tag{10.52}$$

The probabilistic analog to (10.50) can be made from (10.52) by restricting the set of actions to a single plan, π, to obtain

$$G_\pi(x) = l(x,\pi(x)) + \sum_{x' \in X} G_\pi(x') P(x'|x,\pi(x)), \tag{10.53}$$

POLICY ITERATION ALGORITHM

1. Pick an initial plan π, in which u_T is applied at each $x \in X_G$ and all other actions are chosen arbitrarily.

2. Use (10.53) to compute G_π for each $x \in X$ under the plan π.

3. Substituting the computed G_π values for G^*, use (10.51) to compute a better plan, π':

$$\pi'(x) = \underset{u \in U(x)}{\operatorname{argmin}} \Big\{ l(x,u) + \sum_{x' \in X} G_\pi(x') P(x'|x,u) \Big\}. \tag{10.54}$$

4. If π' produces at least one lower cost-to-go value than π, then let $\pi = \pi'$ and repeat Steps 2 and 3. Otherwise, declare π to be the optimal plan, π^*.

Figure 10.4: The policy iteration algorithm iteratively searches the space of plans by evaluating and improving plans.

in which x' is the next state.

The cost-to-go for each $x \in X$ under the application of π can be determined by writing (10.53) for each state. Note that all quantities except G_π are known. This means that if there are n states, then there are n linear equations and n unknowns ($G_\pi(x)$ for each $x \in X$). The same was true when (10.50) was used, except the equations were much simpler. In the probabilistic setting, a system of n linear equations must be solved to determine G_π. This may be performed using classical linear algebra techniques, such as *singular value decomposition (SVD)* [399, 961].

Now that we have a method for evaluating the cost of a plan, the policy iteration method is straightforward, as specified in Figure 10.4. Note that in Step 3, the cost-to-go G_π, which was developed for one plan, π, is used to evaluate other plans. The result is the cost that will be obtained if a new action is tried in the first stage and then π is used for all remaining stages. If a new action cannot reduce the cost, then π must have already been optimal because it means that (10.54) has become equivalent to the stationary dynamic programming equation, (10.49). If it is possible to improve π, then a new plan is obtained. The new plan must be strictly better than the previous plan, and there is only a finite number of possible plans in total. Therefore, the policy iteration method converges after a finite number of iterations.

Example 10.7 (An Illustration of Policy Iteration) A simple example will now be used to illustrate policy iteration. Let $X = \{a, b, c\}$ and $U = \{1, 2, u_T\}$. Let $X_G = \{c\}$. Let $l(x,u) = 1$ for all $x \in X$ and $u \in U \setminus \{u_T\}$ (if u_T is applied, there is no cost). The probabilistic state transition graphs for each action are shown in Figure 10.5. The first step is to pick an initial plan. Let $\pi(a) = 1$ and $\pi(b) = 1$; let $\pi(c) = u_T$ because $c \in X_G$.

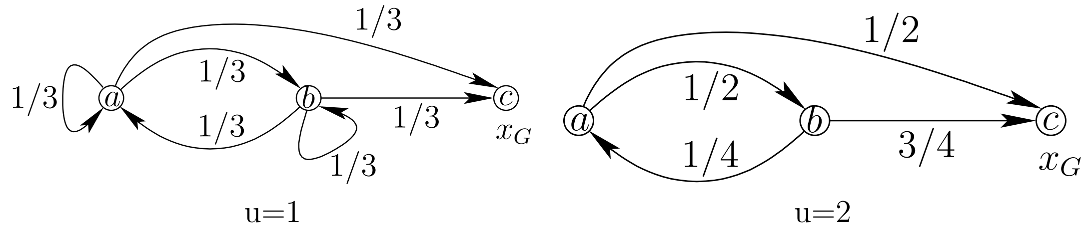


Figure 10.5: The probabilistic state transition graphs for $u = 1$ and $u = 2$. Transitions out of c are not shown because it is assumed that a termination action is always applied from x_g.

Now use (10.53) to compute G_π. This yields three equations:

$$G_\pi(a) = 1 + G_\pi(a)P(a \mid a, 1) + G_\pi(b)P(b \mid a, 1) + G_\pi(c)P(c \mid a, 1) \tag{10.55}$$

$$G_\pi(b) = 1 + G_\pi(a)P(a \mid b, 1) + G_\pi(b)P(b \mid b, 1) + G_\pi(c)P(c \mid b, 1) \tag{10.56}$$

$$G_\pi(c) = 0 + G_\pi(a)P(a \mid c, u_T) + G_\pi(b)P(b \mid c, u_T) + G_\pi(c)P(c \mid c, u_T). \tag{10.57}$$

Each equation represents a different state and uses the appropriate action from π. The final equation reduces to $G_\pi(c) = 0$ because of the basic rules of applying a termination condition. After substituting values for $P(x'|x,u)$ and using $G_\pi(c) = 0$, the other two equations become

$$G_\pi(a) = 1 + \tfrac{1}{3}G_\pi(a) + \tfrac{1}{3}G_\pi(b) \tag{10.58}$$

and

$$G_\pi(b) = 1 + \tfrac{1}{3}G_\pi(a) + \tfrac{1}{3}G_\pi(b). \tag{10.59}$$

The solutions are $G_\pi(a) = G_\pi(b) = 3$.

Now use (10.54) for each state with $G_\pi(a) = G_\pi(b) = 3$ and $G_\pi(c) = 0$ to find a better plan, π'. At state a, it is found by solving

$$\pi'(a) = \underset{u \in U}{\operatorname{argmin}} \Big\{ l(x,a) + \sum_{x' \in X} G_\pi(x')P(x'|x,a) \Big\}. \tag{10.60}$$

The best action is $u = 2$, which produces cost $5/2$ and is computed as

$$l(x,2) + \sum_{x' \in X} G_\pi(x')P(x'|x,2) = 1 + 0 + (3)\tfrac{1}{2} + (0)\tfrac{1}{4} = \tfrac{5}{2}. \tag{10.61}$$

Thus, $\pi'(a) = 2$. Similarly, $\pi'(b) = 2$ can be computed, which produces cost $7/4$. Once again, $\pi'(c) = u_T$, which completes the determination of an improved plan.

Since an improved plan has been found, replace π with π' and return to Step 2. The new plan yields the equations

$$G_\pi(a) = 1 + \tfrac{1}{2}G_\pi(b) \tag{10.62}$$

and

$$G_\pi(b) = 1 + \tfrac{1}{4} G_\pi(a). \tag{10.63}$$

Solving these yields $G_\pi(a) = 12/7$ and $G_\pi(b) = 10/7$. The next step attempts to find a better plan using (10.54), but it is determined that the current plan cannot be improved. The policy iteration method terminates by correctly reporting that $\pi^* = \pi$. ■

Policy iteration may appear preferable to value iteration, especially because it usually converges in fewer iterations than value iteration. The equation solving that determines the cost of a plan effectively considers multiple stages at once. However, for most planning problems, X is large and the large linear system of equations that must be solved at every iteration can become unwieldy. In some applications, either the state space may be small enough or sparse matrix techniques may allow efficient solutions over larger state spaces. In general, value-based methods seem preferable for most planning problems.

10.2.3 Graph Search Methods

Value iteration is quite general; however, in many instances, most of the time is wasted on states that do not update their values because either the optimal cost-to-go is already known or the goal is not yet reached. Policy iteration seems to alleviate this problem, but it is limited to small state spaces. These shortcomings motivate the consideration of alternatives, such as extending the graph search methods of Section 2.2. In some cases, Dijkstra's algorithm can even be extended to quickly obtain optimal solutions, but a strong assumption is required on the structure of solutions. In the nondeterministic setting, search methods can be developed that produce only feasible solutions, without regard for optimality. For the methods in this section, X need not be finite, as long as the search method is systematic, in the sense defined in Section 2.2.

Backward search with backprojections A backward search can be conducted by incrementally growing a plan outward from X_G by using backprojections. A complete algorithm for computing feasible plans under nondeterministic uncertainty is outlined in Figure 10.6. Let S denote the set of states for which the plan has been computed. Initially, $S = X_G$ and, if possible, S may grow until $S = X$. The plan definition starts with $\pi(x) = u_T$ for each $x \in X_G$ and is incrementally extended to new states during execution.

Step 2 takes every state x that is not already in S and checks whether it should be added. This requires determining whether some action, u, can be applied from x, with the next state guaranteed to lie in S, as shown in Figure 10.7. If so, then $\pi(x) = u$ is assigned and S is extended to include x. If no such progress can be made, then the algorithm must terminate. Otherwise, every state is checked again

BACKPROJECTION ALGORITHM

1. Initialize $S = X_G$, and let $\pi(x) = u_T$ for each $x \in X_G$.
2. For each $x \in X \setminus S$, if there exists some $u \in U(x)$ such that $x \in \mathrm{SB}(S, u)$ then: 1) let $\pi(x) = u$, and 2) insert x into S.
3. If Step 2 failed to extend S, then exit. This implies that $\mathrm{SB}(S) = S$, which means no more progress can be made. Otherwise, go to Step 2.

Figure 10.6: A general algorithm for computing a feasible plan under nondeterministic uncertainty.

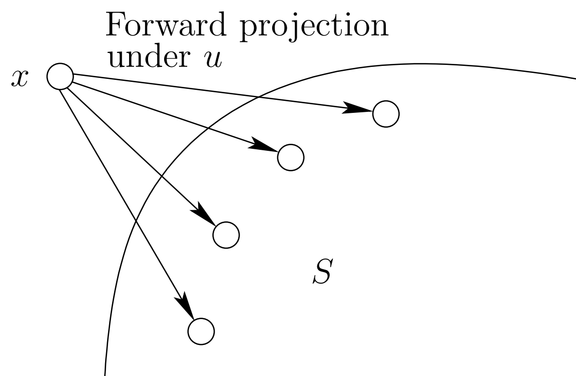


Figure 10.7: A state x can be added to S if there exists an action $u \in U(x)$ such that the one-stage forward projection is contained in S.

by returning to Step 2. This is necessary because S has grown, and in the next iteration new states may lie in its strong backprojection.

For efficiency reasons, the $X \setminus S$ set in Step 2 may be safely replaced with the smaller set, $\mathrm{WB}(S) \setminus S$, because it is impossible for other states in X to be affected. Depending on the problem, this condition may provide a quick way to prune many hopeless states from consideration. As an example, consider a grid-like environment in which a maximum of two steps in any direction is possible at a given time. A simple distance test can be implemented to eliminate many states from possible inclusion into S in Step 2.

As long as the consideration of states to include in S is systematic, as considered in Section 2.2, numerous variations of the algorithm in Figure 10.6 are possible. One possibility is to keep track of the cost-to-go and grow S based on incrementally inserting minimal-cost states. This leads to a nondeterministic version of Dijkstra's algorithm, which is covered next.

Nondeterministic Dijkstra Figure 10.8 shows an extension of Dijkstra's algorithm for solving the problem of Formulation 10.1 under nondeterministic uncertainty. It can also be considered as a variant of the algorithm in Figure 10.6

NONDETERMINISTIC DIJKSTRA

1. Initialize $S = \emptyset$ and $A = X_G$. Associate u_T with every $x \in A$. Assign $G(x) = 0$ for all $x \in A$ and $G(x) = \infty$ for all other states.
2. Unless A is empty, remove the $x_s \in A$ and its corresponding u, for which G is smallest. If A was empty, then exit (no further progress is possible).
3. Designate $\pi^*(x_s) = u$ as part of the optimal plan and insert x_s into S. Declare $G^*(x_s) = G(x_s)$.
4. Compute $G(x)$ using (10.64) for any x in the frontier set, $\mathrm{Front}(x_s, S)$, and insert $\mathrm{Front}(x_s, S)$ into A and with associated actions for each inserted state. For states already in A, retain whichever G value is lower, either its original value or the new computed value. Go to Step 2.

Figure 10.8: A Dijkstra-based algorithm for computing an optimal feasible plan under nondeterministic uncertainty.

because it grows S by using backprojections. The algorithm in Figure 10.8 represents a backward-search version of Dijkstra's algorithm; therefore, it maintains the worst-case cost-to-go, G, which sometimes becomes the optimal, worst-case cost-to-go, G^*. Initially, $G = 0$ for states in the goal, and $G = \infty$ for all others.

Step 1 performs the initialization. Step 2 selects the state in A that has the smallest value. As in Dijkstra's algorithm for deterministic problems, it is known that the cost-to-go for this state is the smallest possible. It is therefore declared in Step 3 that $G^*(x_s) = G(x_s)$, and π^* is extended to include x_s.

Step 4 updates the costs for some states and expands the active set, A. Which costs could be immediately affected by the insertion of x_s into S? These are states $x_k \in X \setminus S$ for which there exists some $u_k \in U(x_k)$ that produces a one-stage forward projection, $X_{k+1}(x_k, u_k)$, such that: 1) $x_s \in X_{k+1}(x_k, u_k)$ and 2) $X_{k+1}(x_k, u_k) \subseteq S$. This is depicted in Figure 10.9. Let the set of states that satisfy these constraints be called the *frontier set*, denoted by $\mathrm{Front}(x_s, S)$. For each $x \in \mathrm{Front}(x_s, S)$, let $U_f(x) \subseteq U(x)$ denote the set of all actions for which the forward projection satisfies the two previous conditions.

The frontier set can be interpreted in terms of backprojections. The weak backprojection $\mathrm{WB}(x_s)$ yields all states that can possibly reach x_s in one step. However, the cost-to-go is only finite for states in $\mathrm{SB}(S)$ (here S already includes x_s). The states in S should certainly be excluded because their optimal costs are already known. These considerations reduce the set of candidate frontier states to $(\mathrm{WB}(x_s) \cap \mathrm{SB}(S)) \setminus S$. This set is still too large because the same action, u, must produce a one-stage forward projection that includes x_s and is a subset of S.

The worst-case cost-to-go is computed for all $x \in \mathrm{Front}(x_s, S)$ as

$$G(x) = \min_{u \in U_f(x)} \Big\{ \max_{\theta \in \Theta(x,u)} \Big\{ l(x, u, \theta) + G(f(x, u, \theta)) \Big\} \Big\}, \tag{10.64}$$

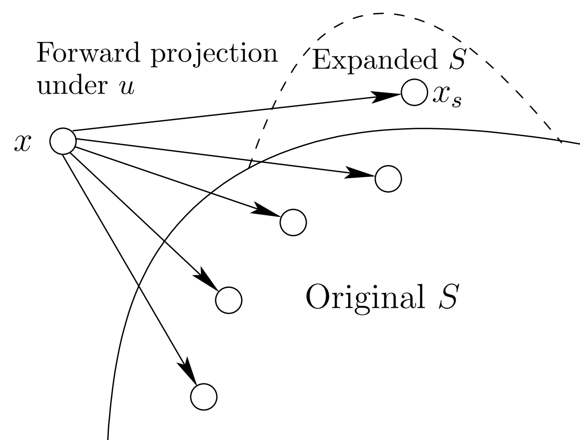


Figure 10.9: The worst-case cost-to-go is computed for any state x such that there exists a $u \in U(x)$ for which the one-stage forward projection is contained in the updated S and one state in the forward projection is x_s.

in which the restricted action set, $U_f(x)$, is used. If x was already in A and a previous $G(x)$ was computed, then the minimum of its previous value and (10.64) is kept.

Probabilistic Dijkstra A probabilistic version of Dijkstra's algorithm does not exist in general; however, for some problems, it can be made to work. The algorithm in Figure 10.8 is adapted to the probabilistic case by using

$$G(x) = \min_{u \in U_f(x)} \Big\{ E_\theta \Big[l(x,u,\theta) + G(f(x,u,\theta)) \Big] \Big\} \tag{10.65}$$

in the place of (10.64). The definition of Front remains the same, and the nondeterministic forward projections are still applied to the probabilistic problem. Only edges in the transition graph that have nonzero probability are actually considered as possible future states. Edges with zero probability are precluded from the forward projection because they cannot affect the computed cost values.

The probabilistic version of Dijkstra's algorithm can be successfully applied if there exists a plan, π, for which from any $x_k \in X$ there is probability one that $G_\pi(x_{k+1}) < G_\pi(x_k)$. What does this condition mean? From any x_k, all possible next states that have nonzero probability of occurring must have a lower cost value. If all edge costs are positive, this means that all paths in the multi-stage forward projection will make monotonic progress toward the goal. In the deterministic case, this always occurs if $l(x,u)$ is always positive. If nonmonotonic paths are possible, then Dijkstra's algorithm breaks down because the region in which cost-to-go values change is no longer contained within a propagating band, which arises in Dijkstra's algorithm for deterministic problems.

10.3 Infinite-Horizon Problems

In stochastic control theory and artificial intelligence research, most problems considered to date do not specify a goal set. Therefore, there are no associated termination actions. The task is to develop a plan that minimizes the expected cost (or maximize expected reward) over some number of stages. If the number of stages is finite, then it is straightforward to apply the value iteration method of Section 10.2.1. The adapted version of backward value iteration simply terminates when the first stage is reached. The problem becomes more challenging if the number of stages is infinite. This is called an *infinite-horizon problem.*

The number of stages for the planning problems considered in Section 10.1 is also infinite; however, it was expected that if the goal could be reached, termination would occur in a finite number of iterations. If there is no termination condition, then the costs tend to infinity. There are two alternative cost models that force the costs to become finite. The *discounted cost model* shrinks the per-stage costs as the stages extend into the future; this yields a geometric series for the total cost that converges to a finite value. The *average cost-per-stage model* divides the total cost by the number of stages. This essentially normalizes the accumulating cost, once again preventing its divergence to infinity. Some of the computation methods of Section 10.2 can be adapted to these models. This section formulates these two infinite-horizon cost models and presents computational solutions.

10.3.1 Problem Formulation

Both of the cost models presented in this section were designed to force the cumulative cost to become finite, even though there is an infinite number of stages. Each can be considered as a minor adaptation of cost functional used in Formulation 10.1.

The following formulation will be used throughout Section 10.3.

Formulation 10.2 (Infinite-Horizon Problems)

1. A nonempty, finite *state space* X.

2. For each state $x \in X$, a finite *action space* $U(x)$ (there is no termination action, contrary to Formulation 10.1).

3. A finite *nature action space* $\Theta(x,u)$ for each $x \in X$ and $u \in U(x)$.

4. A *state transition function* f that produces a state, $f(x,u,\theta)$, for every $x \in X$, $u \in U(x)$, and $\theta \in \Theta(x,u)$.

5. A set of *stages*, each denoted by k, that begins at $k = 1$ and continues indefinitely.

6. A stage-additive cost functional, $L(\tilde{x}, \tilde{u}, \tilde{\theta})$, in which $\tilde{x}$, $\tilde{u}$, and $\tilde{\theta}$ are infinite state, action, and nature histories, respectively. Two alternative forms of L will be given shortly.

In comparison to Formulation 10.1, note that here there is no initial or goal state. Therefore, there are no termination actions. Without the specification of a goal set, this may appear to be a strange form of planning. A feedback plan, π, still takes the same form; $\pi(x)$ produces an action $u \in U(x)$ for each $x \in X$.

As a possible application, imagine a robot that delivers materials in a factory from several possible source locations to several destinations. The robot operates over a long work shift and has a probabilistic model of when requests to deliver materials will arrive. Formulation 10.2 can be used to define a problem in which the goal is to minimize the average amount of time that materials wait to be delivered. This strategy should not depend on the length of the shift; therefore, an infinite number of stages is reasonable. If the shift is too short, the robot may focus only on one delivery, or it may not even have enough time to accomplish that.

Discounted cost In Formulation 10.2, the cost functional in Item 6 must be defined carefully to ensure that finite values are always obtained, even though the number of stages tends to infinity. The *discounted cost model* provides one simple way to achieve this by rapidly decreasing costs in future stages. Its definition is based on the standard geometric series. For any real parameter $\alpha \in (0, 1)$,

$$\lim_{K \to \infty} \left(\sum_{k=0}^{K} \alpha^k \right) = \frac{1}{1 - \alpha}. \tag{10.66}$$

The simplest case, $\alpha = 1/2$, yields $1+1/2+1/4+1/8+\cdots$, which clearly converges to 2.

Now let $\alpha \in (0, 1)$ denote a *discount factor*, which is applied in the definition of a cost functional:

$$L(\tilde{x}, \tilde{u}, \tilde{\theta}) = \lim_{K \to \infty} \left(\sum_{k=0}^{K} \alpha^k l(x_k, u_k, \theta_k) \right). \tag{10.67}$$

Let l_k denote the cost, $l(x_k, u_k, \theta_k)$, received at stage k. For convenience in this setting, the first stage is $k = 0$, as opposed to $k = 1$, which has been used previously. As the maximum stage, K, increases, the diminished importance of costs far in the future can easily be observed, as indicated in Figure 10.10.

The rate of cost decrease depends strongly on α. For example, if $\alpha = 1/2$, the costs decrease very rapidly. If $\alpha = 0.999$, the convergence to zero is much slower. The trade-off is that with a large value of α, more stages are taken into account, and the designed plan is usually of higher quality. If a small value of α is used, methods such as value iteration converge much more quickly; however, the solution quality may be poor because of "short sightedness."

Stage	L_K^*
$K = 0$	l_0
$K = 1$	$l_0 + \alpha l_1$
$K = 2$	$l_0 + \alpha l_1 + \alpha^2 l_2$
$K = 3$	$l_0 + \alpha l_1 + \alpha^2 l_2 + \alpha^3 l_3$
$K = 4$	$l_0 + \alpha l_1 + \alpha^2 l_2 + \alpha^3 l_3 + \alpha^4 l_4$
$\vdots$	

Figure 10.10: The cost magnitudes decease exponentially over the stages.

The term $l(x_k, u_k, \theta_k)$ in (10.67) assumes different values depending on x_k, u_k, and θ_k. Since there are only a finite number of possibilities, they must be bounded by some positive constant c.[1] Hence,

$$\lim_{K \to \infty} \left(\sum_{k=0}^{K} \alpha^k l(x_k, u_k, \theta_k) \right) \leq \lim_{K \to \infty} \left(\sum_{k=0}^{K} \alpha^k c \right) \leq \frac{c}{1 - \alpha}, \tag{10.68}$$

which means that $L(\tilde{x}, \tilde{u}, \tilde{\theta})$ is bounded from above, as desired. A similar lower bound can be constructed, which ensures that the resulting total cost is always finite.

Average cost-per-stage An alternative to discounted cost is to use the *average cost-per-stage model*, which keeps the cumulative cost finite by dividing out the total number of stages:

$$L(\tilde{x}, \tilde{u}, \tilde{\theta}) = \lim_{K \to \infty} \left(\frac{1}{K} \sum_{k=0}^{K-1} l(x_k, u_k, \theta_k) \right). \tag{10.69}$$

Using the maximum per-stage cost bound c, it is clear that (10.69) grows no larger than c, even as $K \to \infty$. This model is sometimes preferable because the cost does not depend on an arbitrary parameter, α.

10.3.2 Solution Techniques

Straightforward adaptations of the value and policy iteration methods of Section 10.2 exist for infinite-horizon problems. These will be presented here; however, it is important to note that many other important issues exist regarding their convergence and numerical stability [96]. There are several other variants of these algorithms that are too involved to cover here but nevertheless are important because they address many of these additional issues. The main point in this section is to understand the simple relationship to the problems considered so far in Sections 10.1 and 10.2.

[1] The state space X may even be infinite, but this requires that the set of possible costs is bounded.

Value iteration for discounted cost A backward value iteration solution will be presented that follows naturally from the method given in Section 10.2.1. For notational convenience, let the first stage be designated as $k = 0$ so that α^{k-1} may be replaced by α^k. In the probabilistic case, the expected optimal cost-to-go is

$$G^*(x) = \lim_{K\to\infty} \left(\min_{\tilde{u}} \left\{ E_{\tilde{\theta}} \left[\sum_{k=1}^{K} \alpha^k l(x_k, u_k, \theta_k) \right] \right\} \right). \tag{10.70}$$

The expectation is taken over all nature histories, each of which is an infinite sequence of nature actions. The corresponding expression for the nondeterministic case is

$$G^*(x) = \lim_{K\to\infty} \left(\min_{\tilde{u}} \left\{ \max_{\tilde{\theta}} \left\{ \sum_{k=1}^{K} \alpha^k l(x_k, u_k, \theta_k) \right\} \right\} \right). \tag{10.71}$$

Since the probabilistic case is more common, it will be covered here. The nondeterministic version is handled in a similar way (see Exercise 17). As before, backward value iterations will be performed because they are simpler to express. The discount factor causes a minor complication that must be fixed to make the dynamic programming recurrence work properly.

One difficulty is that the stage index now appears in the cost function, in the form of α^k. This means that the shift-invariant property from Section 2.3.1.1 is no longer preserved. We must therefore be careful about assigning stage indices. This is a problem because for backward value iteration the final stage index has been unknown and unimportant.

Consider a sequence of discounted decision-making problems, by increasing the maximum stage index: $K = 0$, $K = 1$, $K = 2$, ... Look at the neighboring cost expressions in Figure 10.10. What is the difference between finding the optimal cost-to-go for the $K+1$-stage problem and the K-stage problem? In Figure 10.10 the last four terms of the cost for $K = 4$ can be obtained by multiplying all terms for $K = 3$ by α and adding a new term, l_0. The only difference is that the stage indices need to be shifted by one on each l_i that was borrowed from the $K = 3$ case. In general, the optimal costs of a K-stage optimization problem can serve as the optimal costs of the $K + 1$-stage problem if they are first multiplied by α. The $K + 1$-stage optimization problem can be solved by optimizing over the sum of the first-stage cost plus the optimal cost for the K-stage problem, discounted by α.

This can be derived using straightforward dynamic programming arguments as follows. Suppose that K is fixed. The cost-to-go can be expressed recursively for k from 0 to K as

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \left\{ E_{\theta_k} \left[\alpha^k l(x_k, u_k, \theta_k) + G^*_{k+1}(x_{k+1}) \right] \right\}, \tag{10.72}$$

in which $x_{k+1} = f(x_k, u_k, \theta_k)$. The problem, however, is that the recursion depends on k through α^k, which makes it appear nonstationary.

The idea of using neighboring cost values as shown in Figure 10.10 can be applied by making a notational change. Let $J^*_{K-k}(x_k) = \alpha^{-k} G^*_k(x_k)$. This reverses the direction of the stage indices to avoid specifying the final stage and also scales by α^{-k} to correctly compensate for the index change. Substitution into (10.72) yields

$$\alpha^k J^*_{K-k}(x_k) = \min_{u_k \in U(x_k)} \Big\{ E_{\theta_k} \Big[\alpha^k l(x_k, u_k, \theta_k) + \alpha^{k+1} J^*_{K-k-1}(x_{k+1}) \Big] \Big\}. \tag{10.73}$$

Dividing by α^k and then letting $i = K - k$ yields

$$J^*_i(x_k) = \min_{u_k \in U(x_k)} \Big\{ E_{\theta_k} \Big[l(x_k, u_k, \theta_k) + \alpha J^*_{i-1}(x_{k+1}) \Big] \Big\}, \tag{10.74}$$

in which J^*_i represents the expected cost for a finite-horizon discounted problem in which $K = i$. Note that (10.74) expresses J^*_i in terms of J^*_{i-1}, but x_k is given, and the right-hand side uses x_{k+1}. The indices appear to run in opposite directions because this is simply backward value iteration with a notational change that reverses some of the indices. The particular stage indices of x_k and x_{k+1} are not important in (10.74), as long as $x_{k+1} = f(x_k, u_k, \theta_k)$ (for example, the substitutions $x = x_k$, $x' = x_{k+1}$, $u = u_k$, and $\theta = \theta_k$ can be safely made).

Value iteration proceeds by first letting $J^*_0(x_0) = 0$ for all $x \in X$. Successive cost-to-go functions are computed by iterating (10.74) over the state space. Under the cycle-avoiding assumptions of Section 10.2.1, the convergence is usually asymptotic due to the infinite horizon. The discounting gradually causes the cost differences to diminish until they are within the desired tolerance. The stationary form of the dynamic programming recurrence, which is obtained in the limit as i tends to infinity, is

$$J^*(x) = \min_{u \in U(x)} \Big\{ E_{\theta_k} \Big[l(x, u, \theta) + \alpha J^*(f(x, u, \theta)) \Big] \Big\}. \tag{10.75}$$

If the cost terms do not depend on nature, then the simplified form is

$$J^*(x) = \min_{u \in U(x)} \Big\{ l(x, u) + \alpha \sum_{x' \in X} J^*(x') P(x'|x, u) \Big\}. \tag{10.76}$$

As explained in Section 10.2.1, the optimal action, $\pi^*(x)$, is assigned as the $u \in U(x)$ that satisfies (10.75) or (10.76) at x.

Policy iteration for discounted cost The policy iteration method may alternatively be applied to the probabilistic discounted-cost problem. Recall the method given in Figure 10.4. The general approach remains the same: A search is conducted over the space of plans by solving a linear system of equations in each iteration. In Step 2, (10.53) is replaced by

$$J_\pi(x) = l(x, u) + \alpha \sum_{x' \in X} J_\pi(x') P(x'|x, u), \tag{10.77}$$

which is a special form of (10.76) for evaluating a fixed plan. In Step 3, (10.54) is replaced by

$$\pi'(x) = \operatorname*{argmin}_{u \in U(x)} \Big\{ l(x,u) + \alpha \sum_{x' \in X} J_\pi(x') P(x'|x,u) \Big\}. \tag{10.78}$$

Using these alterations, the policy iteration algorithm proceeds in the same way as in Section 10.2.2.

Solutions for the average cost-per-stage model A value iteration algorithm for the average cost model can be obtained by simply neglecting to divide by K. Selecting actions that optimize the total cost also optimizes the average cost as the number of stages approaches infinity. This may cause costs to increase toward $\pm\infty$; however, only a finite number of iterations can be executed in practice.

The backward value iterations of Section 10.2.1 can be followed with very little modification. Initially, let $G^*(x_F) = 0$ for all $x_F \in X$. The value iterations are computed using the standard form

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \Bigg\{ \sum_{\theta \in \Theta(x_k,u_k)} \Big(l(x_k,u_k,\theta_k) + G^*_{k+1}(f(x_k,u_k,\theta_k)) \Big) P(\theta_k|x_k,u_k) \Bigg\}. \tag{10.79}$$

The iterations continue until convergence occurs. To determine whether a solution of sufficient quality has been obtained, a reasonable criterion for is

$$\max_{x \in X} \Big\{ \big| G^*_k(x)/N - G^*_{k+1}(x)/(N-1) \big| \Big\} < \epsilon, \tag{10.80}$$

in which ϵ is the error tolerance and N is the number of value iterations that have been completed (it is required in (10.80) that $N > 1$). Once (10.80) has been satisfied, the iterations can be terminated.

A numerical problem may exist with the growing values obtained for $G^*(x)$. This can be alleviated by periodically reducing all values by some constant factor to ensure that the numbers fit within the allowable floating point range. In [96], a method called *relative value iteration* is presented, which selects one state, $s \in X$, arbitrarily and expresses the cost-to-go values by subtracting off the cost at s. This trims down all values simultaneously to keep them bounded while still maintaining the convergence properties of the algorithm.

Policy iteration can alternatively be performed by using the method given in Figure 10.4 with only minor modification.

10.4 Reinforcement Learning

10.4.1 The General Philosophy

This section briefly introduces the basic ideas of a framework that has been highly popular in the artificial intelligence community in recent years. It was developed

and used primarily by machine learning researchers [19, 930], and therefore this section is called *reinforcement learning*. The problem generally involves computing optimal plans for probabilistic infinite-horizon problems. The basic idea is to combine the problems of learning the probability distribution, $P(\theta|x,u)$, and computing the optimal plan into the same algorithm.

Terminology Before detailing the method further, some explanation of existing names seems required. Consider the term *reinforcement learning*. In machine learning, most decision-theoretic models are expressed in terms of *reward* instead of *cost*. Thus, the task is to make decisions or find plans that *maximize* a *reward functional*. Choosing good actions under this model appears to provide positive reinforcement in the form of a reward. Therefore, the term *reinforcement* is used. Using cost and minimization instead, some alternative names may be *decision-theoretic learning* or *cost-based learning*.

The term *learning* is associated with the problem because estimating the probability distribution $P(\theta|x,u)$ or $P(x'|x,u)$ is clearly a learning problem. However, it is important to remember that there is also the planning problem of computing cost-to-go functions (or reward-to-go functions) and determining a plan that optimizes the costs (or rewards). Therefore, the term *reinforcement planning* may be just as reasonable.

The general framework is referred to as *neuro-dynamic programming* in [97] because the formulation and resulting algorithms are based on dynamic programming. Most often, a variant of value iteration is obtained. The *neuro* part refers to a family of functions that can be used to approximate plans and cost-to-go values. This term is fairly specific, however, because other function families may be used. Furthermore, for some problems (e.g., over small, finite state spaces), the cost values and plans are represented without approximation.

The name *simulation-based methods* is used in [95], which is perhaps one of the most accurate names (when used in the context of dynamic programming). Thus, *simulation-based dynamic programming* or *simulation-based planning* nicely reflects the framework explained here. The term *simulation* comes from the fact that a Monte Carlo simulator is used to generate samples for which the required distributions are learned during planning. You are, of course, welcome to use your favorite name, but keep in mind that under all of the names, the idea remains the same. This will be helpful to remember if you intend to study related literature.

The general framework The framework is usually applied to infinite-horizon problems under probabilistic uncertainty. The discounted-cost model is most popular; however, we will mostly work with Formulation 10.1 because it is closer to the main theme of this book. It has been assumed so far that when planning under Formulation 10.1, all model components are known, including $P(x_{k+1}|x_k,u_k)$. This can be considered as a *traditional* framework, in which there are three general phases:

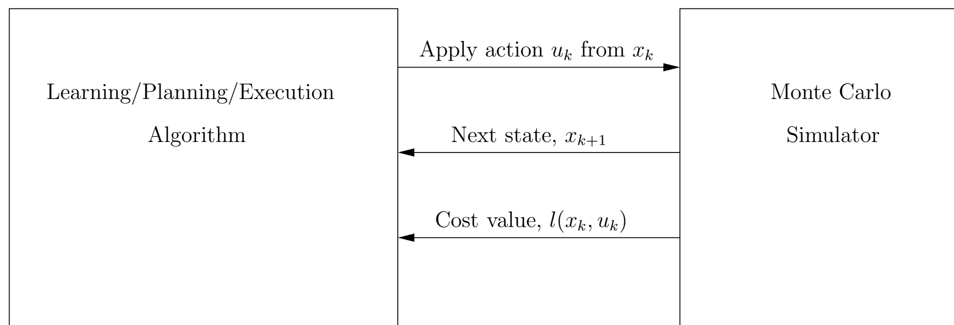


Figure 10.11: The general framework for reinforcement learning (or simulation-based dynamic programming).

> **Learning phase:** The transition probabilities are estimated by visiting states in X, trying actions, and gathering statistics. When this phase concludes, the model of the environment is completely known.
>
> **Planning phase:** An algorithm computes a feedback plan using a method such as value iteration or policy iteration.
>
> **Execution phase:** The plan is executed on a machine that is connected to the same environment on which the learning phase was applied.

The simulation-based framework combines all three of these phases into one. Learning, planning, and execution are all conducted by a machine that initially knows nothing about the state transitions or even the cost terms. Ideally, the machine should be connected to a physical environment for which the Markov model holds. However, in nearly all implementations, the machine is instead connected to a Monte Carlo simulator as shown in Figure 10.11. Based on the current state, the algorithm sends an action, u_k, to the simulator, and the simulator computes its effect by sampling according to its internal probability distributions. Obviously, the designer of the simulator knows the transition probabilities, but these are not given directly to the planning algorithm. The simulator then sends the next state, x_{k+1}, and cost, $l(x_k, u_k)$, back to the algorithm.

For simplicity, $l(x_k, u_k)$ is used instead of allowing the cost to depend on the particular nature action, which would yield $l(x_k, u_k, \theta_k)$. The explicit characterization of nature is usually not needed in this framework. The probabilities $P(x_{k+1}|x_k, u_k)$ are directly learned without specifying nature actions. It is common to generalize the cost term from $l(x_k, u_k)$ to $l(x_k, u_k, x_{k+1})$, but this is avoided here for notational convenience. The basic ideas remain the same, and only slight variations of the coming equations are needed to handle this generalization.

The simulator is intended to simulate "reality," in which the machine interacts with the physical world. It replaces the environment in Figure 1.16b from Section 1.4. Using the interpretation of that section, the algorithms presented in this context can be considered as a plan as shown in Figure 1.18b. If the learning component is terminated, then the resulting feedback plan can be programmed into

another machine, as shown in Figure 1.18a. This step is usually not performed, however, because often it is assumed that the machine continues to learn over its lifetime.

One of the main issues is *exploration vs. exploitation* [930]. Some repetitive exploration of the state space is needed to gather enough data that reliably estimate the model. For true theoretical convergence, each state-action pair must be tried infinitely often. On the other hand, information regarding the model should be exploited to efficiently accomplish tasks. These two goals are often in conflict. Focusing too much on exploration will not optimize costs. Focusing too much on exploitation may prevent useful solutions from being developed because better alternatives have not yet been discovered.

10.4.2 Evaluating a Plan via Simulation

The simulation method is based on averaging the information gained incrementally from samples. Suppose that you receive a sequence of costs, c_1, c_2, $\ldots$, and would like to incrementally compute their average. You are not told the total number of samples in advance, and at any point you are required to report the current average. Let m_i denote the average of the first i samples,

$$m_i = \frac{1}{i} \sum_{j=1}^{i} c_j. \tag{10.81}$$

To efficiently compute m_i from m_{i-1}, multiply m_{i-1} by $i-1$ to recover the total, add c_i, and then divide by i:

$$m_i = \frac{(i-1)m_{i-1} + c_i}{i}. \tag{10.82}$$

This can be manipulated into

$$m_i = m_{i-1} + \frac{1}{i}(c_i - m_{i-1}). \tag{10.83}$$

Now consider the problem of estimating the expected cost-to-go, $G_\pi(x)$, at every $x \in X$ for some fixed plan, π. If $P(x'|x,u)$ and the costs $l(x,u)$ were known, then it could be computed by solving

$$G_\pi(x) = l(x,u) + \sum_{x'} P(x'|x,u) G_\pi(x'). \tag{10.84}$$

However, without this information, we must rely on the simulator.

From each $x \in X$, suppose that 1000 trials are conducted, and the resulting costs to get to the goal are recorded and averaged. Each trial is an iterative process in which π selects the action, and the simulator indicates the next state and its incremental cost. Once the goal state is reached, the costs are totaled to yield the

measured cost-to-go for that trial (this assumes that $\pi(x) = u_T$ for all $x \in X_G$). If c_i denotes this total cost at trial i, then the average, m_i, over i trials provides an estimate of $G_\pi(x)$. As i tends to infinity, we expect m_i to converge to $G_\pi(x)$. The update formula (10.83) can be conveniently used to maintain the improving sequence of cost-to-go estimates. Let $\hat{G}_\pi(x)$ denote the current estimate of $G_\pi(x)$. The update formula based on (10.83) can be expressed as

$$\hat{G}_\pi(x) := \hat{G}_\pi(x) + \frac{1}{i}(l(x_1,u_1) + l(x_2,u_2) + \cdots + l(x_K,u_K) - \hat{G}_\pi(x)), \quad (10.85)$$

in which := means assignment, in the sense used in some programming languages.

It turns out that a single trial can actually yield update values for multiple states [930, 96]. Suppose that a trial is performed from x that results in the sequence $x_1 = x$, x_2, $\ldots$, x_k, $\ldots$, x_K, x_F of visited states. For every state, x_k, in the sequence, a cost-to-go value can be measured by recording the cost that was accumulated from x_k to x_K:

$$c_k(x_k) = \sum_{j=k}^{K} l(x_j, u_j). \quad (10.86)$$

It is much more efficient to make use of (10.85) on every state that is visited along the path.

Temporal differences Rather than waiting until the end of each trial to compute $c_i(x_i)$, it is possible to update each state, x_i, immediately after it is visited and $l(x_i, u_i)$ is received from the simulator. This leads to a well-known method of estimating the cost-to-go called *temporal differences* [929]. It is very similar to the method already given but somewhat more complicated. It will be introduced here because the method frequently appears in reinforcement learning literature, and an extension of it leads to a nice simulation-based method for updating the estimated cost-to-go.

Once again, consider the sequence x_1, $\ldots$, x_K, x_F generated by a trial. Let d_k denote a *temporal difference*, which is defined as

$$d_k = l(x_k, u_k) + \hat{G}_\pi(x_{k+1}) - \hat{G}_\pi(x_k). \quad (10.87)$$

Note that both $l(x_k, u_k) + \hat{G}_\pi(x_{k+1})$ and $\hat{G}_\pi(x_k)$ could each serve as an estimate of $G_\pi(x_k)$. The difference is that the right part of (10.87) utilizes the latest cost obtained from the simulator for the first step and then uses $\hat{G}_\pi(x_{k+1})$ for an estimate of the remaining cost. In this and subsequent expressions, every action, u_k, is chosen using the plan: $u_k = \pi(x_k)$.

Let v_k denote the number of times that x_k has been visited so far, for each $1 \leq k \leq K$, including previous trials and the current visit. The following update algorithm can be used during the trial. When x_2 is reached, the value at x_1 is updated as

$$\hat{G}_\pi(x_1) := \hat{G}_\pi(x_1) + \frac{1}{v_1} d_1. \quad (10.88)$$

When x_3 is reached, the values at x_1 and x_2 are updated as

$$\begin{aligned} \hat{G}_\pi(x_1) &:= \hat{G}_\pi(x_1) + \frac{1}{v_1} d_2, \\ \hat{G}_\pi(x_2) &:= \hat{G}_\pi(x_2) + \frac{1}{v_2} d_2. \end{aligned} \tag{10.89}$$

Now consider what has been done so far at x_1. The temporal differences partly collapse:

$$\begin{aligned} \hat{G}_\pi(x_1) :=& \hat{G}_\pi(x_1) + \frac{1}{v_1} d_1 + \frac{1}{v_1} d_2 \\ =& \hat{G}_\pi(x_1) + \frac{1}{v_1}(l(x_1,u_1) + \hat{G}_\pi(x_2) - \hat{G}_\pi(x_1) + l(x_2,u_2) + \hat{G}_\pi(x_3) - \hat{G}_\pi(x_2)) \\ =& \hat{G}_\pi(x_1) + \frac{1}{v_1}(l(x_1,u_1) + l(x_2,u_2) - \hat{G}_\pi(x_1) + \hat{G}_\pi(x_3)). \end{aligned} \tag{10.90}$$

When x_4 is reached, similar updates are performed. At x_k, the updates are

$$\begin{aligned} \hat{G}_\pi(x_1) :=& \hat{G}_\pi(x_1) + \frac{1}{v_1} d_k, \\ \hat{G}_\pi(x_2) :=& \hat{G}_\pi(x_2) + \frac{1}{v_2} d_k, \\ &\vdots \\ \hat{G}_\pi(x_k) :=& \hat{G}_\pi(x_k) + \frac{1}{v_k} d_k. \end{aligned} \tag{10.91}$$

The updates are performed in this way until $x_F \in X_G$ is reached. Now consider what was actually computed for each x_k. The temporal differences form a telescoping sum that collapses, as shown in (10.90) after two iterations. After all iterations have been completed, the value at x_k has been updated as

$$\begin{aligned} \hat{G}_\pi(x_k) :=& \hat{G}_\pi(x_k) + \frac{1}{v_k} d_k + \frac{1}{v_{k+1}} d_{k+1} + \cdots + \frac{1}{v_K} d_K + \frac{1}{v_F} d_F \\ =& \hat{G}_\pi(x_k) + \frac{1}{v_k}(l(x_1,u_1) + l(x_2,u_2) + \cdots + l(x_K,u_K) - \hat{G}_\pi(x_k) + \hat{G}_\pi(x_F)) \\ =& \hat{G}_\pi(x_k) + \frac{1}{v_k}(l(x_1,u_1) + l(x_2,u_2) + \cdots + l(x_K,u_K) - \hat{G}_\pi(x_k)). \end{aligned} \tag{10.92}$$

The final $\hat{G}_\pi(x_F)$ was deleted because its value is zero, assuming that the termination action is applied by π. The resulting final expression is equivalent to (10.85) if each visited state in the sequence was distinct. This is often not true, which makes the method discussed above differ slightly from the method of (10.85) because the

count, v_k, may change during the trial in the temporal difference scheme. This difference, however, is negligible, and the temporal difference method computes estimates that converge to $\hat{G}_\pi$ [96, 97].

The temporal difference method presented so far can be generalized in a way that often leads to faster convergence in practice. Let $\lambda \in [0,1]$ be a specified parameter. The $TD(\lambda)$ temporal difference method replaces the equations in (10.91) with

$$\begin{aligned}
\hat{G}_\pi(x_1) :=& \hat{G}_\pi(x_1) + \lambda^{k-1}\left(\frac{1}{v_1} d_k\right), \\
\hat{G}_\pi(x_2) :=& \hat{G}_\pi(x_2) + \lambda^{k-2}\left(\frac{1}{v_2} d_k\right), \\
&\vdots \\
\hat{G}_\pi(x_{k-1}) :=& \hat{G}_\pi(x_{k-1}) + \lambda\left(\frac{1}{v_{k-1}} d_k\right), \\
\hat{G}_\pi(x_k) :=& \hat{G}_\pi(x_k) + \frac{1}{v_k} d_k.
\end{aligned} \tag{10.93}$$

This has the effect of discounting costs that are received far away from x_k. The method in (10.91) was the special case of $\lambda = 1$, yielding $TD(1)$.

Another interesting special case is $TD(0)$, which becomes

$$\hat{G}_\pi(x_k) = \hat{G}_\pi(x_k) + \frac{1}{v_k}\Big(l(x_k, u_k) + \hat{G}_\pi(x_{k+1}) - \hat{G}_\pi(x_k)\Big). \tag{10.94}$$

This form appears most often in reinforcement learning literature (although it is applied to the discounted-cost model instead). Experimental evidence indicates that lower values of λ help to improve the convergence rate. Convergence for all values of λ is proved in [97].

One source of intuition about why (10.94) works is that it is a special case of a *stochastic iterative algorithm* or the *Robbins-Monro algorithm* [88, 97, 566]. This is a general statistical estimation technique that is used for solving systems of the form $h(y) = y$ by using a sequence of samples. Each sample represents a measurement of $h(y)$ using Monte Carlo simulation. The general form of this iterative approach is to update y as

$$y := (1 - \rho)y + \rho h(y), \tag{10.95}$$

in which $\rho \in [0,1]$ is a parameter whose choice affects the convergence rate. Intuitively, (10.95) updates y by interpolating between its original value and the most recent sample of $h(y)$. Convergence proofs for this algorithm are not given here; see [97] for details. The typical behavior is that a smaller value of ρ leads to more reliable estimates when there is substantial noise in the simulation process, but this comes at the cost of slowing the convergence rate. The convergence is asymptotic, which requires that all edges (that have nonzero probability) in the plan-based state transition graph should be visited infinitely often.

A general approach to obtaining $\hat{G}_\pi$ can be derived within the stochastic iterative framework by generalizing $TD(0)$:

$$\hat{G}_\pi(x) := (1-\rho)\hat{G}_\pi(x) + \rho\left(l(x,u) + \hat{G}_\pi(x')\right). \tag{10.96}$$

The formulation of $TD(0)$ in (10.94) essentially selects the ρ parameter by the way it was derived, but in (10.96) any $\rho \in (0,1)$ may be used.

It may appear incorrect that the update equation does not take into account the transition probabilities. It turns out that they are taken into account in the simulation process because transitions that are more likely to occur have a stronger effect on (10.96). The same thing occurs when the mean of a nonuniform probability density function is estimated by using samples from the distribution. The values that occur with higher frequency make stronger contributions to the average, which automatically gives them the appropriate weight.

10.4.3 Q-Learning: Computing an Optimal Plan

This section moves from evaluating a plan to computing an optimal plan in the simulation-based framework. The most important idea is the computation of *Q-factors*, $Q^*(x,u)$. This is an extension of the optimal cost-to-go, G^*, that records optimal costs for each possible combination of a state, $x \in X$, and action $u \in U(x)$. The interpretation of $Q^*(x,u)$ is the expected cost received by starting from state x, applying u, and then following the optimal plan from the resulting next state, $x' = f(x,u,\theta)$. If u happens to be the same action as would be selected by the optimal plan, $\pi^*(x)$, then $Q^*(x,u) = G^*(x)$. Thus, the Q-value can be thought of as the cost of making an arbitrary choice in the first stage and then exhibiting optimal decision making afterward.

Value iteration A simulation-based version of value iteration can be constructed from Q-factors. The reason for their use instead of G^* is that a minimization over $U(x)$ will be avoided in the dynamic programming. Avoiding this minimization enables a sample-by-sample approach to estimating the optimal values and ultimately obtaining the optimal plan. The optimal cost-to-go can be obtained from the Q-factors as

$$G^*(x) = \min_{u \in U(x)} \Big\{ Q^*(x,u) \Big\}. \tag{10.97}$$

This enables the dynamic programming recurrence in (10.46) to be expressed as

$$Q^*(x,u) = l(x,u) + \sum_{x' \in X} P(x'|x,u) \min_{u' \in U(x')} \Big\{ Q^*(x',u') \Big\}. \tag{10.98}$$

By applying (10.97) to the right side of (10.98), it can also be expressed using G^* as

$$Q^*(x,u) = l(x,u) + \sum_{x' \in X} P(x'|x,u) G^*(x'). \tag{10.99}$$

If $P(x'|x,u)$ and $l(x,u)$ were known, then (10.98) would lead to an alternative, storage-intensive way to perform value iteration. After convergence occurs, (10.97) can be used to obtain the G^* values. The optimal plan is constructed as

$$\pi^*(x) = \operatorname*{argmin}_{u \in U(x)} \Big\{ Q^*(x,u) \Big\}. \tag{10.100}$$

Since the costs and transition probabilities are unknown, a simulation-based approach is needed. The stochastic iterative algorithm idea can be applied once again. Recall that (10.96) estimated the cost of a plan by using individual samples and required a convergence-rate parameter, ρ. Using the same idea here, a simulation-based version of value iteration can be derived as

$$\hat{Q}^*(x,u) := (1-\rho)\hat{Q}^*(x,u) + \rho \left(l(x,u) + \min_{u' \in U(x')} \Big\{ \hat{Q}^*(x',u') \Big\} \right), \tag{10.101}$$

in which x' is the next state and $l(x,u)$ is the cost obtained from the simulator when u is applied at x. Initially, all Q-factors are set to zero. Sample trajectories that arrive at the goal can be generated using simulation, and (10.101) is applied to the resulting states and costs in each stage. Once again, the update equation may appear to be incorrect because the transition probabilities are not explicitly mentioned, but this is taken into account automatically through the simulation.

In most literature, Q-learning is applied to the discounted cost model. This yields a minor variant of (10.101):

$$\hat{Q}^*(x,u) := (1-\rho)\hat{Q}^*(x,u) + \rho \left(l(x,u) + \alpha \min_{u' \in U(x')} \Big\{ \hat{Q}^*(x',u') \Big\} \right), \tag{10.102}$$

in which the discount factor α appears because the update equation is derived from (10.76).

Policy iteration A simulation-based policy iteration algorithm can be derived using Q-factors. Recall from Section 10.2.2 that methods are needed to: 1) evaluate a given plan, π, and 2) improve the plan by selecting better actions. The plan evaluation previously involved linear equation solving. Now any plan, π, can be evaluated without even knowing $P(x'|x,u)$ by using the methods of Section 10.4.2. Once $\hat{G}_\pi$ is computed reliably from every $x \in X$, further simulation can be used to compute $Q_\pi(x,u)$ for each $x \in X$ and $u \in U$. This can be achieved by defining a version of (10.99) that is constrained to π:

$$Q_\pi(x,u) = l(x,u) + \sum_{x' \in X} P(x'|x,u) G_\pi(x'). \tag{10.103}$$

The transition probabilities do not need to be known. The Q-factors are computed by simulation and averaging. The plan can be improved by setting

$$\pi'(x) = \operatorname*{argmin}_{u \in U(x)} \Big\{ Q^*(x,u) \Big\}, \tag{10.104}$$

which is based on (10.97).

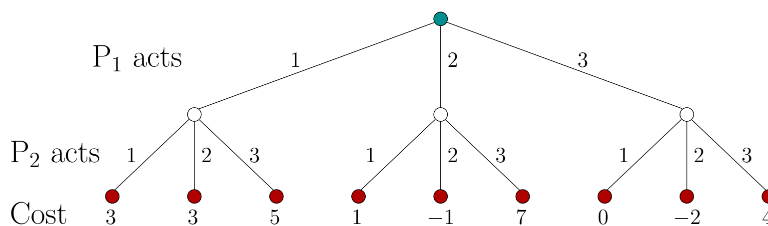


Figure 10.12: A 3×3 matrix game expressed using a game tree.

10.5 Sequential Game Theory

So far in the chapter, the sequential decision-making process has only involved a game against nature. In this section, other decision makers are introduced to the game. The single-stage games and their equilibrium concepts from Sections 9.3 and 9.4 will be extended into a sequence of games. Section 10.5.1 introduces sequential zero-sum games that are represented using game trees, which help visualize the concepts. Section 10.5.2 covers sequential zero-sum games using the state-space representation. Section 10.5.3 briefly covers extensions to other games, including nonzero-sum games and games that involve nature. The formulations in this section will be called *sequential game theory*. Another common name for them is *dynamic game theory* [59]. If there is a continuum of stages, which is briefly considered in Section 13.5, then *differential game theory* is obtained [59, 477, 783, 985].

10.5.1 Game Trees

In most literature, sequential games are formulated in terms of *game trees*. A state-space representation, which is more in alignment with the representations used in this chapter, will be presented in Section 10.5.2. The tree representation is commonly referred to as the *extensive form* of a game (as opposed to the *normal form*, which is the cost matrix representation used in Chapter 9). The representation is helpful for visualizing many issues in game theory. It is perhaps most helpful for visualizing information states; this aspect of game trees will be deferred until Section 11.7, after information spaces have been formally introduced. Here, game trees are presented for cases that are simple to describe without going deeply into information spaces.

Before a sequential game is introduced, consider representing a single-stage game in a tree form. Recall Example 9.14, which is a zero-sum, 3×3 matrix game. It can be represented as a *game tree* as shown in Figure 10.12. At the root, P_1 has three choices. At the next level, P_2 has three choices. Based on the choices by both, one of nine possible leaves will be reached. At this point, a cost is obtained, which is written under the leaf. The entries of the cost matrix, (9.53), appear across the leaves of the tree. Every nonleaf vertex is called a *decision vertex*: One player must select an action.

There are two possible interpretations of the game depicted in Figure 10.12:

1. Before it makes its decision, P_2 knows which action was applied by P_1. This does not correspond to the zero-sum game formulation introduced in Section 9.3 because P_2 seems as powerful as nature. In this case, it is not equivalent to the game in Example 9.14.

2. P_2 does not know the action applied by P_1. This is equivalent to assuming that both P_1 and P_2 make their decisions at the same time, which is consistent with Formulation 9.7. The tree could have alternatively been represented with P_2 acting first.

Now imagine that P_1 and P_2 play a *sequence* of games. A sequential version of the zero-sum game from Section 9.3 will be defined by extending the game tree idea given so far to more levels. This will model the following *sequential game*:

Formulation 10.3 (Zero-Sum Sequential Game in Tree Form)

1. Two players, P_1 and P_2, take turns playing a game. A stage as considered previously is now stretched into two *substages*, in which each player acts individually. It is usually assumed that P_1 always starts, followed by P_2, then P_1 again, and so on. Player alternations continue until the game ends. The model reflects the rules of many popular games such as chess or poker. Let $\mathcal{K} = \{1, \ldots, K\}$ denote the set of stages at which P_1 and P_2 both take a turn.

2. As each player takes a turn, it chooses from a nonempty, finite set of actions. The available set could depend on the decision vertex.

3. At the end of the game, a cost for P_1 is incurred based on the sequence of actions chosen by each player. The cost is interpreted as a reward for P_2.

4. The amount of information that each player has when making its decision must be specified. This is usually expressed by indicating what portions of the action histories are known. For example, if P_1 just acted, does P_2 know its choice? Does it know what action P_1 chose in some previous stage?

The *game tree* can now be described in detail. Figure 10.13 shows a particular example for two stages (hence, $K = 2$ and $\mathcal{K} = \{1, 2\}$). Every vertex corresponds to a point at which a decision needs to be made by one player. Each edge emanating from a vertex represents an action. The root of the tree indicates the beginning of the game, which usually means that P_1 chooses an action. The leaves of the tree represent the end of the game, which are the points at which a cost is received. The cost is usually shown below each leaf. One final concern is to specify the information available to each player, just prior to its decision. Which actions among those previously applied by itself or other players are known?

For the game tree in Figure 10.13, there are two players and two stages. Therefore, there are four levels of decision vertices. The action sets for the players are $U = V = \{L, R\}$, for "left" and "right." Since there are always two actions, a

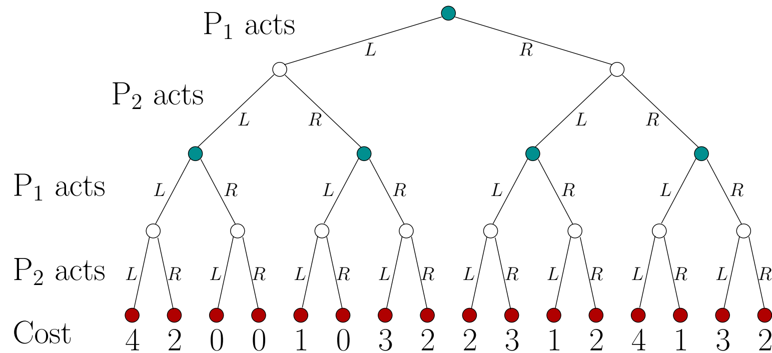


Figure 10.13: A two-player, two-stage game expressed using a game tree.

binary tree is obtained. There are 16 possible outcomes, which correspond to all pairwise combinations of four possible two-stage plans for each player.

For a single-stage game, both deterministic and randomized strategies were defined to obtain saddle points. Recall from Section 9.3.3 that randomized strategies were needed to guarantee the existence of a saddle point. For a sequential game, these are extended to *deterministic* and *randomized plans*, respectively. In Section 10.1.3, a (deterministic) plan was defined as a mapping from the state space to an action space. This definition can be applied here for each player; however, we must determine what is a "state" for the game tree. This depends on the information that each player has available when it plays.

A general framework for representing information in game trees is covered in Section 11.7. Three simple kinds of information will be discussed here. In every case, each player knows its own actions that were applied in previous stages. The differences correspond to knowledge of actions applied by the other player. These define the "state" that is used to make the decisions in a plan.

The three information models considered here are as follows.

Alternating play: The players take turns playing, and all players know all actions that have been previously applied. This is the situation obtained, for example, in a game of chess. To define a plan, let N_1 and N_2 denote the set of all vertices from which P_1 and P_2 must make a decision, respectively. In Figure 10.13, N_1 is the set of dark vertices and N_2 is the set of white vertices. Let $U(n_1)$ and $V(n_2)$ be the action spaces for P_1 and P_2, respectively, which depend on the vertex. A *(deterministic) plan for* P_1 is defined as a function, π_1, on N_1 that yields an action $u \in U(n_1)$ for each $n_1 \in N_1$. Similarly, a *(deterministic) plan for* P_2 is defined as a function, π_2, on N_2 that yields an action $v \in V(n_2)$ for each $n_2 \in N_2$. For the randomized case, let $W(n_1)$ and $Z(n_2)$ denote the sets of all probability distributions over $U(n_1)$ and $V(n_2)$, respectively. A *randomized plan for* P_1 is defined as a function that yields some $w \in W(n_1)$ for each $n_1 \in N_1$. Likewise, a *randomized plan for* P_2 is defined as a function that maps from N_2 into $Z(n_2)$.

Stage-by-stage: Each player knows the actions applied by the other in all

previous stages; however, there is no information about actions chosen by others in the current stage. This effectively means that both players act simultaneously in each stage. In this case, a deterministic or randomized plan for P_1 is defined as in the alternating play case; however, plans for P_2 are defined as functions on N_1, instead of N_2. This is because at the time it makes its decision, P_2 has available precisely the same information as P_1. The action spaces for P_2 must conform to be dependent on elements of N_1, instead of N_2; otherwise, P_2 would not know what actions are available. Therefore, they are defined as $V(n_1)$ for each $n_1 \in N_1$.

Open loop: Each player has no knowledge of the previous actions of the other. They only know how many actions have been applied so far, which indicates the stage of the game. Plans are defined as functions on $\mathcal{K}$, the set of stages, because the particular vertex is not known. Note that an open-loop plan is just a sequence of actions in the deterministic case (as in Section 2.3) and a sequence of probability distributions in the randomized case. Again, the action spaces must conform to the information. Thus, they are $U(k)$ and $V(k)$ for each $k \in \mathcal{K}$.

For a single-stage game, as in Figure 10.12, the stage-by-stage and open-loop models are equivalent.

10.5.1.1 Determining a security plan

The notion of a security strategy from Section 9.3.2 extends in a natural way to sequential games. This yields a *security plan* in which each player performs worst-case analysis by treating the other player as nature under nondeterministic uncertainty. A security plan and its resulting cost can be computed by propagating costs from the leaves up to the root. The computation of the security plan for P_1 for the game in Figure 10.13 is shown in Figure 10.14. The actions that would be chosen by P_2 are determined at all vertices in the second-to-last level of the tree. Since P_2 tries to maximize costs, the recorded costs at each of these vertices is the maximum over the costs of its children. At the next higher level, the actions that would be chosen by P_1 are determined. At each vertex, the minimum cost among its children is recorded. In the next level, P_2 is considered, and so on, until the root is reached. At this point, the lowest cost that P_1 could secure is known. This yields the *upper value*, $\overline{L}^*$, for the sequential game. The security plan is defined by providing the action that selects the lowest cost child vertex, for each $n_1 \in N_1$. If P_2 responds rationally to the security plan of P_1, then the path shown in bold in Figure 10.14d will be followed. The execution of P_1's security plan yields the action sequence (L, L) for P_1 and (R, L) for P_2. The upper value is $\overline{L}^* = 1$.

A security plan for P_2 can be computed similarly; however, the order of the decisions must be swapped. This is not easy to visualize, unless the order of the players is swapped in the tree. If P_2 acts first, then the resulting tree is as shown in Figure 10.15. The costs on the leaves appear in different order; however, for

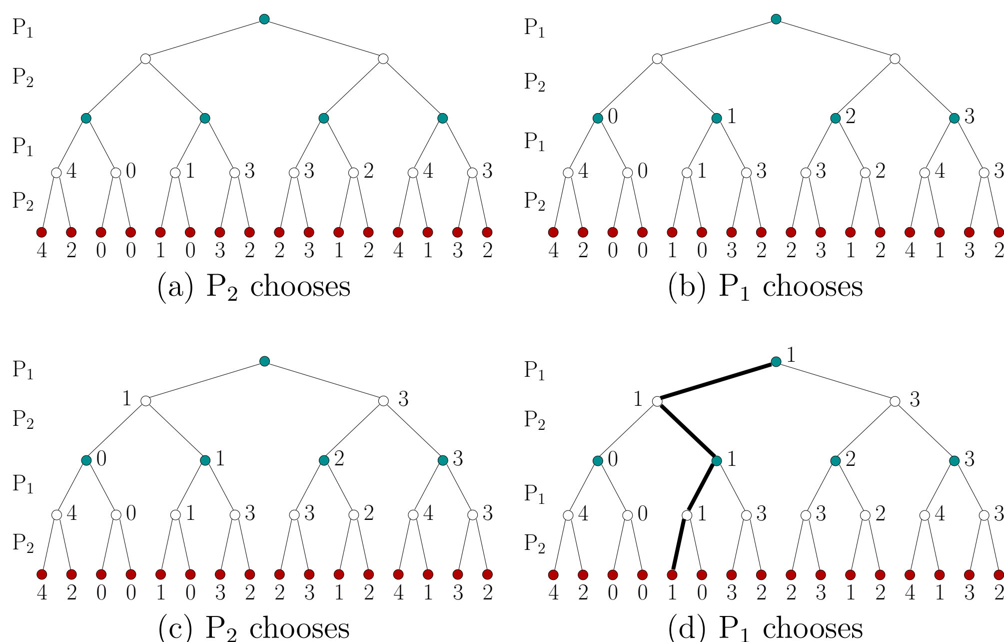


Figure 10.14: The security plan for P_1 is determined by propagating costs upward from the leaves. The choices involved in the security plan are shown in the last picture. An upper value of 1 is obtained for the game.

the same action sequences chosen by P_1 and P_2, the costs obtained at the end of the game are the same as those in Figure 10.14. The resulting *lower value* for the game is found to be $\underline{L}^* = 1$. The resulting security plan is defined by assigning the action to each $n_2 \in N_2$ that maximizes the cost value of its children. If P_1 responds rationally to the security plan of P_2, then the actions executed will be (L, L) for P_1 and (R, L) for P_2. Note that these are the same as those obtained from executing the security plan of P_1, even though they appear different in the trees because the player order was swapped. In many cases, however, different action sequences will be obtained.

As in the case of a single-stage game, $\underline{L}^* = \overline{L}^*$ implies that the game has a deterministic saddle point and the *value* of the sequential game is $L^* = \underline{L}^* = \overline{L}^*$. This particular game has a unique, deterministic saddle point. This yields predictable, identical choices for the players, even though they perform separate, worst-case analyses.

A substantial reduction in the cost of computing the security strategies can be obtained by recognizing when certain parts of the tree do not need to be explored because they cannot yield improved costs. This idea is referred to as *alpha-beta pruning* in AI literature (see [839], pp. 186-187 for references and a brief history). Suppose that the tree is searched in depth-first order to determine the security strategy for P_1. At some decision vertex for P_1, suppose it has been determined that a cost c would be secured if a particular action, u, is applied; however, there

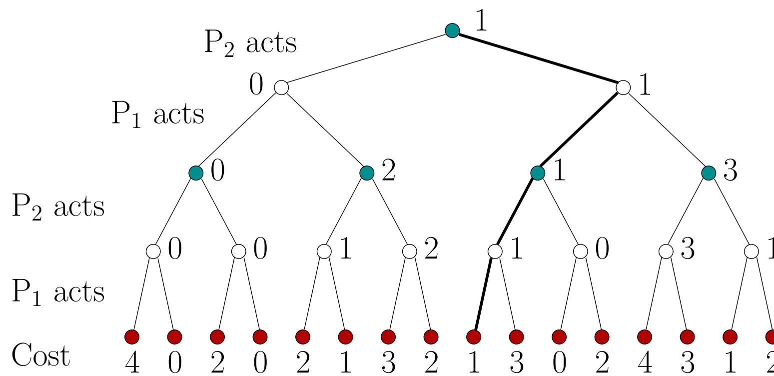


Figure 10.15: The security plan can be found for P_2 by swapping the order of P_1 and P_2 (the order of the costs on the leaves also become reshuffled).

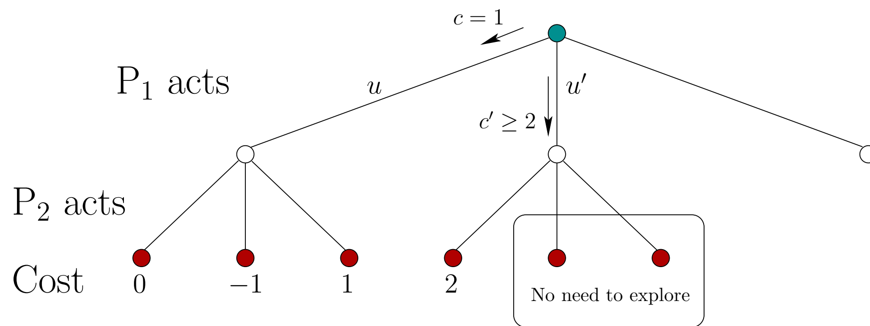


Figure 10.16: If the tree is explored in depth-first order, there are situations in which some children (and hence whole subtrees) do not need to be explored. This is an example that eliminates children of P_2. Another case exists, which eliminates children of P_1.

are still other actions for which it is not known what costs could be secured. Consider determining the cost that could be secured for one of these remaining actions, denoted by u'. This requires computing how P_2 will maximize cost to respond to u'. As soon as P_2 has at least one option for which the cost, c', is greater than c, the other children of P_2 do not need to be explored. Why? This is because P_1 would never choose u' if P_2 could respond in a way that leads to a higher cost than what P_1 can already secure by choosing u. Figure 10.16 shows a simple example. This situation can occur at any level in the tree, and when an action does not need to be considered, an entire subtree is eliminated. In other situations, children of P_1 can be eliminated because P_2 would not make a choice that allows P_1 to improve the cost below a value that P_2 can already secure for itself.

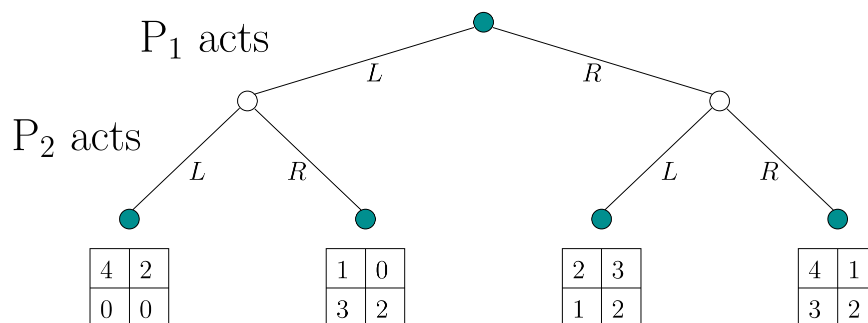


Figure 10.17: Under the stage-by-stage model, the game in Figure 10.13 can instead be represented as a tree in which each player acts once, and then they play a matrix game to determine the cost.

10.5.1.2 Computing a saddle point

The security plan for P_1 constitutes a valid solution to the game under the alternating play model. P_2 has only to choose an optimal response to the plan of P_1 at each stage. Under the stage-by-stage model, the "solution" concept is a saddle point, which occurs when the upper and lower values coincide. The procedure just described could be used to determine the value and corresponding plans; however, what happens when the values do not coincide? In this case, *randomized security plans* should be developed for the players. As in the case of a single-stage game, a *randomized upper value* $\overline{\mathcal{L}}^*$ and a *randomized lower value* $\underline{\mathcal{L}}^*$ are obtained. In the space of randomized plans, it turns out that a saddle point always exists. This implies that the game always has a *randomized value*, $\mathcal{L}^* = \underline{\mathcal{L}}^* = \overline{\mathcal{L}}^*$. This saddle point can be computed from the bottom up, in a manner similar to the method just used to compute security plans.

Return to the example in Figure 10.13. This game actually has a deterministic saddle point, as indicated previously. It still, however, serves as a useful illustration of the method because any deterministic plan can once again be interpreted as a special case of a randomized plan (all of the probability mass is placed on a single action). Consider the bottom four subtrees of Figure 10.13, which are obtained by using only the last two levels of decision vertices. In each case, P_1 and P_2 must act in parallel to end the sequential game. Each subtree can be considered as a matrix game because the costs are immediately obtained after the two players act.

This leads to an alternative way to depict the game in Figure 10.13, which is shown in Figure 10.17. The bottom two layers of decision vertices are replaced by matrix games. Now compute the randomized value for each game and place it at the corresponding leaf vertex, as shown in Figure 10.18. In the example, there are only two layers of decision vertices remaining. This can be represented as the game

$$U \quad \overset{V}{\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array}}\ , \qquad (10.105)$$

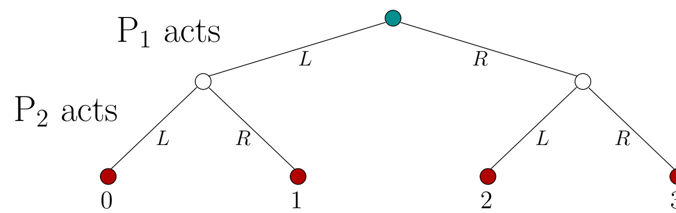


Figure 10.18: Each matrix in Figure 10.17 can be replaced by its randomized value. This clips one level from the original tree. For this particular example, the randomized value is also a deterministic value. Note that these are exactly the costs that appeared in Figure 10.14c. This occurred because each of the matrix games has a deterministic value; if they do not, then the costs will not coincide.

which has a value of 1 and occurs if P_1 applies L and P_2 applies R. Thus, the solution to the original sequential game has been determined by solving matrix games as an alternative to the method applied to obtain the security plans. The benefit of the new method is that if any matrix does not have a deterministic saddle point, its randomized value can instead be computed. A randomized strategy must be played by the players if the corresponding decision vertex is reached during execution.

10.5.1.3 Converting the tree to a single-stage game

Up to this point, solutions have been determined for the alternating-play and the stage-by-stage models. The open-loop model remains. In this case, there is no exchange of information between the players until the game is finished and they receive their costs. Therefore, imagine that players engaged in such a sequential game are equivalently engaged in a large, single-stage game. Recall that a plan under the open-loop model is a function over $\mathcal{K}$. Let Π_1 and Π_2 represent the sets of possible plans for P_1 and P_2, respectively. For the game in Figure 10.13, Π_i is a set of four possible plans for each player, which will be specified in the following order: (L, L), (L, R), (R, L), and (R, R). These can be arranged into a 4×4 matrix game:

$$\Pi_1 \quad \overset{\Pi_2}{\begin{array}{|c|c|c|c|} \hline 4 & 2 & 1 & 0 \\ \hline 0 & 0 & 3 & 2 \\ \hline 2 & 3 & 4 & 1 \\ \hline 1 & 2 & 3 & 2 \\ \hline \end{array}} \; . \tag{10.106}$$

This matrix game does not have a deterministic saddle point. Unfortunately, a four-dimensional linear programming problem must be solved to find the randomized value and equilibrium. This is substantially different than the solution obtained for the other two information models.

The matrix-game form can also be derived for sequential games defined under the stage-by-stage model. In this case, however, the space of plans is even larger.

For the example in Figure 10.13, there are 32 possible plans for each player (there are 5 decision vertices for each player, at which two different actions can be applied; hence, $|\Pi_i| = 2^5$ for $i = 1$ and $i = 2$). This results in a 32×32 matrix game! This game should admit the same saddle point solution that we already determined. The advantage of using the tree representation is that this enormous game was decomposed into many tiny matrix games. By treating the problem stage-by-stage, substantial savings in computation results. This power arises because the dynamic programming principle was implicitly used in the tree-based computation method of decomposing the sequential game into small matrix games. The connection to previous dynamic programming methods will be made clearer in the next section, which considers sequential games that are defined over a state space.

10.5.2 Sequential Games on State Spaces

An apparent problem in the previous section is that the number of vertices grows exponentially in the number of stages. In some games, however, there may be multiple action sequences that lead to the same state. This is true of many popular games, such as chess, checkers, and tic-tac-toe. In this case, it is convenient to define a state space that captures the complete set of unique game configurations. The player actions then transform the state. If there are different action sequences that lead to the same state, then separate vertices are not needed. This converts the game tree into a *game graph* by declaring vertices that represent the same state to be equivalent. The game graph is similar in many ways to the transition graphs discussed in Section 10.1, in the sequential game against nature. The same idea can be applied when there are opposing players.

We will arrive at a sequential game that is played over a state space by collapsing the game tree into a game graph. We will also allow the more general case of costs occurring on any transition edges, as opposed to only the leaves of the original game tree. Only the stage-by-stage model from the game tree is generalized here. Generalizations that use other information models are considered in Section 11.7. In the formulation that follows, P_2 can be can viewed as the replacement for nature in Formulation 10.1. The new formulation is still a generalization of Formulation 9.7, which was a single-stage, zero-sum game. To keep the concepts simpler, all spaces are assumed to be finite. The formulation is as follows.

Formulation 10.4 (Sequential Zero-Sum Game on a State Space)

1. Two players, P_1 and P_2.
2. A finite, nonempty *state space* X.
3. For each state $x \in X$, a finite, nonempty *action space* $U(x)$ for P_1.
4. For each state $x \in X$, a finite, nonempty *action space* $V(x)$ for P_2. To allow an extension of the alternating play model from Section 10.5.1, $V(x, u)$ could

alternatively be defined, to enable the set of actions available to P_2 to depend on the action $u \in U$ of P_1.

5. A *state transition function* f that produces a state, $f(x, u, v)$, for every $x \in X$, $u \in U(x)$, and $v \in V(x)$.

6. A set $\mathcal{K}$ of K *stages*, each denoted by k, which begins at $k = 1$ and ends at $k = K$. Let $F = K + 1$, which is the final stage, after the last action is applied.

7. An *initial state* $x_I \in X$. For some problems, this may not be specified, in which case a solution must be found from all initial states.

8. A stage-additive cost functional L. Let $\tilde{v}_K$ denote the history of P_2's actions up to stage K. The cost functional may be applied to any combination of state and action histories to yield

$$L(\tilde{x}_F, \tilde{u}_K, \tilde{v}_K) = \sum_{k=1}^{K} l(x_k, u_k, v_k) + l_F(x_F). \qquad (10.107)$$

It will be assumed that both players always know the current state. Note that there are no termination actions in the formulation. The game terminates after each player has acted K times. There is also no direct formulation of a goal set. Both termination actions and goal sets can be added to the formulation without difficulty, but this is not considered here. The action sets can easily be extended to allow a dependency on the stage, to yield $U(x, k)$ and $V(x, k)$. The methods presented in this section can be adapted without trouble. This is avoided, however, to make the notation simpler.

Defining a plan for each player Each player must now have its own plan. As in Section 10.1, it seems best to define a plan as a mapping from states to actions, because it may not be clear what actions will be taken by the other decision maker. In Section 10.1, the other decision maker was nature, and here it is a rational opponent. Let π_1 and π_2 denote plans for P_1 and P_2, respectively. Since the number of stages in Formulation 10.4 is fixed, stage-dependent plans of the form $\pi_1 : X \times \mathcal{K} \to U$ and $\pi_2 : X \times \mathcal{K} \to V$ are appropriate (recall that stage-dependent plans were defined in Section 10.1.3). Each produces an action $\pi_1(x, k) \in U(x)$ and $\pi_2(x, k) \in V(x)$, respectively.

Now consider different solution concepts for Formulation 10.4. For P_1, a *deterministic plan* is a function $\pi_1 : X \times \mathcal{K} \to U$, that produces an action $u = \pi(x) \in U(x)$, for each state $x \in X$ and stage $k \in \mathcal{K}$. For P_2 it is instead $\pi_2 : X \times \mathcal{K} \to V$, which produces an action $v = \pi(x) \in V(x)$, for each $x \in X$ and $k \in \mathcal{K}$. Now consider defining a randomized plan. Let $W(x)$ and $Z(x)$ denote the sets of all probability distributions over $U(x)$ and $V(x)$, respectively. A *randomized plan for* P_1 yields some $w \in W(x)$ for each $x \in X$ and $k \in \mathcal{K}$. Likewise, a *randomized plan for* P_2 yields some $z \in Z(x)$ for each $x \in X$ and $k \in \mathcal{K}$.

Saddle points in a sequential game A saddle point will be obtained once again by defining security strategies for each player. Each player treats the other as nature, and if the same worst-case value is obtained, then the result is a saddle point for the game. If the values are different, then a randomized plan is needed to close the gap between the upper and lower values.

Upper and lower values now depend on the initial state, $x_1 \in X$. There was no equivalent for this in Section 10.5.1 because the root of the game tree is the only possible starting point.

If sequences, $\tilde{u}_K$ and $\tilde{v}_K$, of actions are applied from x_1, then the state history, $\tilde{x}_F$, can be derived by repeatedly using the state transition function, f. The *upper value* from x_1 is defined as

$$\overline{L}^*(x_1) = \min_{u_1} \max_{v_1} \min_{u_2} \max_{v_2} \cdots \min_{u_K} \max_{v_K} \Big\{ L(\tilde{x}_F, \tilde{u}_K, \tilde{v}_K) \Big\}, \tag{10.108}$$

which is identical to (10.33) if P_2 is replaced by nature. Also, (10.108) generalizes (9.44) to multiple stages. The *lower value* from x_1, which generalizes (9.46), is

$$\underline{L}^*(x_1) = \max_{v_1} \min_{u_1} \max_{v_2} \min_{u_2} \cdots \max_{v_K} \min_{u_K} \Big\{ L(\tilde{x}_F, \tilde{u}_K, \tilde{v}_K) \Big\}. \tag{10.109}$$

If $\overline{L}^*(x_1) = \underline{L}^*(x_2)$, then a deterministic saddle point exists from x_1. This implies that the order of max and min can be swapped inside of every stage.

Value iteration A value-iteration method can be derived by adapting the derivation that was applied to (10.33) to instead apply to (10.108). This leads to the dynamic programming recurrence

$$\overline{L}^*_k(x_k) = \min_{u_k \in U(x_k)} \Big\{ \max_{v_k \in V(x_k)} \Big\{ l(x_k, u_k, v_k) + \overline{L}^*_{k+1}(x_{k+1}) \Big\} \Big\}, \tag{10.110}$$

which is analogous to (10.39). This can be used to iteratively compute a *security plan for* P_1. The *security plan for* P_2 can be computed using

$$\underline{L}^*_k(x_k) = \max_{v_k \in V(x_k)} \Big\{ \min_{u_k \in U(x_k)} \Big\{ l(x_k, u_k, v_k) + \underline{L}^*_{k+1}(x_{k+1}) \Big\} \Big\}, \tag{10.111}$$

which is the dynamic programming equation derived from (10.109).

Starting from the final stage, F, the upper and lower values are determined directly from the cost function:

$$\overline{L}^*_F(x_F) = \underline{L}^*_F(x_F) = l_F(x_F). \tag{10.112}$$

Now compute $\overline{L}^*_K$ and $\underline{L}^*_K$. From every state, x_K, (10.110) and (10.111) are evaluated to determine whether $\overline{L}^*_K(x_K) = \underline{L}^*_K(x_K)$. If this occurs, then $L^*_L(x_K) = \overline{L}^*_K(x_K) = \underline{L}^*_K(x_K)$ is the *value* of the game from x_K at stage K. If it is determined that from any particular state, $x_K \in X$, the upper and lower values are not equal, then there is no deterministic saddle point from x_K. Furthermore, this will

prevent the existence of deterministic saddle points from other states at earlier stages; these are encountered in later value iterations. Such problems are avoided by allowing randomized plans, but the optimization is more complicated because linear programming is repeatedly involved.

Suppose for now that $\overline{L}^*_K(x_K) = \underline{L}^*_K(x_K)$ for all $x_K \in X$. The value iterations proceed in the usual way from $k = K$ down to $k = 1$. Again, suppose that at every stage, $\overline{L}^*_k(x_k) = \underline{L}^*_k(x_k)$ for all $x_k \in X$. Note that L^*_{k+1} can be written in the place of $\overline{L}^*_{k+1}$ and $\underline{L}^*_{k+1}$ in (10.110) and (10.111) because it is assumed that the upper and lower values coincide. If they do not, then the method fails because randomized plans are needed to obtain a randomized saddle point.

Once the resulting values are computed from each $x_1 \in X_1$, a security plan π^*_1 for P_1 is defined for each $k \in \mathcal{K}$ and $x_k \in X$ as any action u that satisfies the min in (10.110). A security plan π^*_2 is similarly defined for P_2 by applying any action v that satisfies the max in (10.111).

Now suppose that there exists no deterministic saddle point from one or more initial states. To avoid regret, randomized security plans must be developed. These follow by direct extension of the randomized security strategies from Section 9.3.3. The vectors w and z will be used here to denote probability distributions over $U(x)$ and $V(x)$, respectively. The probability vectors are selected from $W(x)$ and $Z(x)$, which correspond to the set of all probability distributions over $U(x)$ and $V(x)$, respectively. For notational convenience, assume $U(x) = \{1, \ldots, m(x)\}$ and $V(x) = \{1, \ldots, n(x)\}$, in which $m(x)$ and $n(x)$ are positive integers.

Recall (9.61) and (9.62), which defined the randomized upper and lower values of a single-stage game. This idea is generalized here to randomized upper and lower value of a *sequential* game. Their definitions are similar to (10.108) and (10.109), except that: 1) the alternating min's and max's are taken over probability distributions on the space of actions, and 2) the expected cost is used.

The dynamic programming principle can be applied to the *randomized upper value* to derive

$$\overline{\mathcal{L}}^*_k(x_k) = \min_{w \in W(x_k)} \left\{ \max_{z \in Z(x_k)} \left\{ \sum_{i=1}^{m(x_k)} \sum_{j=1}^{n(x_k)} \Big(l(x_k, i, j) + \overline{\mathcal{L}}^*_{k+1}(x_{k+1}) \Big) w_i z_j \right\} \right\}, \tag{10.113}$$

in which $x_{k+1} = f(x_k, i, j)$. The *randomized lower value* is similarly obtained as

$$\underline{\mathcal{L}}^*_k(x_k) = \max_{z \in Z(x_k)} \left\{ \min_{w \in W(x_k)} \left\{ \sum_{i=1}^{m(x_k)} \sum_{j=1}^{n(x_k)} \Big(l(x_k, i, j) + \underline{\mathcal{L}}^*_{k+1}(x_{k+1}) \Big) w_i z_j \right\} \right\}. \tag{10.114}$$

In many games, the cost term may depend only on the state: $l(x, u, v) = l(x)$ for all $x \in X$, $u \in U(x)$ and $v \in V(x)$. In this case, (10.113) and (10.114) simplify to

$$\overline{\mathcal{L}}^*_k(x_k) = \min_{w \in W(x_k)} \left\{ \max_{z \in Z(x_k)} \left\{ l(x_k) + \sum_{i=1}^{m(x_k)} \sum_{j=1}^{n(x_k)} \overline{\mathcal{L}}^*_{k+1}(x_{k+1}) w_i z_j \right\} \right\} \tag{10.115}$$

and

$$\underline{\mathcal{L}}^*_k(x_k) = \max_{z \in Z(x_k)} \left\{ \min_{w \in W(x_k)} \left\{ l(x_k) + \sum_{i=1}^{m(x_k)} \sum_{j=1}^{n(x_k)} \underline{\mathcal{L}}^*_{k+1}(x_{k+1}) w_i z_j \right\} \right\}, \qquad (10.116)$$

which is similar to the simplification obtained in (10.46), in which θ_k was assumed not to appear in the cost term. The summations are essentially generalizations of (9.57) to the multiple-stage case. If desired, these could even be written as matrix multiplications, as was done in Section 9.3.3.

Value iteration can be performed over the equations above to obtain the randomized values of the sequential game. Since the upper and lower values are always the same, there is no need to check for discrepancies between the two. In practice, it is best in every evaluation of (10.113) and (10.114) (or their simpler forms) to first check whether a deterministic saddle exists from x_k. Whenever one does not exist, the linear programming problem formulated in Section 9.3.3 must be solved to determine the value and the best randomized plan for each player. This can be avoided if a deterministic saddle exists from the current state and stage.

10.5.3 Other Sequential Games

Most of the ideas presented so far in Section 10.5 extend naturally to other sequential game problems. This subsection briefly mentions some of these possible extensions.

Nash equilibria in sequential games Formulations 10.3 and 10.4 can be extended to sequential nonzero-sum games. In the case of game trees, a *cost vector*, with one element for each player, is written at each of the leaves. Under the stage-by-stage model, deterministic and randomized Nash equilibria can be computed using the bottom-up technique that was presented in Section 10.5.1. This will result in the computation of a single Nash equilibrium. To represent all Nash equilibria is considerably more challenging. As usual, the game tree is decomposed into many matrix games; however, in each case, all Nash equilibria must be found and recorded along with their corresponding costs. Instead of propagating a single cost up the tree, a set of cost vectors, along with the actions associated with each cost vector, must be propagated up the tree to the root. As in the case of a single-stage game, nonadmissible Nash equilibria can be removed from consideration. Thus, from every matrix game encountered in the computation, only the admissible Nash equilibria and their costs should be propagated upward.

Formulation 10.4 can be extended by introducing the cost functions L_1 and L_2 for P_1 and P_2, respectively. The value-iteration approach can be extended in a way similar to the extension of the game tree method. Multiple value vectors and their corresponding actions must be maintained for each combination of state and stage. These correspond to the admissible Nash equilibria.

The nonuniqueness of Nash equilibria causes the greatest difficulty in the sequential game setting. There are typically many more equilibria in a sequential

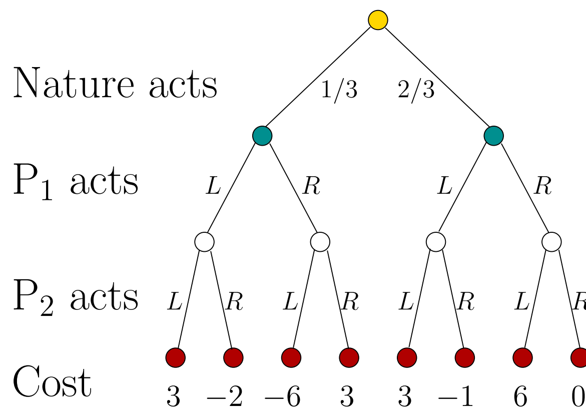


Figure 10.19: This is a single-stage, zero-sum game that involves nature. It is assumed that all players act at the same time.

game than in a single-stage game. Therefore, the concept is not very useful in the design of a planning approach. It may be more useful, for example, in modeling the possible outcomes of a complicated economic system. A thorough treatment of the subject appears in [59].

Introducing nature A nature player can easily be introduced into a game. Suppose, for example, that nature is introduced into a zero-sum game. In this case, there are three players: P_1, P_2, and nature. Figure 10.19 shows a game tree for a single-stage, zero-sum game that involves nature. It is assumed that all three players act at the same time, which fits the stage-by-stage model. Many other information models are possible. Suppose that probabilistic uncertainty is used to model nature, and it is known that nature chooses the left branch with probability 1/3 and the right branch with probability 2/3. Depending on the branch chosen by nature, it appears that P_1 and P_2 will play a specific 2×2 matrix game. With probability 1/3, the cost matrix will be

$$U \overset{V}{\begin{array}{|c|c|}\hline 3 & -2 \\ \hline -6 & 3 \\ \hline \end{array}} , \tag{10.117}$$

and with probability 2/3 it will be

$$U \overset{V}{\begin{array}{|c|c|}\hline 3 & -1 \\ \hline 6 & 0 \\ \hline \end{array}} . \tag{10.118}$$

Unfortunately, P_1 and P_2 do not know which matrix game they are actually playing. The regret can be eliminated in the expected sense, if the game is played over many independent trials. Let A_1 and A_2 denote (10.117) and (10.118), respectively. Define a new cost matrix as $A = (1/3)A_1 + (2/3)A_2$ (a scalar multiplied by

a matrix scales every value of the matrix). The resulting matrix is

$$U \quad \overset{V}{\begin{array}{|c|c|}\hline 3 & 0 \\ \hline 2 & 1 \\ \hline \end{array}} \; . \tag{10.119}$$

This matrix game has a deterministic saddle point in which P_1 chooses L (row 2) and P_2 chooses R (column 1), which yields a cost of 2. This means that they can play a deterministic strategy to obtain an expected cost of 2, if the game play is averaged over many independent trials. If this matrix did not admit a deterministic saddle point, then a randomized strategy would be needed. It is interesting to note that randomization is not needed for this example, even though P_1 and P_2 each play against both nature and an intelligent adversary.

Several other variations are possible. If nature is modeled nondeterministically, then a matrix of worst-case regrets can be formed to determine whether it is possible to eliminate regret. A sequential version of games such as the one in Figure 10.19 can be considered. In each stage, there are three substages in which nature, P_1, and P_2 all act. The bottom-up approach from Section 10.5.1 can be applied to decompose the tree into many single-stage games. Their costs can be propagated upward to the root in the same way to obtain an equilibrium solution.

Formulation 10.4 can be easily extended to include nature in games over state spaces. For each x, a nature action set is defined as $\Theta(x)$. The state transition equation is defined as

$$x_{k+1} = f(x_k, u_k, v_k, \theta_k), \tag{10.120}$$

which means that the next state depends on all three player actions, in addition to the current state. The value-iteration method can be extended to solve problems of this type by properly considering the effect of nature in the dynamic programming equations. In the probabilistic case, for example, an expectation over nature is needed in every iteration. The resulting sequential game is often referred to as a *Markov game* [774].

Introducing more players Involving more players poses no great difficulty, other than complicating the notation. For example, suppose that a set of n players, P_1, P_2, $\ldots$, P_n, takes turns playing a game. Consider using a game tree representation. A stage is now stretched into n *substages*, in which each player acts individually. Suppose that P_1 always starts, followed by P_2, and so on, until P_n. After P_n acts, then the next stage is started, and P_1 acts. The circular sequence of player alternations continues until the game ends. Again, many different information models are possible. For example, in the stage-by-stage model, each player does not know the action chosen by the other $n-1$ players in the current stage. The bottom-up computation method can be used to compute Nash equilibria; however, the problems with nonuniqueness must once again be confronted.

A state-space formulation that generalizes Formulation 10.4 can be made by introducing action sets $U^i(x)$ for each player P_i and state $x \in X$. Let u^i_k denote

the action chosen by P_i at stage k. The state transition becomes

$$x_{k+1} = f(x_k, u_k^1, u_k^2, \ldots, u_k^n). \tag{10.121}$$

There is also a cost function, L_i, for each P_i. Value iteration, adapted to maintain multiple equilibria and cost vectors can be used to compute Nash equilibria.

10.6 Continuous State Spaces

Virtually all of the concepts covered in this chapter extend to continuous state spaces. This enables them to at least theoretically be applied to configuration spaces. Thus, a motion planning problem that involves uncertainty or noncooperating robots can be modeled using the concepts of this chapter. Such problems also inherit the feedback concepts from Chapter 8. This section covers feedback motion planning problems that incorporate uncertainty due to nature. In particular contexts, it may be possible to extend some of the methods of Sections 8.4 and 8.5. Solution feedback plans must ensure that the goal is reached in spite of nature's efforts. Among the methods in Chapter 8, the easiest to generalize is value iteration with interpolation, which was covered in Section 8.5.2. Therefore, it is the main focus of the current section. For games in continuous state spaces, see Section 13.5.

10.6.1 Extending the value-iteration method

The presentation follows in the same way as in Section 8.5.2, by beginning with the discrete problem and making various components continuous. Begin with Formulation 10.1 and let X be a bounded, open subset of $\mathbb{R}^n$. Assume that $U(x)$ and $\Theta(x, u)$ are finite. The value-iteration methods of Section 10.2.1 can be directly applied by using the interpolation concepts from Section 8.5.2 to compute the cost-to-go values over X. In the nondeterministic case, the recurrence is (10.39), in which G^*_{k+1} is represented on a finite sample set $S \subset X$ and is evaluated on all other points in $R(S)$ by interpolation (recall from Section 8.5.2 that $R(S)$ is the interpolation region of S). In the probabilistic case, (10.45) or (10.46) may once again be used, but G^*_{k+1} is evaluated by interpolation.

If $U(x)$ is continuous, then it can be sampled to evaluate the min in each recurrence, as suggested in Section 8.5.2. Now suppose $\Theta(x, u)$ is continuous. In the nondeterministic case, $\Theta(x, u)$ can be sampled to evaluate the max in (10.39) or it may be possible to employ a general optimization technique directly over $\Theta(x, u)$. In the probabilistic case, the expectation must be taken over a continuous probability space. A probability density function, $p(\theta|x, u)$, characterizes nature's action. A probabilistic state transition density function can be derived from this as $p(x_{k+1}|x_k, u_k)$. Using these densities, the continuous versions of (10.45) and

(10.46) become

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \left\{ \int_{\Theta(x_k,u_k)} \Big(l(x_k,u_k,\theta_k) + G^*_{k+1}(f(x_k,u_k,\theta_k)) \Big) p(\theta_k|x_k,u_k) d\theta_k \right\} \tag{10.122}$$

and

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \left\{ l(x_k,u_k) + \int_X G^*_{k+1}(x_{k+1}) p(x_{k+1}|x_k,u_k) dx_{k+1} \right\}, \tag{10.123}$$

respectively. Sampling can be used to evaluate the integrals. One straightforward method is to approximate $p(\theta|x,u)$ by a discrete distribution. For example, in one dimension, this can be achieved by partitioning $\Theta(x,u)$ into intervals, in which each interval is declared to be a discrete nature action. The probability associated with the discrete nature action is just the integral of $p(\theta|x,u)$ over the associated interval.

Section 8.5.2 concluded by describing Dijkstra-like algorithms for continuous spaces. These were derived mainly by using backprojections, (8.66), to conclude that some samples cannot change their values because they are too far from the active set. The same principle can be applied in the current setting; however, the weak backprojection, (10.20), must be used instead. Using the weak backprojection, the usual value iterations can be applied while removing all samples that are not in the active set. For many problems, however, the size of the active set may quickly become unmanageable because the weak backprojection often causes much faster propagation than the original backprojection. Continuous-state generalizations of the Dijkstra-like algorithms in Section 10.2.3 can be made; however, this requires the additional condition that in every iteration, it must be possible to extend D by forcing the next state to lie in $R(D)$, in spite of nature.

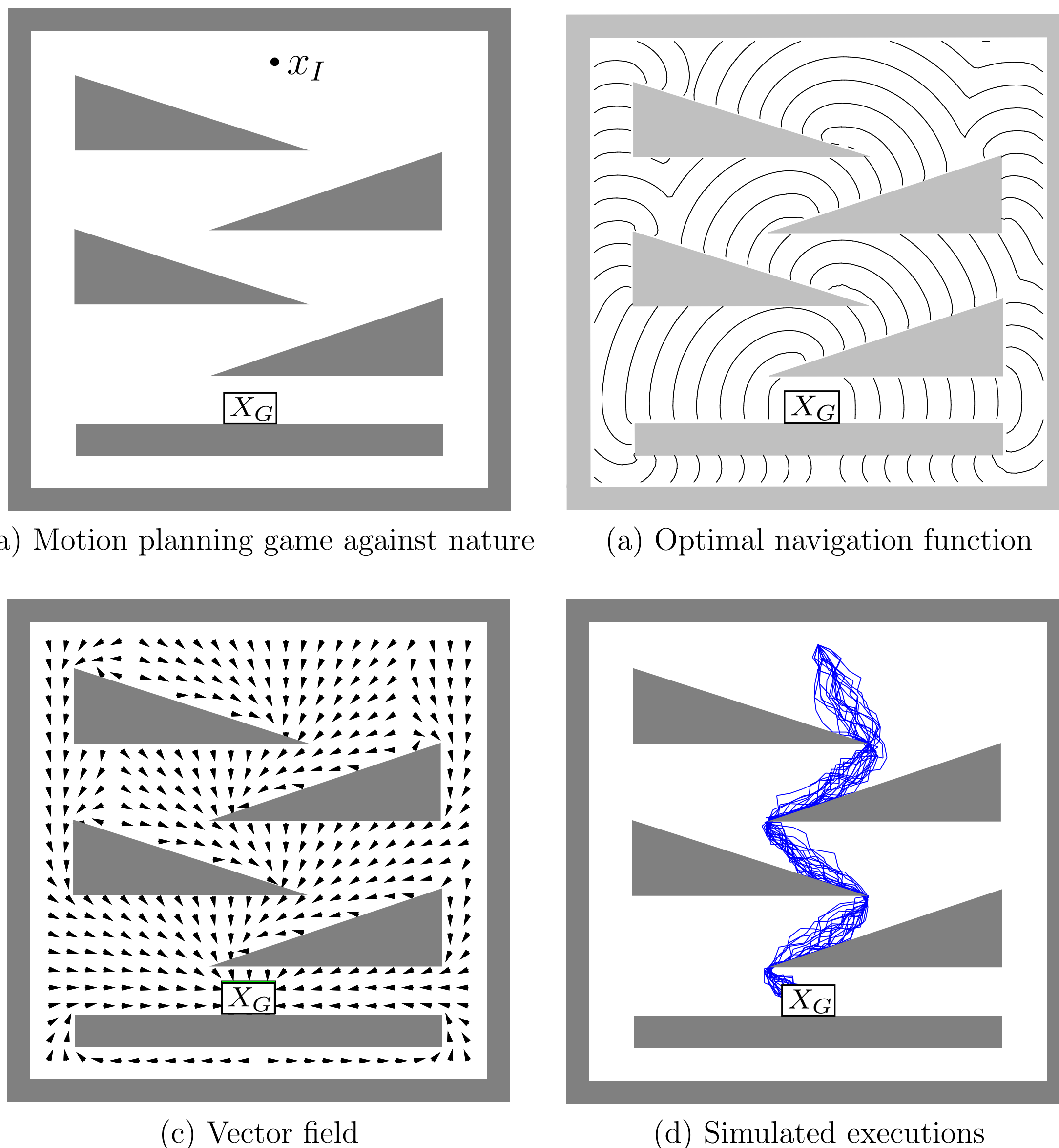
10.6.2 Motion planning with nature

Recall from Section 8.5.2 that value iteration with interpolation can be applied to motion planning problems that are approximated in discrete time. Nature can even be introduced into the discrete-time approximation. For example, (8.62) can be replaced by

$$x(t + \Delta t) = x(t) + \Delta t \; (u + \theta), \tag{10.124}$$

in which θ is chosen from a bounded set, $\Theta(x,u)$. Using (10.124), value iterations can be performed as described so far. An example of a 2D motion planning problem under this model using probabilistic uncertainty is shown in Figure 10.20. It is interesting that when the plan is executed from a fixed initial state, a different trajectory is obtained each time. The average cost over multiple executions, however, is close to the expected optimum.

Interesting hybrid system examples can be made in which nature is only allowed to interfere with the mode. Recall Formulation 7.3 from Section 7.3. Nature can be added to yield the following formulation.



(a) Motion planning game against nature (a) Optimal navigation function

(c) Vector field (d) Simulated executions

Figure 10.20: (a) A 2D planning problem is shown in which nature is probabilistic (uniform density over an interval of angles) and can interfere with the direction of motion. Contact with obstacles is actually allowed in this problem. (b) Level sets of the computed, optimal cost-to-go (navigation) function. (c) The vector field derived from the navigation function. (d) Several dozen execution trials are superimposed [605].

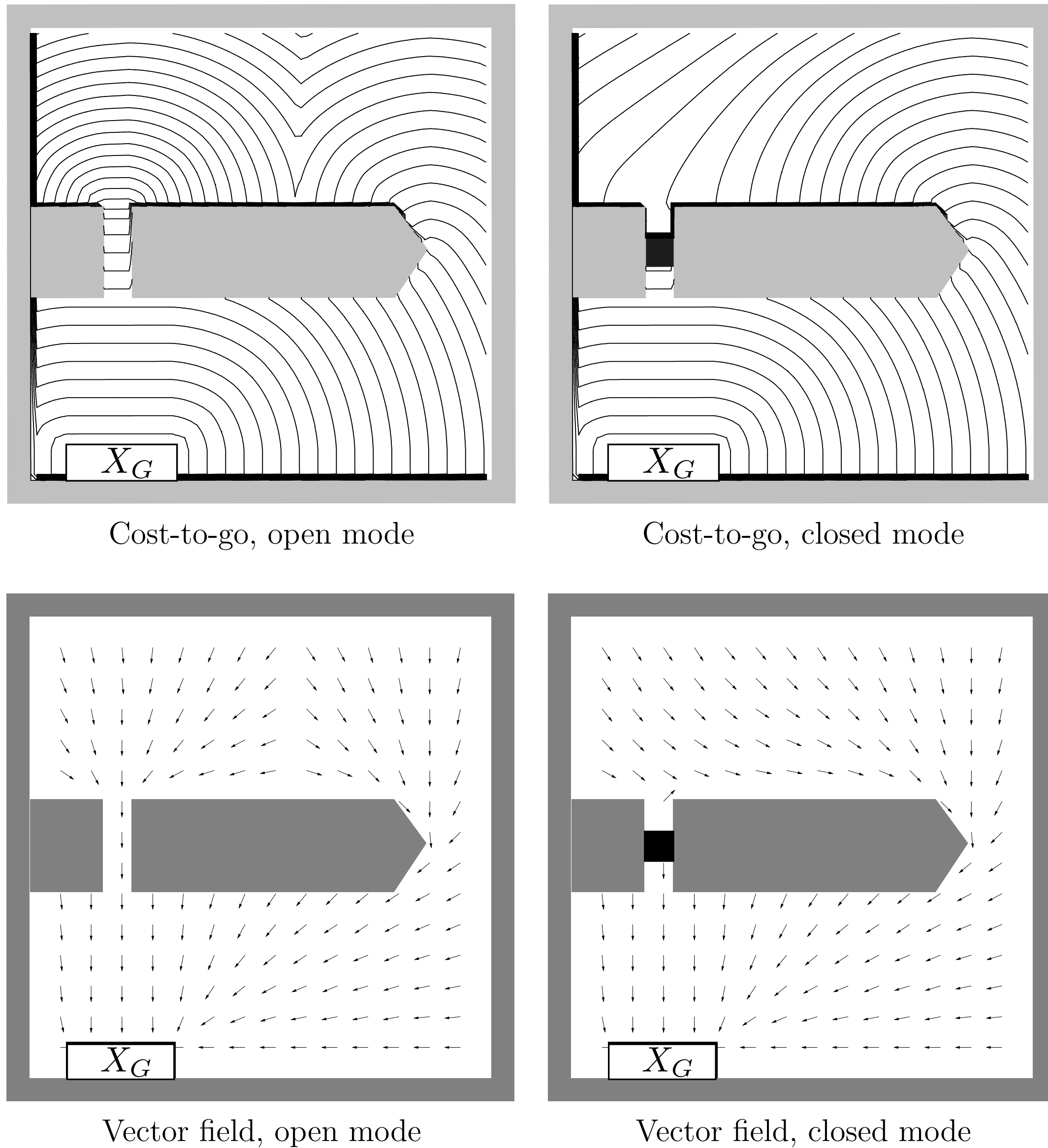


Figure 10.21: Level sets of the optimal navigation function and resulting vector field are shown for a stochastic, hybrid motion planning problem. There are two modes, which correspond to whether a door is closed. The goal is to reach the rectangle at the bottom left [613]

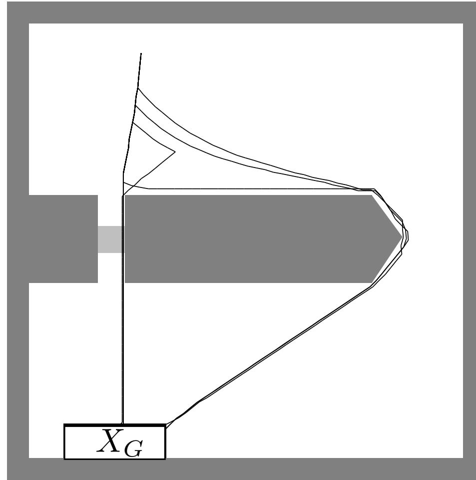


Figure 10.22: Several executions from the same initial state are shown. A different trajectory results each time because of the different times when the door is open or closed.

Formulation 10.5 (Hybrid System Motion Planning with Nature)

1. Assume all of the definitions from Formulation 7.3, except for the transition functions, f_m and f. The state is represented as $x = (q, m)$.

2. A finite *nature action space* $\Theta(x, u)$ for each $x \in X$ and $u \in U(x)$.

3. A *mode transition function* f_m that produces a mode $f_m(x, u, \theta)$ for every $x \in X$, $u \in U(x)$, and $\theta \in \Theta(x, u)$.

4. A *state transition function* f that is derived from f_m by changing the mode and holding the configuration fixed. Thus, $f((q, m), u, \theta) = (q, f_m(q, m, \theta))$ (the only difference with respect to Formulation 7.3 is that θ has been included).

5. An unbounded *time interval* $T = [0, \infty)$.

6. A continuous-time cost-functional,

$$L(\tilde{x}_{t_F}, \tilde{u}_{t_F}) = \int_0^{t_F} l(x(t), u(t))dt + l_F(x(t_F)). \tag{10.125}$$

Value iteration proceeds in the same way for such hybrid problems. Interpolation only needs to be performed over the configuration space. Along the mode "axis" no interpolation is needed because the mode set is already finite. The resulting computation time grows linearly in the number of modes. A 2D motion planning example for a point robot, taken from [613], is shown in Figures 10.21 and 10.22. In this case, the environment contains a door that is modeled as a stationary Markov process. The configuration space is sampled using a 40×40 grid. There are two

modes: door open or door closed. Thus, the configuration space has two layers, one for each mode. The robot wishes to minimize the expected time to reach the goal. The navigation function for each layer cannot be computed independently because each takes into account the transition probabilities for the mode. For example, if the door is almost always open, then its plan would be different from one in which the door is almost always closed. If the door is almost always open, then the robot should go toward the door, even if it is currently closed, because it is highly likely that it will open soon. Numerous variations can be made on this example. More modes could be added, and other interpretations are possible, such as hazardous regions and shelters (the mode might be imagined as rain occurring and the robot must run for shelter) or requests to deliver objects [613, 870, 871].

Further Reading

Since this chapter considers sequential versions of single-stage decision problems, the suggested reading at the end of Chapter 9 is also relevant here. The probabilistic formulation in Section 10.1 is a basic problem of stochastic control theory [95, 564]. The framework is also popular in artificial intelligence [79, 267, 471, 839]. For an early, influential work on stochastic control, see [109], in which the notion of sequential games against nature is developed. The forward projection and backprojection topics are not as common in control theory and are instead inspired from [281, 313, 659]. The nondeterministic formulation is obtained by eliminating probabilities from the formulation; worst-case analysis also appears extensively in control theory [57, 58, 301]. A case for using randomized strategies in robotics is made in [314].

Section 10.2 is based on classical dynamic programming work, but with emphasis on the *stochastic shortest-path problem*. For more reading on value and policy iteration in this context, see [95]. Section 10.2.3 is based on extending Dijkstra's algorithm. For convergence issues due to approximations of continuous problems, see [92, 567, 720]. For complexity results for games against nature, see [764, 767].

Section 10.3 was inspired by coverage in [95]. For further reading on reinforcement learning, the subject of Section 10.4, see [19, 74, 97, 895].

Section 10.5 was based on material in [59], but with an emphasis on unifying concepts from previous sections. Also contained in [59] are sequential game formulations on continuous spaces and even in continuous time. In continuous time, these are called *differential games*, and they are introduced in Section 13.5. Dynamic programming principles extend nicely into game theory. Furthermore, they extend to Pareto optimality [242].

The main purpose of Section 10.6 is to return to motion planning by considering continuous state spaces. Few works exist on combining stochastic optimal control with motion planning. The presented material is based mainly on [599, 605, 613, 867, 868].

Exercises

1. Show that $SB(S, u)$ cannot be expressed as the union of all $SB(x, u)$ for $x \in S$.

2. Show that for any $S \subset X$ and any state transition equation, $x' = f(x, u, \theta)$, it

follows that $SB(S) \subseteq WB(S)$.

3. Generalize the strong and weak backprojections of Section 10.1.2 to work for multiple stages.

4. Assume that nondeterministic uncertainty is used, and there is no limit on the number of stages. Determine an expression for the forward projection at any stage $k > 1$, given that π is applied.

5. Give an algorithm for computing nondeterministic forward projections that uses matrices with binary entries. What is the asymptotic running time and space for your algorithm?

6. Develop a variant of the algorithm in Figure 10.6 that is based on *possibly* achieving the goal, as opposed to *guaranteeing* that it is achieved.

7. Develop a forward version of value iteration for nondeterministic uncertainty, by paralleling the derivation in Section 10.2.1.

8. Do the same as in Exercise 7, but for probabilistic uncertainty.

9. Give an algorithm that computes probabilistic forward projections directly from the plan-based state transition graph, $\mathcal{G}_\pi$.

10. Augment the nondeterministic value-iteration method of Section 10.2.1 to detect and handle states from which the goal is *possibly* reachable but not *guaranteed* reachable.

11. Derive a generalization of (10.39) for the case of stage-dependent state-transition equations, $x_{k+1} = f(x_k, u_k, \theta_k, k)$, and cost terms, $l(x_k, u_k, \theta_k, k)$, under nondeterministic uncertainty.

12. Do the same as in Exercise 11, but for probabilistic uncertainty.

13. Extend the policy-iteration method of Figure 10.4 to work for the more general case of nature-dependent cost terms, $l(x_k, u_k, \theta_k)$.

14. Derive a policy-iteration method that is the nondeterministic analog to the method in Figure 10.4. Assume that the cost terms do not depend on nature.

15. Can policy iteration be applied to solve problems under Formulation 2.3, which involve no uncertainties? Explain what happens in this case.

16. Show that the probabilistic infinite-horizon problem under the discounted-cost model is equivalent in terms of cost-to-go to a particular stochastic shortest-path problem (under Formulation 10.1). [Hint: See page 378 of [95].]

17. Derive a value-iteration method for the infinite-horizon problem with the discounted-cost model and nondeterministic uncertainty. This method should compute the cost-to-go given in (10.71).

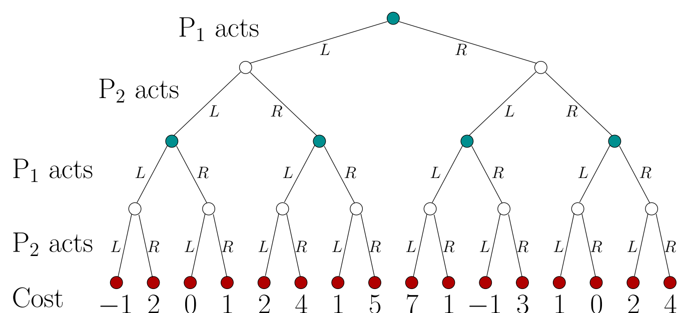


Figure 10.23: A two-player, two-stage game expressed using a game tree.

18. Figure 10.23 shows a two-stage, zero-sum game expressed as a game tree. Compute the randomized value of this sequential game and give the corresponding randomized security plans for each player.

19. Generalize alpha-beta pruning beyond game trees so that it works for sequential games defined on a state space, starting from a fixed initial state.

20. Derive (10.110) and (10.111).

21. Extend Formulation 2.4 to allow nondeterministic uncertainty. This can be accomplished by specifying sets of possible effects of operators.

22. Extend Formulation 2.4 to allow probabilistic uncertainty. For this case, assign probabilities to the possible operator effects.

Implementations

23. Implement probabilistic backward value iteration and study the convergence issue depicted in Figure 10.3. How does this affect performance in problems for which there are many cycles in the state transition graph? How does performance depend on particular costs and transition probabilities?

24. Implement the nondeterministic version of Dijkstra's algorithm and test it on a few examples.

25. Implement and test the probabilistic version of Dijkstra's algorithm. Make sure that the condition $G_\pi(x_{k+1}) < G_\pi(x_k)$ from 10.2.3 is satisfied. Study the performance of the algorithm on problems for which the condition is almost violated.

26. Experiment with the simulation-based version of value iteration, which is given by (10.101). For some simple examples, characterize how the performance depends on the choice of ρ.

27. Implement a recursive algorithm that uses dynamic programming to determine the upper and lower values for a sequential game expressed using a game tree under the stage-by-stage model.