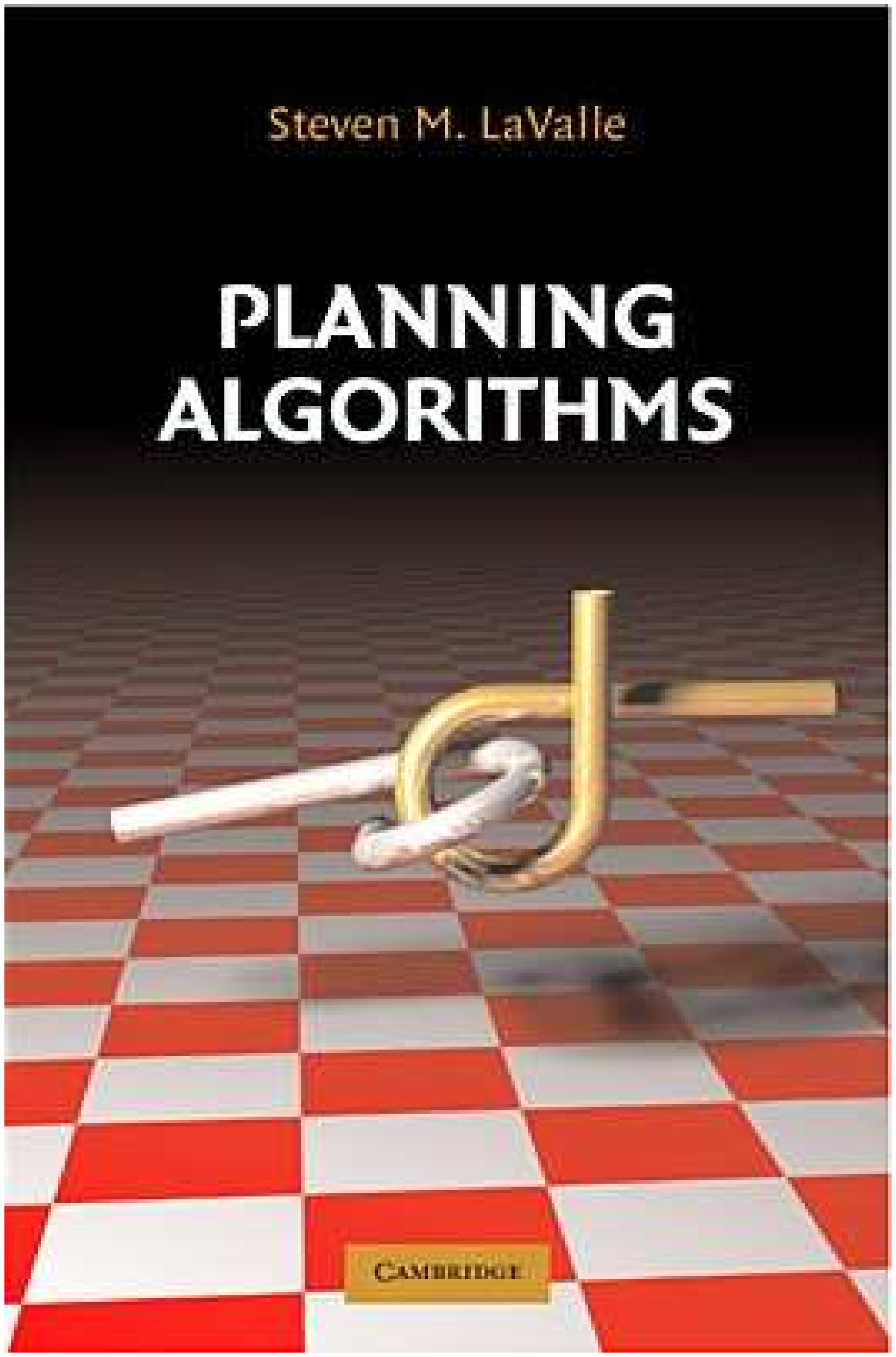


Part III

Decision-Theoretic Planning

Steven M. LaValle

University of Illinois

Published by Cambridge University Press

Overview of Part III: Decision-Theoretic Planning

Planning Under Uncertainty

As in Part II, it also seems appropriate to give two names to Part III. It is officially called *decision-theoretic planning*, but it can also be considered as *planning under uncertainty*. All of the concepts in Parts I and II avoided models of uncertainties. Chapter 8 considered plans that can overcome some uncertainties, but there was no explicit modeling of uncertainty.

In this part, uncertainties generally interfere with two aspects of planning:

1. **Predictability:** Due to uncertainties, it is not known what will happen in the future when certain actions are applied. This means that future states are not necessarily predictable.

2. **Sensing:** Due to uncertainties, the current state is not necessarily known. Information regarding the state is obtained from initial conditions, sensors, and the memory of previously applied actions.

These two kinds of uncertainty are independent in many ways. Each has a different effect on the planning problem.

Making a single decision Chapter 9 provides an introduction to Part III by presenting ways to represent uncertainty in the process of making a single decision. The view taken in this chapter is that uncertainty can be modeled as interference from another decision maker. A special decision maker called *nature* will be introduced. The task is to make good decisions, in spite of actions applied by nature. Either *worst-case* or *probabilistic* models can be used to characterize nature's decision-making process. Some planning problems might involve multiple rational decision makers. This leads to game theory, which arises from the uncertainty about how other players will behave when they have conflicting goals. All of the concepts in Chapter 9 involve making a single decision; therefore, a state space is generally not necessary because there would only be one application of the state transition equation. One purpose of the chapter is to introduce and carefully evaluate the assumptions that are typically made in different forms of decision theory. This forms the basis of more complicated problems that follow, especially sequential decision making and control theory.

Uncertainty in predictability Chapter 10 takes the concepts from Chapter 9 and iterates them over multiple stages. This brings in the notions of states and state transitions, and can be considered as a blending of discrete planning concepts from Chapter 2 with the uncertainty concepts of Chapter 9. Some coverage of continuous state spaces and continuous time is also given, which extends ideas

from Part II. The state transition equation is generally extended to allow future states to depend on unknown actions taken by nature. In a game-theoretic setting, the state transitions may even depend on the actions of more than two decision makers.

For all of the models in Chapter 10, only uncertainty in predictability exists; the current state is always known. A plan is defined as a function that indicates the appropriate action to take from any current state. Plans are not formulated as a sequence of actions because future states are unpredictable, and responses to the future states may be required at the time they are achieved. Thus, for a fixed plan, the execution may be different each time: Different actions are applied and different states are reached. Plans are generally evaluated using worst-case, expected-case, or game-equilibrium analysis.

Uncertainty in sensing: The information space Chapter 11 introduces perhaps the most important concept of this book: the *information space*. If there is uncertainty in sensing the current state, then the planning problem naturally lives in an information space. An analogy can be made to the configuration space and motion planning. Before efforts to unify motion planning by using configuration space concepts [158, 176, 245], most algorithms were developed on a case-by-case basis. For example, robot manipulators and mobile robots have very different characteristics when defined in the world. However, once viewed in the configuration space, it is easier to consider general algorithms, such as those from Chapters 5 and 6.

A similar kind of unification should be possible for planning problems that involve sensing uncertainties (i.e., are unable to determine the current state). Presently, the methods in the literature are developed mainly around individual models and problems, as basic motion planning once was. Therefore, it is difficult to provide a perspective as unified as the techniques in Part II. Nevertheless, the concepts from Chapter 11 are used to provide a unified introduction to many planning problems that involve sensing uncertainties in Chapter 12. As in the case of the configuration space, some effort is required to learn the information space concepts; however, it will pay great dividends if the investment is made.

Chapter 12 presents several different problems and solutions for planning under sensing uncertainty. The problems include exploring new environments with robots, playing a pursuit-evasion game with cameras, and manipulating objects with little or no sensing. The chapter provides many interesting applications of information space concepts, but it should also leave you with the feeling that much more remains to be done. Planning in information spaces remains a challenging research problem throughout much of robotics, control theory, and artificial intelligence.

Chapter 9

Basic Decision Theory

This chapter serves as a building block for modeling and solving planning problems that involve more than one decision maker. The focus is on making a single decision in the presence of other decision makers that may interfere with the outcome. The planning problems in Chapters 10 to 12 will be viewed as a sequence of decision-making problems. The ideas presented in this chapter can be viewed as making a one-stage plan. With respect to Chapter 2, the present chapter reduces the number of stages down to one and then introduces more sophisticated ways to model a single stage. Upon returning to multiple stages in Chapter 10, it will quickly be seen that many algorithms from Chapter 2 extend nicely to incorporate the decision-theoretic concepts of this chapter.

Since there is no information to carry across stages, there will be no need for a state space. Instead of designing a *plan* for a *robot*, in this chapter we will refer to designing a *strategy* for a *decision maker* (DM). The *planning problem* reduces down to a *decision-making problem*. In later chapters, which describe sequential decision making, planning terminology will once again be used. It does not seem appropriate yet in this chapter because making a single decision appears too degenerate to be referred to as planning.

A consistent theme throughout Part III will be the interaction of multiple DMs. In addition to the primary DM, which has been referred to as the robot, there will be one or more other DMs that cannot be predicted or controlled by the robot. A special DM called *nature* will be used as a universal way to model uncertainties. Nature will usually be fictitious in the sense that it is not a true entity that makes intelligent, rational decisions for its own benefit. The introduction of nature merely serves as a convenient modeling tool to express many different forms of uncertainty. In some settings, however, the DMs may actually be intelligent opponents who make decisions out of their own self-interest. This leads to *game theory*, in which all decision makers (including the robot) can be called *players*.

Section 9.1 provides some basic review and perspective that will help in understanding and relating later concepts in the chapter. Section 9.2 covers making a single decision under uncertainty, which is typically referred to as *decision theory*.

Sections 9.3 and 9.4 address *game theory*, in which two or more DMs make their decisions simultaneously and have conflicting interests. In *zero-sum game theory*, which is covered in Section 9.3, there are two DMs that have diametrically opposed interests. In *nonzero-sum game theory*, covered in Section 9.4, any number of DMs come together to form a *noncooperative game*, in which any degree of conflict or competition is allowable among them. Section 9.5 concludes the chapter by covering justifications and criticisms of the general models formulated in this chapter. It useful when trying to apply decision-theoretic models to planning problems in general.

This chapter was written without any strong dependencies on Part II. In fact, even the concepts from Chapter 2 are not needed because there are no stages or state spaces. Occasional references to Part II will be given, but these are not vital to the understanding. Most of the focus in this chapter is on discrete spaces.

9.1 Preliminary Concepts

9.1.1 Optimization

Optimizing a single objective

Before progressing to complicated decision-making models, first consider the simple case of a single decision maker that must make the best decision. This leads to a familiar *optimization* problem, which is formulated as follows.

Formulation 9.1 (Optimization)

1. A nonempty set U called the *action space*. Each $u \in U$ is referred to as an *action*.
2. A function $L : U \to \mathbb{R} \cup \{\infty\}$ called the *cost function*.

Compare Formulation 9.1 to Formulation 2.2. State space, X, and state transition concepts are no longer needed because there is only one decision. Since there is no state space, there is also no notion of initial and goal states. A *strategy* simply consists of selecting the best action.

What does it mean to be the "best" action? If U is finite, then the best action, $u^* \in U$ is

$$u^* = \operatorname*{argmin}_{u \in U} \Big\{ L(u) \Big\}. \qquad (9.1)$$

If U is infinite, then there are different cases. Suppose that $U = (-1, 1)$ and $L(u) = u$. Which action produces the lowest cost? We would like to declare that -1 is the lowest cost, but $-1 \notin U$. If we had instead defined $U = [-1, 1]$, then this would work. However, if $U = (-1, 1)$ and $L(u) = u$, then there is no action that produces minimum cost. For any action $u \in U$, a second one, $u' \in U$, can always be chosen for which $L(u') < L(u)$. However, if $U = (-1, 1)$ and $L(u) = |u|$, then

(9.1) correctly reports that $u = 0$ is the best action. There is no problem in this case because the minimum occurs in the interior, as opposed to on the boundary of U. In general it is important to be aware that an optimal value may not exist.

There are two ways to fix this frustrating behavior. One is to require that U is a closed set and is bounded (both were defined in Section 4.1). Since closed sets include their boundary, this problem will be avoided. The bounded condition prevents a problem such as optimizing $U = \mathbb{R}$, and $L(u) = u$. What is the best $u \in U$? Smaller and smaller values can be chosen for u to produce a lower cost, even though $\mathbb{R}$ is a closed set.

The alternative way to fix this problem is to define and use the notion of an *infimum*, denoted by inf. This is defined as the largest lower bound that can be placed on the cost. In the case of $U = (-1, 1)$ and $L(u) = u$, this is

$$\inf_{u \in (-1,1)} \Big\{ L(u) \Big\} = -1. \tag{9.2}$$

The only difficulty is that there is no action $u \in U$ that produces this cost. The infimum essentially uses the closure of U to evaluate (9.2). If U happened to be closed already, then u would be included in U. Unbounded problems can also be handled. The infimum for the case of $U = \mathbb{R}$ and $L(u) = u$ is $-\infty$.

As a general rule, if you are not sure which to use, it is safer to write inf in the place were you would use min. The infimum happens to yield the minimum whenever a minimum exists. In addition, it gives a reasonable answer when no minimum exists. It may look embarrassing, however, to use inf in cases where it is obviously not needed (i.e., in the case of a finite U).

It is always possible to make an "upside-down" version of an optimization problem by multiplying L by -1. There is no fundamental change in the result, but sometimes it is more natural to formulate a problem as one of maximization instead of minimization. This will be done, for example, in the discussion of utility theory in Section 9.5.1. In such cases, a *reward function*, R, is defined instead of a cost function. The task is to select an action $u \in U$ that *maximizes* the reward. It will be understood that a maximization problem can easily be converted into a minimization problem by setting $L(u) = -R(u)$ for all $u \in U$. For maximization problems, the infimum can be replaced by the *supremum*, sup, which is the least upper bound on $R(u)$ over all $u \in U$.

For most problems in this book, the selection of an optimal $u \in U$ in a single decision stage is straightforward; planning problems are instead complicated by many other aspects. It is important to realize, however, that optimization itself is an extremely challenging if U and L are complicated. For example, U may be finite but extremely large, or U may be a high-dimensional (e.g., 1000) subset of $\mathbb{R}^n$. Also, the cost function may be extremely difficult or even impossible to express in a simple closed form. If the function is simple enough, then standard calculus tools based on first and second derivatives may apply. It most real-world applications, however, more sophisticated techniques are needed. Many involve a form of gradient descent and therefore only ensure that a local minimum is

found. In many cases, sampling-based techniques are needed. In fact, many of the sampling ideas of Section 5.2, such as dispersion, were developed in the context of optimization. For some classes of problems, combinatorial solutions may exist. For example, *linear programming* involves finding the min or max of a collection of linear functions, and many combinatorial approaches exist [65, 68, 178, 201]. This optimization problem will appear in Section 9.4.

Given the importance of sampling-based and combinatorial methods in optimization, there are interesting parallels to motion planning. Chapters 5 and 6 each followed these two philosophies, respectively. Optimal motion planning actually corresponds to an optimization problem on the space of paths, which is extremely difficult to characterize. In some special cases, as in Section 6.2.4, it is possible to find optimal solutions, but in general, such problems are extremely challenging. *Calculus of variations* is a general approach for addressing optimization problems over a space of paths that must satisfy differential constraints [242]; this will be covered in Section 13.4.1.

Multiobjective optimization

Suppose that there is a collection of cost functions, each of which evaluates an action. This leads to a generalization of Formulation 9.1 to multiobjective optimization.

Formulation 9.2 (Multiobjective Optimization)

1. A nonempty set U called the *action space*. Each $u \in U$ is referred to as an *action*.

2. A vector-valued *cost function* of the form $L : U \rightarrow \mathbb{R}^d$ for some integer d. If desired, ∞ may also be allowed for any of the cost components.

A version of this problem was considered in Section 7.7.2, which involved the optimal coordination of multiple robots. Two actions, u and u', are called *equivalent* if $L(u) = L(u')$. An action u is said to *dominate* an action u' if they are not equivalent and $L_i(u) \leq L_i(u')$ for all i such that $1 \leq i \leq d$. This defines a partial ordering, $\leq$, on the set of actions. Note that many actions may be *incomparable*. An action is called *Pareto optimal* if it is not dominated by any others. This means that it is minimal with respect to the partial ordering.

Example 9.1 (Simple Example of Pareto Optimality) Suppose that $U = \{1, 2, 3, 4, 5\}$ and $d = 2$. The costs are assigned as $L(1) = (4, 0)$, $L(2) = (3, 3)$, $L(3) = (2, 2)$, $L(4) = (5, 7)$, and $L(5) = (9, 0)$. The actions 2, 4, and 5 can be eliminated because they are dominated by other actions. For example, $(3, 3)$ is dominated by $(2, 2)$; hence, action $u = 3$ is preferable to $u = 2$. The remaining two actions, $u = 1$ and $u = 3$, are Pareto optimal. ■

Based on this simple example, the notion of Pareto optimality seems mostly aimed at discarding dominated actions. Although there may be multiple Pareto-optimal solutions, it at least narrows down U to a collection of the best alternatives.

Example 9.2 (Pennsylvania Turnpike) Imagine driving across the state of Pennsylvania and being confronted with the Pennsylvania Turnpike, which is a toll highway that once posted threatening signs about speed limits and the according fines for speeding. Let $U = \{50, 51, \ldots, 100\}$ represent possible integer speeds, expressed in miles per hour (mph). A posted sign indicates that the speeding fines are 1) \$50 for being caught driving between 56 and 65 mph, 2) \$100 for being caught between 66 and 75, 3) \$200 between 76 and 85, and 4) \$500 between 86 and 100. Beyond 100 mph, it is assumed that the penalty includes jail time, which is so severe that it will not be considered.

The two criteria for a driver are 1) the time to cross the state, and 2) the amount of money spent on tickets. It is assumed that you will be caught violating the speed limit. The goal is to minimize both. What are the resulting Pareto-optimal driving speeds? Compare driving 56 mph to driving 57 mph. Both cost the same amount of money, but driving 57 mph takes less time. Therefore, 57 mph dominates 56 mph. In fact, 65 mph dominates all speeds down to 56 mph because the cost is the same, and it reduces the time the most. Based on this argument, the Pareto-optimal driving speeds are 55, 65, 75, 85, and 100. It is up to the individual drivers to decide on the particular best action for them; however, it is clear that no speeds outside of the Pareto-optimal set are sensible. ■

The following example illustrates the main frustration with Pareto optimality. Removing nondominated solutions may not be useful enough. In come cases, there may even be a continuum of Pareto-optimal solutions. Therefore, the Pareto-optimal concept is not always useful. Its value depends on the particular application.

Example 9.3 (A Continuum of Pareto-Optimal Solutions) Let $U = [0, 1]$ and $d = 2$. Let $L(u) = (u, 1 - u)$. In this case, every element of U is Pareto optimal. This can be seen by noting that a slight reduction in one criterion causes an increase in the other. Thus, any two actions are incomparable. ■

9.1.2 Probability Theory Review

This section reviews some basic probability concepts and introduces notation that will be used throughout Part III.

Probability space A *probability space* is a three-tuple, $(S, \mathcal{F}, P)$, in which the three components are

1. **Sample space:** A nonempty set S called the *sample space*, which represents all possible outcomes.

2. **Event space:** A collection $\mathcal{F}$ of subsets of S, called the *event space*. If S is discrete, then usually $\mathcal{F} = \mathrm{pow}(S)$. If S is continuous, then $\mathcal{F}$ is usually a sigma-algebra on S, as defined in Section 5.1.3.

3. **Probability function:** A function, $P : \mathcal{F} \rightarrow \mathbb{R}$, that assigns probabilities to the events in $\mathcal{F}$. This will sometimes be referred to as a *probability distribution* over S.

The probability function, P, must satisfy several basic axioms:

1. $P(E) \geq 0$ for all $E \in \mathcal{F}$.

2. $P(S) = 1$.

3. $P(E \cup F) = P(E) + P(F)$ if $E \cap F = \emptyset$, for all $E, F \in \mathcal{F}$.

If S is discrete, then the definition of P over all of $\mathcal{F}$ can be inferred from its definition on single elements of S by using the axioms. It is common in this case to write $P(s)$ for some $s \in S$, which is slightly abusive because s is not an event. It technically should be $P(\{s\})$ for some $\{s\} \in \mathcal{F}$.

Example 9.4 (Tossing a Die) Consider tossing a six-sided cube or die that has numbers 1 to 6 painted on its sides. When the die comes to rest, it will always show one number. In this case, $S = \{1, 2, 3, 4, 5, 6\}$ is the sample space. The event space is $\mathrm{pow}(S)$, which is all 2^6 subsets of S. Suppose that the probability function is assigned to indicate that all numbers are equally likely. For any individual $s \in S$, $P(\{s\}) = 1/6$. The events include all subsets so that any probability statement can be formulated. For example, what is the probability that an even number is obtained? The event $E = \{2, 4, 6\}$ has probability $P(E) = 1/2$ of occurring. ■

The third probability axiom looks similar to the last axiom in the definition of a measure space in Section 5.1.3. In fact, P is technically a special kind of measure space as mentioned in Example 5.12. If S is continuous, however, this measure cannot be captured by defining probabilities over the singleton sets. The probabilities of singleton sets are usually zero. Instead, a *probability density function*, $p : S \rightarrow \mathbb{R}$, is used to define the probability measure. The probability function, P, for any event $E \in \mathcal{F}$ can then be determined via integration:

$$P(E) = \int_E p(x)dx, \tag{9.3}$$

in which $x \in E$ is the variable of integration. Intuitively, P indicates the total probability mass that accumulates over E.

Conditional probability A *conditional probability* is expressed as $P(E|F)$ for any two events $E, F \in \mathcal{F}$ and is called the "probability of E, given F." Its definition is

$$P(E|F) = \frac{P(E \cap F)}{P(F)}. \tag{9.4}$$

Two events, E and F, are called *independent* if and only if $P(E \cap F) = P(E)P(F)$; otherwise, they are called *dependent*. An important and sometimes misleading concept is *conditional independence*. Consider some third event, $G \in \mathcal{F}$. It might be the case that E and F are dependent, but when G is given, they become independent. Thus, $P(E \cap F) \neq P(E)P(F)$; however, $P(E \cap F|G) = P(E|G)P(F|G)$. Such examples occur frequently in practice. For example, E might indicate someone's height, and F is their reading level. These will generally be dependent events because children are generally shorter and have a lower reading level. If we are given the person's age as an event G, then height is no longer important. It seems intuitive that there should be no correlation between height and reading level once the age is given.

The definition of conditional probability, (9.4), imposes the constraint that

$$P(E \cap F) = P(F)P(E|F) = P(E)P(F|E), \tag{9.5}$$

which nicely relates $P(E|F)$ to $P(F|E)$. This results in *Bayes' rule*, which is a convenient way to swap E and F:

$$P(F|E) = \frac{P(E|F)P(F)}{P(E)}. \tag{9.6}$$

The probability distribution, $P(F)$, is referred to as the *prior*, and $P(F|E)$ is the *posterior*. These terms indicate that the probabilities come before and after E is considered, respectively.

If all probabilities are conditioned on some event, $G \in \mathcal{F}$, then *conditional Bayes' rule* arises, which only differs from (9.6) by placing the condition G on all probabilities:

$$P(F|E,G) = \frac{P(E|F,G)P(F|G)}{P(E|G)}. \tag{9.7}$$

Marginalization Let the events $F_1, F_2, \ldots, F_n$ be any partition of S. The probability of an event E can be obtained through *marginalization* as

$$P(E) = \sum_{i=1}^{n} P(E|F_i)P(F_i). \tag{9.8}$$

One of the most useful applications of marginalization is in the denominator of Bayes' rule. A substitution of (9.8) into the denominator of (9.6) yields

$$P(F|E) = \frac{P(E|F)P(F)}{\displaystyle\sum_{i=1}^{n} P(E|F_i)P(F_i)}. \tag{9.9}$$

This form is sometimes easier to work with because $P(E)$ appears to be eliminated.

Random variables Assume that a probability space $(S, \mathcal{F}, P)$ is given. A *random variable*[1] X is a function that maps S into $\mathbb{R}$. Thus, X assigns a real value to every element of the sample space. This enables statistics to be conveniently computed over a probability space. If S is already a subset of $\mathbb{R}$, X may by default represent the identity function.

Expectation The *expectation* or *expected value* of a random variable X is denoted by $E[X]$. It can be considered as a kind of weighted average for X, in which the weights are obtained from the probability distribution. If S is discrete, then

$$E[X] = \sum_{s \in S} X(s)P(s). \tag{9.10}$$

If S is continuous, then[2]

$$E[X] = \int_S X(s)p(s)ds. \tag{9.11}$$

One can then define *conditional expectation*, which applies a given condition to the probability distribution. For example, if S is discrete and an event F is given, then

$$E[X|F] = \sum_{s \in S} X(s)P(s|F). \tag{9.12}$$

Example 9.5 (Tossing Dice) Returning to Example 9.4, the elements of S are already real numbers. Hence, a random variable X can be defined by simply letting $X(s) = s$. Using (9.11), the expected value, $E[X]$, is 3.5. Note that the expected value is not necessarily a value that is "expected" in practice. It is impossible to actually obtain 3.5, even though it is not contained in S. Suppose that the expected value of X is desired only over trials that result in numbers greater then 3. This can be described by the event $F = \{4, 5, 6\}$. Using conditional expectation, (9.12), the expected value is $E[X|F] = 5$.

[1]This is a terrible name, which often causes confusion. A random variable is not "random," nor is it a "variable." It is simply a function, $X : S \to \mathbb{R}$. To make matters worse, a capital letter is usually used to denote it, whereas lowercase letters are usually used to denote functions.

[2]Using the language of measure theory, both definitions are just special cases of the Lebesgue integral. Measure theory nicely unifies discrete and continuous probability theory, thereby avoiding the specification of separate cases. See [103, 148, 239].

Now consider tossing two dice in succession. Each element $s \in S$ is expressed as $s = (i, j)$ in which $i, j \in \{1, 2, 3, 4, 5, 6\}$. Since $S \not\subset \mathbb{R}$, the random variable needs to be slightly more interesting. One common approach is to count the sum of the dice, which yields $X(s) = i + j$ for any $s \in S$. In this case, $E[X] = 7$. ■

9.1.3 Randomized Strategies

Up until now, any actions taken in a plan have been *deterministic*. The plans in Chapter 2 specified actions with complete certainty. Formulation 9.1 was solved by specifying the best action. It can be viewed as a *strategy* that trivially makes the same decision every time.

In some applications, the decision maker may not want to be predictable. To achieve this, randomization can be incorporated into the strategy. If U is discrete, a *randomized strategy*, w, is specified by a probability distribution, $P(u)$, over U. Let W denote the set of all possible randomized strategies. When the strategy is applied, an action $u \in U$ is chosen by sampling according to the probability distribution, $P(u)$. We now have to make a clear distinction between *defining the strategy* and *applying the strategy*. So far, the two have been equivalent; however, a randomized strategy must be *executed* to determine the resulting action. If the strategy is executed repeatedly, it is assumed that each trial is independent of the actions obtained in previous trials. In other words, $P(u_k|u_i) = P(u_k)$, in which $P(u_k|u_i)$ represents the probability that the strategy chooses action u_k in trial k, given that u_i was chosen in trial i for some $i < k$. If U is continuous, then a randomized strategy may be specified by a probability density function, $p(u)$. In decision-theory and game-theory literature, deterministic and randomized strategies are often referred to as *pure* and *mixed*, respectively.

Example 9.6 (Basing Decisions on a Coin Toss) Let $U = \{a, b\}$. A randomized strategy w can be defined as

1. Flip a fair coin, which has two possible outcomes: heads (H) or tails (T).
2. If the outcome is H, choose a; otherwise, choose b.

Since the coin is fair, w is defined by assigning $P(a) = P(b) = 1/2$. Each time the strategy is applied, it not known what action will be chosen. Over many trials, however, it converges to choosing a half of the time. ■

A deterministic strategy can always be viewed as a special case of a randomized strategy, if you are not bothered by events that have probability zero. A deterministic strategy, $u_i \in U$, can be simulated by a random strategy by assigning $P(u) = 1$ if $u = u_i$, and $P(u) = 0$ otherwise. Only with probability zero can different actions be chosen (possible, but not probable!).

Imagine using a randomized strategy to solve a problem expressed using Formulation 9.1. The first difficulty appears to be that the cost cannot be predicted. If the strategy is applied numerous times, then we can define the average cost. As the number of times tends to infinity, this average would converge to the expected cost, denoted by $\bar{L}(w)$, if L is treated as a random variable (in addition to the cost function). If U is discrete, the expected cost of a randomized strategy w is

$$\bar{L}(w) = \sum_{u \in U} L(u)P(u) = \sum_{u \in U} L(u)w_i, \tag{9.13}$$

in which w_i is the component of w corresponding to the particular $u \in U$.

An interesting question is whether there exists some $w \in W$ such that $\bar{L}(w) < L(u)$, for all $u \in U$. In other words, do there exist randomized strategies that are better than all deterministic strategies, using Formulation 9.1? The answer is *no* because the best strategy is always to assign probability one to the action, u^*, that minimizes L. This is equivalent to using a deterministic strategy. If there are two or more actions that obtain the optimal cost, then a randomized strategy could arbitrarily distribute all of the probability mass between these. However, there would be no further reduction in cost. Therefore, randomization seems pointless in this context, unless there are other considerations.

One important example in which a randomized strategy is of critical importance is when making decisions in competition with an intelligent adversary. If the problem is repeated many times, an opponent could easily learn any deterministic strategy. Randomization can be used to weaken the prediction capabilities of an opponent. This idea will be used in Section 9.3 to obtain better ways to play zero-sum games.

Following is an example that illustrates the advantage of randomization when repeatedly playing against an intelligent opponent.

Example 9.7 (Matching Pennies) Consider a game in which two players repeatedly play a simple game of placing pennies on the table. In each trial, the players must place their coins simultaneously with either heads (H) facing up or tails (T) facing up. Let a two-letter string denote the outcome. If the outcome is HH or TT (the players choose the same), then Player 1 pays Player 2 one Peso; if the outcome is HT or TH, then Player 2 pays Player 1 one Peso. What happens if Player 1 uses a deterministic strategy? If Player 2 can determine the strategy, then he can choose his strategy so that he always wins the game. However, if Player 1 chooses the best randomized strategy, then he can expect at best to break even on average. What randomized strategy achieves this?

A generalization of this to three actions is the famous game of Rock-Paper-Scissors [286]. If you want to design a computer program that repeatedly plays this game against smart opponents, it seems best to incorporate randomization. ■

9.2 A Game Against Nature

9.2.1 Modeling Nature

For the first time in this book, uncertainty will be directly modeled. There are two DMs:

> **Robot:** This is the name given to the primary DM throughout the book. So far, there has been only one DM. Now that there are two, the name is more important because it will be used to distinguish the DMs from each other.
>
> **Nature:** This DM is a mysterious force that is unpredictable to the robot. It has its own set of actions, and it can choose them in a way that interferes with the achievements of the robot. Nature can be considered as a synthetic DM that is constructed for the purposes of modeling uncertainty in the decision-making or planning process.

Imagine that the robot and nature each make a decision. Each has a set of actions to choose from. Suppose that the cost depends on which actions are chosen by each. The cost still represents the effect of the outcome on the robot; however, the robot must now take into account the influence of nature on the cost. Since nature is unpredictable, the robot must formulate a model of its behavior. Assume that the robot has a set, U, of actions, as before. It is now assumed that nature also has a set of actions. This is referred to as the *nature action space* and is denoted by Θ. A *nature action* is denoted as $\theta \in \Theta$. It now seems appropriate to call U the *robot action space*; however, for convenience, it will often be referred to as the *action space*, in which the *robot* is implied.

This leads to the following formulation, which extends Formulation 9.1.

Formulation 9.3 (A Game Against Nature)

1. A nonempty set U called the *(robot) action space*. Each $u \in U$ is referred to as an *action*.
2. A nonempty set Θ called the *nature action space*. Each $\theta \in \Theta$ is referred to as a *nature action*.
3. A function $L : U \times \Theta \rightarrow \mathbb{R} \cup \{\infty\}$, called the *cost function*.

The cost function, L, now depends on $u \in U$ and $\theta \in \Theta$. If U and Θ are finite, then it is convenient to specify L as a $|U| \times |\Theta|$ matrix called the *cost matrix*.

Example 9.8 (A Simple Game Against Nature) Suppose that U and Θ each contain three actions. This results in nine possible outcomes, which can be specified by the following cost matrix:

$$\begin{array}{c|c|c|c|} & \multicolumn{3}{c}{\Theta} \\ \cline{2-4} & 1 & -1 & 0 \\ \cline{2-4} U & -1 & 2 & -2 \\ \cline{2-4} & 2 & -1 & 1 \\ \cline{2-4} \end{array}$$

The robot action, $u \in U$, selects a row, and the nature action, $\theta \in \Theta$, selects a column. The resulting cost, $L(u, \theta)$, is given by the corresponding matrix entry. ■

In Formulation 9.3, it appears that both DMs act at the same time; nature does not know the robot action before deciding. In many contexts, nature may know the robot action. In this case, a different nature action space can be defined for every $u \in U$. This generalizes Formulation 9.3 to obtain:

Formulation 9.4 (Nature Knows the Robot Action)

1. A nonempty set U called the *action space*. Each $u \in U$ is referred to as an *action*.
2. For each $u \in U$, a nonempty set $\Theta(u)$ called the *nature action space*.
3. A function $L : U \times \Theta \rightarrow \mathbb{R} \cup \{\infty\}$, called the *cost function*.

If the robot chooses an action $u \in U$, then nature chooses from $\Theta(u)$.

9.2.2 Nondeterministic vs. Probabilistic Models

What is the best decision for the robot, given that it is engaged in a game against nature? This depends on what information the robot has regarding how nature chooses its actions. It will always be assumed that the robot does not know the precise nature action to be chosen; otherwise, it is pointless to define nature. Two alternative models that the robot can use for nature will be considered. From the robot's perspective, the possible models are

> **Nondeterministic**: I have no idea what nature will do.
>
> **Probabilistic**: I have been observing nature and gathering statistics.

Under both models, it is assumed that the robot knows Θ in Formulation 9.3 or $\Theta(u)$ for all $u \in U$ in Formulation 9.4. The nondeterministic and probabilistic terminology are borrowed from Erdmann [92]. In some literature, the term *possibilistic* is used instead of *nondeterministic*. This is an excellent term, but it is unfortunately too similar to *probabilistic* in English.

Assume first that Formulation 9.3 is used and that U and Θ are finite. Under the nondeterministic model, there is no additional information. One reasonable approach in this case is to make a decision by assuming the worst. It can even be imagined that nature knows what action the robot will take, and it will spitefully choose a nature action that drives the cost as high as possible. This pessimistic view is sometimes humorously referred to as Murphy's Law ("If anything can go wrong, it will.") [33] or Sod's Law. In this case, the best action, $u^* \in U$, is selected as

$$u^* = \operatorname*{argmin}_{u \in U} \Big\{ \max_{\theta \in \Theta} \Big\{ L(u,\theta) \Big\} \Big\}. \tag{9.14}$$

The action u^* is the lowest cost choice using *worst-case analysis*. This is sometimes referred to as a *minimax* solution because of the min and max in (9.14). If U or Θ is infinite, then the min or max may not exist and should be replaced by inf or sup, respectively.

Worst-case analysis may seem too pessimistic in some applications. Perhaps the assumption that all actions in Θ are equally likely may be preferable. This can be handled as a special case of the probabilistic model, which is described next.

Under the probabilistic model, it is assumed that the robot has gathered enough data to reliably estimate $P(\theta)$ (or $p(\theta)$ if Θ is continuous). In this case, it is imagined that nature applies a randomized strategy, as defined in Section 9.1.3. It assumed that the applied nature actions have been observed over many trials, and in the future they will continue to be chosen in the same manner, as predicted by the distribution $P(\theta)$. Instead of worst-case analysis, *expected-case analysis* is used. This optimizes the average cost to be received over numerous independent trials. In this case, the best action, $u^* \in U$, is

$$u^* = \operatorname*{argmin}_{u \in U} \Big\{ E_\theta \Big[L(u,\theta) \Big] \Big\}, \tag{9.15}$$

in which E_θ indicates that the expectation is taken according to the probability distribution (or density) over θ. Since Θ and $P(\theta)$ together form a probability space, $L(u,\theta)$ can be considered as a random variable for each value of u (it assigns a real value to each element of the sample space).[3] Using $P(\theta)$, the expectation in (9.15) can be expressed as

$$E_\theta[L(u,\theta)] = \sum_{\theta \in \Theta} L(u,\theta)P(\theta). \tag{9.16}$$

Example 9.9 (Nondeterministic vs. Probabilistic) Return to Example 9.8. Let $U = \{u_1, u_2, u_3\}$ represent the robot actions, and let $\Theta = \{\theta_1, \theta_2, \theta_3\}$ represent the nature actions.

Under the nondeterministic model of nature, $u^* = u_1$, which results in $L(u^*,\theta) = 1$ in the worst case using (9.14). Under the probabilistic model, let $P(\theta_1) = 1/5$, $P(\theta_2) = 1/5$, and $P(\theta_3) = 3/5$. To find the optimal action, (9.15) can be used.

[3] Alternatively, a random variable may be defined over $U \times \Theta$, and conditional expectation would be taken, in which u is given.

This involves computing the expected cost for each action:

$$\begin{aligned} E_\theta[L(u_1,\theta)] &= (1)1/5 + (-1)1/5 + (0)3/5 = 0 \\ E_\theta[L(u_2,\theta)] &= (-1)1/5 + (2)1/5 + (-2)3/5 = -1 \\ E_\theta[L(u_3,\theta)] &= (2)1/5 + (-1)1/5 + (1)3/5 = 4/5. \end{aligned} \tag{9.17}$$

The best action is $u^* = u_2$, which produces the lowest expected cost, -1.

If the probability distribution had instead been $P = [1/10 \;\; 4/5 \;\; 1/10]$, then $u^* = u_1$ would have been obtained. Hence the best decision depends on $P(\theta)$; if this information is statistically valid, then it enables more informed decisions to be made. If such information is not available, then the nondeterministic model may be more suitable.

It is possible, however, to assign $P(\theta)$ as a uniform distribution in the absence of data. This means that all nature actions are equally likely; however, conclusions based on this are dangerous; see Section 9.5. ■

In Formulation 9.4, the nature action space $\Theta(u)$ depends on $u \in U$, the robot action. Under the nondeterministic model, (9.14) simply becomes

$$u^* = \operatorname*{argmin}_{u \in U} \Big\{ \max_{\theta \in \Theta(u)} L(u,\theta) \Big\}. \tag{9.18}$$

Unfortunately, these problems do not have a nice matrix representation because the size of $\Theta(u)$ can vary for different $u \in U$. In the probabilistic case, $P(\theta)$ is replaced by a conditional probability distribution $P(\theta|u)$. Estimating this distribution requires observing numerous independent trials for each possible $u \in U$. The behavior of nature can now depend on the robot action; however, nature is still characterized by a randomized strategy. It does not adapt its strategy across multiple trials. The expectation in (9.16) now becomes

$$E_\theta \Big[L(u,\theta) \Big] = \sum_{\theta \in \Theta(u)} L(u,\theta)P(\theta|u), \tag{9.19}$$

which replaces $P(\theta)$ by $P(\theta|u)$.

Regret It is important to note that the models presented here are not the only accepted ways to make good decisions. In game theory, the key idea is to minimize "regret." This is the feeling you get after making a bad decision and wishing that you could change it after the game is finished. Suppose that after you choose some $u \in U$, you are told which $\theta \in \Theta$ was applied by nature. The regret is the amount of cost that you could have saved by picking a different action, given the nature action that was applied.

For each combination of $u \in U$ and $\theta \in \Theta$, the *regret*, T, is defined as

$$T(u,\theta) = \max_{u' \in U} \Big\{ L(u,\theta) - L(u',\theta) \Big\}. \tag{9.20}$$

For Formulation 9.3, if U and Θ are finite, then a $|\Theta| \times |U|$ *regret matrix* can be defined.

Suppose that minimizing regret is the primary concern, as opposed to the actual cost received. Under the nondeterministic model, the action that minimizes the worst-case regret is

$$u^* = \underset{u \in U}{\operatorname{argmin}} \Big\{ \max_{\theta \in \Theta} \Big\{ T(u,\theta) \Big\} \Big\}. \tag{9.21}$$

In the probabilistic model, the action that minimizes the expected regret is

$$u^* = \underset{u \in U}{\operatorname{argmin}} \Big\{ E_\theta \Big[T(u,\theta) \Big] \Big\}. \tag{9.22}$$

The only difference with respect to (9.14) and (9.15) is that L has been replaced by T. In Section 9.3.2, regret will be discussed in more detail because it forms the basis of optimality concepts in game theory.

Example 9.10 (Regret Matrix) The regret matrix for Example 9.8 is

$$\begin{array}{c|c|c|c|} & \multicolumn{3}{c}{\Theta} \\ \cline{2-4} & 2 & 0 & 2 \\ \cline{2-4} U & 0 & 3 & 0 \\ \cline{2-4} & 3 & 0 & 3 \\ \cline{2-4} \end{array}$$

Using the nondeterministic model, $u^* = u_1$, which results in a worst-case regret of 2 using (9.21). Under the probabilistic model, let $P(\theta_1) = P(\theta_2) = P(\theta_3) = 1/3$. In this case, $u^* = u_1$, which yields the optimal expected regret, calculated as 1 using (9.22).

9.2.3 Making Use of Observations

Formulations 9.3 and 9.4 do not allow the robot to receive any information (other than L) prior to making its decision. Now suppose that the robot has a sensor that it can check just prior to choosing the best action. This sensor provides an *observation* or measurement that contains information about which nature action might be chosen. In some contexts, the nature action can be imagined as a kind of *state* that has already been selected. The observation then provides information about this. For example, nature might select the current temperature in Bangkok. An observation could correspond to a thermometer in Bangkok that takes a reading.

Formulating the problem Let Y denote the *observation space*, which is the set of all possible observations, $y \in Y$. For convenience, suppose that Y, U, and Θ are all discrete. It will be assumed as part of the model that some constraints on

θ are known once y is given. Under the nondeterministic model a set $Y(\theta) \subseteq Y$ is specified for every $\theta \in \Theta$. The set $Y(\theta)$ indicates the possible observations, given that the nature action is θ. Under the probabilistic model a conditional probability distribution, $P(y|\theta)$, is specified. Examples of sensing models will be given in Section 9.2.4. Many others appear in Sections 11.1.1 and 11.5.1, although they are expressed with respect to a state space X that reduces to Θ in this section. As before, the probabilistic case also requires a prior distribution, $P(\Theta)$, to be given. This results in the following formulation.

Formulation 9.5 (A Game Against Nature with an Observation)

1. A finite, nonempty set U called the *action space*. Each $u \in U$ is referred to as an *action*.
2. A finite, nonempty set Θ called the *nature action space*.
3. A finite, nonempty set Y called the *observation space*.
4. A set $Y(\theta) \subseteq Y$ or probability distribution $P(y|\theta)$ specified for every $\theta \in \Theta$. This indicates which observations are possible or probable, respectively, if θ is the nature action. In the probabilistic case a prior, $P(\theta)$, must also be specified.
5. A function $L : U \times \Theta \to \mathbb{R} \cup \{\infty\}$, called the *cost function*.

Consider solving Formulation 9.5. A strategy is now more complicated than simply specifying an action because we want to completely characterize the behavior of the robot before the observation has been received. This is accomplished by defining a *strategy* as a function, $\pi : Y \to U$. For each possible observation, $y \in Y$, the strategy provides an action. We now want to search the space of possible strategies to find the one that makes the best decisions over all possible observations. In this section, Y is actually a special case of an information space, which is the main topic of Chapters 11 and 12. Eventually, a strategy (or plan) will be conditioned on an information state, which generalizes an observation.

Optimal strategies Now consider finding the optimal strategy, denoted by π^*, under the nondeterministic model. The sets $Y(\theta)$ for each $\theta \in \Theta$ must be used to determine which nature actions are possible for each observation, $y \in Y$. Let $\Theta(y)$ denote this, which is obtained as

$$\Theta(y) = \{\theta \in \Theta \mid y \in Y(\theta)\}. \tag{9.23}$$

The optimal strategy, π^*, is defined by setting

$$\pi^*(y) = \underset{u \in U}{\operatorname{argmin}} \Big\{ \max_{\theta \in \Theta(y)} \Big\{ L(u,\theta) \Big\} \Big\}, \tag{9.24}$$

for each $y \in Y$. Compare this to (9.14), in which the maximum was taken over all Θ. The advantage of having the observation, y, is that the set is restricted to $\Theta(y) \subseteq \Theta$.

Under the probabilistic model, an operation analogous to (9.23) must be performed. This involves computing $P(\theta|y)$ from $P(y|\theta)$ to determine the information that y contains regarding θ. Using Bayes' rule, (9.9), with marginalization on the denominator, the result is

$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{\displaystyle\sum_{\theta \in \Theta} P(y|\theta)P(\theta)}. \tag{9.25}$$

To see the connection between the nondeterministic and probabilistic cases, define a probability distribution, $P(y|\theta)$, that is nonzero only if $y \in Y(\theta)$ and use a uniform distribution for $P(\theta)$. In this case, (9.25) assigns nonzero probability to precisely the elements of $\Theta(y)$ as given in (9.23). Thus, (9.25) is just the probabilistic version of (9.23). The optimal strategy, π^*, is specified for each $y \in Y$ as

$$\pi^*(y) = \operatorname*{argmin}_{u \in U} \Big\{ E_\theta \Big[L(u,\theta) \;\Big|\; y \Big] \Big\} = \operatorname*{argmin}_{u \in U} \Big\{ \sum_{\theta \in \Theta} L(u,\theta) P(\theta|y) \Big\}. \tag{9.26}$$

This differs from (9.15) and (9.16) by replacing $P(\theta)$ with $P(\theta|y)$. For each u, the expectation in (9.26) is called the *conditional Bayes' risk*. The optimal strategy, π^*, always selects the strategy that minimizes this risk. Note that $P(\theta|y)$ in (9.26) can be expressed using (9.25), for which the denominator (9.26) represents $P(y)$ and does not depend on u; therefore, it does not affect the optimization. Due to this, $P(y|\theta)P(\theta)$ can be used in the place of $P(\theta|y)$ in (9.26), and the same π^* will be obtained. If the spaces are continuous, then probability densities are used in the place of all probability distributions, and the method otherwise remains the same.

Nature acts twice A convenient, alternative formulation can be given by allowing nature to act twice:

1. First, a nature action, $\theta \in \Theta$, is chosen but is unknown to the robot.
2. Following this, a *nature observation action* is chosen to interfere with the robot's ability to sense θ.

Let ψ denote a *nature observation action*, which is chosen from a *nature observation action space*, $\Psi(\theta)$. A *sensor mapping*, h, can now be defined that yields $y = h(\theta, \psi)$ for each $\theta \in \Theta$ and $\psi \in \Psi(\theta)$. Thus, for each of the two kinds of nature actions, $\theta \in \Theta$ and $\psi \in \Psi$, an observation, $y = h(\theta,\psi)$, is given. This yields an alternative way to express Formulation 9.5:

Formulation 9.6 (Nature Interferes with the Observation)

1. A nonempty, finite set U called the *action space*.
2. A nonempty, finite set Θ called the *nature action space*.
3. A nonempty, finite set Y called the *observation space*.
4. For each $\theta \in \Theta$, a nonempty set $\Psi(\theta)$ called the *nature observation action space*.
5. A sensor mapping $h : \Theta \times \Psi \rightarrow Y$.
6. A function $L : U \times \Theta \rightarrow \mathbb{R} \cup \{\infty\}$ called the *cost function*.

This nicely unifies the nondeterministic and probabilistic models with a single function h. To express a nondeterministic model, it is assumed that any $\psi \in \Psi(\theta)$ is possible. Using h,

$$\Theta(y) = \{\theta \in \Theta \mid \exists \psi \in \Psi(\theta) \text{ such that } y = h(\theta,\psi)\}. \tag{9.27}$$

For a probabilistic model, a distribution $P(\psi|\theta)$ is specified (often, this may reduce to $P(\psi)$). Suppose that when the domain of h is restricted to some $\theta \in \Theta$, then it forms an injective mapping from Ψ to Y. In other words, every nature observation action leads to a unique observation, assuming θ is fixed. Using $P(\psi)$ and h, $P(y|\theta)$ is derived as

$$P(y|\theta) = \begin{cases} P(\psi|\theta) & \text{for the unique } \psi \text{ such that } y = h(\theta,\psi). \\ 0 & \text{if no such } \psi \text{ exists.} \end{cases} \tag{9.28}$$

If the injective assumption is lifted, then $P(\psi|\theta)$ is replaced by a sum over all ψ for which $y = h(\theta,\psi)$. In Formulation 9.6, the only difference between the nondeterministic and probabilistic models is the characterization of ψ, which represents a kind of measurement interference. A strategy still takes the form $\pi : \Theta \rightarrow U$. A hybrid model is even possible in which one nature action is modeled nondeterministically and the other probabilistically.

Receiving multiple observations Another extension of Formulation 9.5 is to allow multiple observations, y_1, y_2, ..., y_n, before making a decision. Each y_i is assumed to belong to an observation space, Y_i. A strategy, π, now depends on all observations:

$$\pi : Y_1 \times Y_2 \times \cdots \times Y_n \rightarrow U. \tag{9.29}$$

Under the nondeterministic model, $Y_i(\theta)$ is specified for each i and $\theta \in \Theta$. The set $\Theta(y)$ is replaced by

$$\Theta(y_1) \cap \Theta(y_2) \cap \cdots \cap \Theta(y_n) \tag{9.30}$$

in (9.24) to obtain the optimal action, $\pi^*(y_1, \ldots, y_n)$.

Under the probabilistic model, $P(y_i|\theta)$ is specified instead. It is often assumed that the observations are conditionally independent given θ. This means for any y_i, θ, and y_j such that $i \neq j$, $P(y_i|\theta, y_j) = P(y_i|\theta)$. The condition $P(\theta|y)$ in (9.26) is replaced by $P(\theta|y_1, \ldots, y_n)$. Applying Bayes' rule, and using the conditional independence of the y_i's given θ, yields

$$P(\theta|y_1, \ldots, y_n) = \frac{P(y_1|\theta)P(y_2|\theta)\cdots P(y_n|\theta)P(\theta)}{P(y_1, \ldots, y_n)}. \tag{9.31}$$

The denominator can be treated as a constant factor that does not affect the optimization. Therefore, it does not need to be explicitly computed unless the optimal expected cost is needed in addition to the optimal action.

Conditional independence allows a dramatic simplification that avoids the full specification of $P(y|\theta)$. Sometimes the conditional independence assumption is used when it is incorrect, just to exploit this simplification. Therefore, a method that uses conditional independence of observations is often called *naive Bayes*.

9.2.4 Examples of Optimal Decision Making

The framework presented so far characterizes *statistical decision theory*, which covers a broad range of applications and research issues. Virtually any context in which a decision must be made automatically, by a machine or a person following specified rules, is a candidate for using these concepts. In Chapters 10 through 12, this decision problem will be repeatedly embedded into complicated planning problems. Planning will be viewed as a sequential decision-making process that iteratively modifies states in a state space. Most often, each decision step will be simpler than what usually arises in common applications of decision theory. This is because planning problems are complicated by many other factors. If the decision step in a particular application is already too hard to solve, then an extension to planning appears hopeless.

It is nevertheless important to recognize the challenges in general that arise when modeling and solving decision problems under the framework of this section. Some examples are presented here to help illustrate its enormous power and scope.

Pattern classification

An active field over the past several decades in computer vision and machine learning has been *pattern classification* [73, 82, 193]. The general problem involves using a set of data to perform classifications. For example, in computer vision, the data correspond to information extracted from an image. These indicate observed features of an object that are used by a vision system to try to classify the object (e.g., "I am looking at a bowl of Vietnamese noodle soup").

The presentation here represents a highly idealized version of pattern classification. We will assume that all of the appropriate model details, including

the required probability distributions, are available. In some contexts, these can be obtained by gathering statistics over large data sets. In many applications, however, obtaining such data is expensive or inaccessible, and classification techniques must be developed in lieu of good information. Some problems are even *unsupervised*, which means that the set of possible classes must also be discovered automatically. Due to issues such as these, pattern classification remains a challenging research field.

The general model is that nature first determines the class, then observations are obtained regarding the class, and finally the robot action attempts to guess the correct class based on the observations. The problem fits under Formulation 9.5. Let Θ denote a finite set of *classes*. Since the robot must guess the class, $U = \Theta$. A simple cost function is defined to measure the mismatch between u and θ:

$$L(u, \theta) = \begin{cases} 0 & \text{if } u = \theta \text{ (correct classification} \\ 1 & \text{if } u \neq \theta \text{ (incorrect classification)} \;. \end{cases} \tag{9.32}$$

The nondeterministic model yields a cost of 1 if it is *possible* that a classification error can be made using action u. Under the probabilistic model, the expectation of (9.32) gives the probability that a classification error will be made given an action u.

The next part of the formulation considers information that is used to make the classification decision. Let Y denote a *feature space*, in which each $y \in Y$ is called a *feature* or *feature vector* (often $y \in \mathbb{R}^n$). The feature in this context is just an observation, as given in Formulation 9.5. The best *classifier* or *classification rule* is a strategy $\pi : Y \to U$ that provides the smallest classification error in the worst case or expected case, depending on the model.

A Bayesian classifier The probabilistic approach is most common in pattern classification. This results in a *Bayesian classifier*. Here it is assumed that $P(y|\theta)$ and $P(\theta)$ are given. The distribution of features for a given class is indicated by $P(y|\theta)$. The overall frequency of class occurrences is given by $P(\theta)$. If large, preclassified data sets are available, then these distributions can be reliably learned. The feature space is often continuous, which results in a density $p(y|\theta)$, even though $P(\theta)$ remains a discrete probability distribution. An optimal classifier, π^*, is designed according to (9.26). It performs classification by receiving a feature vector, y, and then declaring that the class is $u = \pi^*(y)$. The expected cost using (9.32) is the probability of error.

Example 9.11 (Optical Character Recognition) An example of classification is given by a simplified *optical character recognition* (OCR) problem. Suppose that a camera creates a digital image of a page of text. Segmentation is first performed to determine the location of each letter. Following this, the individual letters must be classified correctly. Let $\Theta = \{A, B, C, D, E, F, G, H\}$, which would ordinarily include all of the letters of the alphabet.

Shape	0	A E F H
	1	B C D G
Ends	0	B D
	1	
	2	A C G
	3	F E
	4	H
Holes	0	C E F G H
	1	A D
	2	B

Figure 9.1: A mapping from letters to feature values.

Suppose that there are three different image processing algorithms:

Shape extractor: This returns $s = 0$ if the letter is composed of straight edges only, and $s = 1$ if it contains at least one curve.

End counter: This returns e, the number of segment ends. For example, O has none and X has four.

Hole counter: This returns h, the number of holes enclosed by the character. For example, X has none and O has one.

The feature vector is $y = (s, e, h)$. The values that should be reported under ideal conditions are shown in Figure 9.1. These indicate $\Theta(s)$, $\Theta(e)$, and $\Theta(h)$. The intersection of these yields $\Theta(y)$ for any combination of s, e, and h.

Imagine doing classification under the nondeterministic model, with the assumption that the features always provide correct information. For $y = (0, 2, 1)$, the only possible letter is A. For $y = (1, 0, 2)$, the only letter is B. If each (s, e, h) is consistent with only one or no letters, then a perfect classifier can be constructed. Unfortunately, $(0, 3, 0)$ is consistent with both E and F. In the worst case, the cost of using (9.32) is 1.

One way to fix this is to introduce a new feature. Suppose that an image processing algorithm is used to detect corners. These are places at which two segments meet at a right (90 degrees) angle. Let c denote the number of corners, and let the new feature vector be $y = (s, e, h, c)$. The new algorithm nicely distinguishes E from F, for which $c = 2$ and $c = 1$, respectively. Now all letters can be correctly classified without errors.

Of course, in practice, the image processing algorithms occasionally make mistakes. A Bayesian classifier can be designed to maximize the probability of success. Assume conditional independence of the observations, which means that the classifier can be considered *naive*. Suppose that the four image processing algorithms are run over a training data set and the results are recorded. In each case, the correct classification is determined by hand to obtain probabilities $P(s|\theta)$, $P(e|\theta)$, $P(h|\theta)$, and $P(c|\theta)$. For example, suppose that the hole counter receives the letter A as input. After running the algorithm over many occurrences of A in text, it may be determined that $P(h = 1|\ \theta = A) = 0.9$, which is the correct answer. With smaller probabilities, perhaps $P(h = 0|\ \theta = A) = 0.09$ and $P(h = 2|\ \theta = A) = 0.01$. Assuming that the output of each image processing algorithm is independent given the input letter, a joint probability can be assigned as

$$P(y|\theta) = P(s, e, h, c|\ \theta) = P(s|\theta)P(e|\theta)P(h|\theta)P(c|\theta). \tag{9.33}$$

The value of the prior $P(\theta)$ can be obtained by running the classifier over large amounts of hand-classified text and recording the relative numbers of occurrences of each letter. It is interesting to note that some context-specific information can be incorporated. If the text is known to be written in Spanish, then $P(\theta)$ should be different than from text written in English. Tailoring $P(\theta)$ to the type of text that will appear improves the performance of the resulting classifier.

The classifier makes its decisions by choosing the action that minimizes the probability of error. This error is proportional to

$$\sum_{\theta \in \Theta} P(s|\theta)P(e|\theta)P(h|\theta)P(c|\theta)P(\theta), \tag{9.34}$$

by neglecting the constant $P(y)$ in the denominator of Bayes' rule in (9.26). ■

Parameter estimation

Another important application of the decision-making framework of this section is *parameter estimation* [21, 70]. In this case, nature selects a *parameter*, $\theta \in \Theta$, and Θ represents a *parameter space*. Through one or more independent trials, some observations are obtained. Each observation should ideally be a direct measurement of Θ, but imperfections in the measurement process distort the observation. Usually, $\Theta \subseteq Y$, and in many cases, $Y = \Theta$. The robot action is to guess the parameter that was chosen by nature. Hence, $U = \Theta$. In most applications, all of the spaces are continuous subsets of $\mathbb{R}^n$. The cost function is designed to increase as the error, $\|u - \theta\|$, becomes larger.

Example 9.12 (Parameter Estimation) Suppose that $U = Y = \Theta = \mathbb{R}$. Nature therefore chooses a real-valued parameter, which is estimated. The cost of making a mistake is

$$L(u, \theta) = (u - \theta)^2. \tag{9.35}$$

Suppose that a Bayesian approach is taken. The prior probability density $p(\theta)$ is given as uniform over an interval $[a, b] \subset \mathbb{R}$. An observation is received, but it is noisy. The noise can be modeled as a second action of nature, as described in

Section 9.2.3. This leads to a density $p(y|\theta)$. Suppose that the noise is modeled with a Gaussian, which results in

$$p(y|\theta) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-(y-\theta)^2/2\sigma^2}, \tag{9.36}$$

in which the mean is θ and the standard deviation is σ.

The optimal parameter estimate based on y is obtained by selecting $u \in \mathbb{R}$ to minimize

$$\int_{-\infty}^{\infty} L(u,\theta)p(\theta|y)d\theta, \tag{9.37}$$

in which

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)}, \tag{9.38}$$

by Bayes' rule. The term $p(y)$ does not depend on θ, and it can therefore be ignored in the optimization. Using the prior density, $p(\theta) = 0$ outside of $[a,b]$; hence, the domain of integration can be restricted to $[a,b]$. The value of $p(\theta) = 1/(b-a)$ is also a constant that can be ignored in the optimization. Using (9.36), this means that u is selected to optimize

$$\int_a^b L(u,\theta)p(y|\theta)d\theta, \tag{9.39}$$

which can be expressed in terms of the standard error function, $\text{erf}(x)$ (the integral from 0 to a constant, of a Gaussian density over an interval).

If a sequence, y_1, $\ldots$, y_k, of independent observations is obtained, then (9.39) is replaced by

$$\int_a^b L(u,\theta)p(y_1|\theta)\cdots p(y_k|\theta)d\theta. \tag{9.40}$$

■

9.3 Two-Player Zero-Sum Games

Section 9.2 involved one real decision maker (DM), the robot, playing against a fictitious DM called nature. Now suppose that the second DM is a clever opponent that makes decisions in the same way that the robot would. This leads to a symmetric situation in which two decision makers simultaneously make a decision, without knowing how the other will act. It is assumed in this section that the DMs have diametrically opposing interests. They are two players engaged in a game in which a loss for one player is a gain for the other, and vice versa. This results in the most basic form of *game theory*, which is referred to as a *zero-sum game*.

9.3.1 Game Formulation

Suppose there are two *players*, P_1 and P_2, that each have to make a decision. Each has a finite set of actions, U and V, respectively. The set V can be viewed as the "replacement" of Θ from Formulation 9.3 by a set of actions chosen by a true opponent. Each player has a cost function, which is denoted as $L_i : U \times V \to \mathbb{R}$ for $i = 1, 2$. An important constraint for zero-sum games is

$$L_1(u,v) = -L_2(u,v), \tag{9.41}$$

which means that a cost for one player is a reward for the other. This is the basis of the term *zero sum*, which means that the two costs can be added to obtain zero. In zero-sum games the interests of the players are completely opposed. In Section 9.4 this constraint will be lifted to obtain more general games.

In light of (9.41) it is pointless to represent two cost functions. Instead, the superscript will be dropped, and L will refer to the cost, L_1, of P_1. The goal of P_1 is to minimize L. Due to (9.41), the goal of P_2 is to maximize L. Thus, L can be considered as a *reward* for P_2, but a *cost* for P_1.

A formulation can now be given:

Formulation 9.7 (A Zero-Sum Game)

1. Two players, P_1 and P_2.
2. A nonempty, finite set U called the *action space for* P_1. For convenience in describing examples, assume that U is a set of consecutive integers from 1 to $|U|$. Each $u \in U$ is referred to as an *action of* P_1.
3. A nonempty, finite set V called the *action space for* P_2. Assume that V is a set of consecutive integers from 1 to $|V|$. Each $v \in V$ is referred to as an *action of* P_2.
4. A function $L : U \times V \to \mathbb{R} \cup \{-\infty, \infty\}$ called the *cost function* for P_1. This also serves as a *reward function* for P_2 because of (9.41).

Before discussing what it means to solve a zero-sum game, some additional assumptions are needed. Assume that the players know each other's cost functions. This implies that the motivation of the opponent is completely understood. The other assumption is that the players are *rational*, which means that they will try to obtain the best cost whenever possible. P_1 will not choose an action that leads to higher cost when a lower cost action is available. Likewise, P_2 will not choose an action that leads to lower cost. Finally, it is assumed that both players make their decisions simultaneously. There is no information regarding the decision of P_1 that can be exploited by P_2, and vice versa.

Formulation 9.7 is often referred to as a *matrix game* because L can be expressed with a cost matrix, as was done in Section 9.2. Here the matrix indicates costs for P_1 and P_2, instead of the robot and nature. All of the required information from Formulation 9.7 is specified by a single matrix; therefore, it is a convenient form for expressing zero-sum games.

Example 9.13 (Matrix Representation of a Zero-Sum Game) Suppose that U, the action set for P_1, contains three actions and V contains four actions. There should be $3 \times 4 = 12$ values in the specification of the cost function, L. This can be expressed as a cost matrix,

$$\begin{array}{c|c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{4}{c}{V} \\ \cline{2-5} & 1 & 3 & 3 & 2 \\ \cline{2-5} U & 0 & -1 & 2 & 1 \\ \cline{2-5} & -2 & 2 & 0 & 1 \\ \cline{2-5} \end{array} \; , \tag{9.42}$$

in which each row corresponds to some $u \in U$, and each column corresponds to some $v \in V$. Each entry yields $L(u, v)$, which is the cost for P_1. This representation is similar to that shown in Example 9.8, except that the nature action space, Θ, is replaced by V. The cost for P_2 is $-L(u, v)$. ■

9.3.2 Deterministic Strategies

What constitutes a good solution to Formulation 9.7? Consider the game from the perspective of P_1. It seems reasonable to apply worst-case analysis when trying to account for the action that will be taken by P_2. This results in a choice that is equivalent to assuming that P_2 is nature acting under the nondeterministic model, as considered in Section 9.2.2. For a matrix game, this is computed by first determining the maximum cost over each row. Selecting the action that produces the minimum among these represents the lowest cost that P_1 can guarantee for itself. Let this selection be referred to as a *security strategy* for P_1.

For the matrix game in (9.42), the security strategy is illustrated as

$$\begin{array}{c|cccc|c|} \multicolumn{1}{c}{} & \multicolumn{4}{c}{V} & \multicolumn{1}{c}{} \\ \cline{2-6} & 1 & 3 & 3 & 2 & \rightarrow 3 \\ \cline{2-6} U & 0 & -1 & 2 & 1 & \rightarrow \mathbf{2} \\ \cline{2-6} & -2 & 2 & 0 & 1 & \rightarrow \mathbf{2} \\ \cline{2-6} \end{array} \; , \tag{9.43}$$

in which $u = 2$ and $u = 3$ are the best actions. Each yields a cost no worse than 2, regardless of the action chosen by P_2.

This can be formalized using the existing notation. A security strategy, u^*, for P_1 is defined in general as

$$u^* = \underset{u \in U}{\operatorname{argmin}} \Big\{ \max_{v \in V} \Big\{ L(u, v) \Big\} \Big\}. \tag{9.44}$$

There may be multiple security strategies that satisfy the argmin; however, this does not cause trouble, as will be explained shortly. Let the resulting worst-case cost be denoted by $\overline{L}^*$, and let it be called the *upper value* of the game. This is defined as

$$\overline{L}^* = \max_{v \in V} \Big\{ L(u^*, v) \Big\}. \tag{9.45}$$

Now swap roles, and consider the game from the perspective of P_2, which would like to maximize L. It can also use worst-case analysis, which means that it would like to select an action that guarantees a high cost, in spite of the action of P_1 to potentially reduce it. A security strategy, v^*, for P_2 is defined as

$$v^* = \underset{v \in V}{\operatorname{argmax}} \Big\{ \min_{u \in U} \Big\{ L(u, v) \Big\} \Big\}. \tag{9.46}$$

Note the symmetry with respect to (9.44). There may be multiple security strategies for P_2. A security strategy v^* is just an "upside-down" version of the worst-case analysis applied in Section 9.2.2. The *lower value*, $\underline{L}^*$, is defined as

$$\underline{L}^* = \min_{u \in U} \Big\{ L(u, v^*) \Big\}. \tag{9.47}$$

Returning to the matrix game in (9.42), the last column is selected by applying (9.46):

$$\begin{array}{c|c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{4}{c}{V} \\ \cline{2-5} & 1 & 3 & 3 & 2 \\ & 0 & -1 & 2 & 1 \\ U & -2 & 2 & 0 & 1 \\ \cline{2-5} & \downarrow & \downarrow & \downarrow & \downarrow \\ & -2 & -1 & 0 & \mathbf{1} \end{array} \; . \tag{9.48}$$

An interesting relationship between the upper and lower values is that $\underline{L}^* \leq \overline{L}^*$ for any game using Formulation 9.7. This is shown by observing that

$$\underline{L}^* = \min_{u \in U} \Big\{ L(u, v^*) \Big\} \leq L(u^*, v^*) \leq \max_{v \in V} \Big\{ L(u^*, v) \Big\} = \overline{L}^*, \tag{9.49}$$

in which $L(u^*, v^*)$ is the cost received when the players apply their respective security strategies. If the game is played by rational DMs, then the resulting cost always lies between $\underline{L}^*$ and $\overline{L}^*$.

Regret Suppose that the players apply security strategies, $u^* = 2$ and $v^* = 4$. This results in a cost of $L(2, 4) = 1$. How do the players feel after the outcome? P_1 may feel satisfied because given that P_2 selected $v^* = 4$, it received the lowest cost possible. On the other hand, P_2 may regret its decision in light of the action chosen by P_1. If it had known that $u = 2$ would be chosen, then it could have picked $v = 2$ to receive cost $L(2, 2) = 2$, which is better than $L(2, 4) = 1$. If the

	$\geq$	
$\leq$	L^*	$\leq$
	$\geq$	

Figure 9.2: A saddle point can be detected in a matrix by finding a value L^* that is lowest among all elements in its column and greatest among all elements in its row.

game were to be repeated, then P$_2$ would want to change its strategy in hopes of tricking P$_1$ to obtain a higher reward.

Is there a way to keep both players satisfied? Any time there is a gap between $\underline{L}^*$ and $\overline{L}^*$, there is regret for one or both players. If r_1 and r_2 denote the amount of regret experienced by P$_1$ and P$_2$, respectively, then the total regret is

$$r_1 + r_2 = \overline{L}^* - \underline{L}^*. \tag{9.50}$$

Thus, the only way to satisfy both players is to obtain upper and lower values such that $\underline{L}^* = \overline{L}^*$. These are properties of the game, however, and they are not up to the players to decide. For some games, the values are equal, but for many $\underline{L}^* < \overline{L}^*$. Fortunately, by using randomized strategies, the upper and lower values always coincide; this is covered in Section 9.3.3.

Saddle points If $\underline{L}^* = \overline{L}^*$, the security strategies are called a *saddle point*, and $L^* = \underline{L}^* = \overline{L}^*$ is called the *value* of the game. If this occurs, the order of the max and min can be swapped without changing the value:

$$L^* = \min_{u \in U} \Big\{ \max_{v \in V} \Big\{ L(u,v) \Big\} \Big\} = \max_{v \in V} \Big\{ \min_{u \in U} \Big\{ L(u,v) \Big\} \Big\}. \tag{9.51}$$

A saddle point is sometimes referred to as an *equilibrium* because the players have no incentive to change their choices (because there is no regret). A saddle point is defined as any $u^* \in U$ and $v^* \in V$ such that

$$L(u^*, v) \leq L(u^*, v^*) \leq L(u, v^*) \tag{9.52}$$

for all $u \in U$ and $v \in V$. Note that $L^* = L(u^*, v^*)$. When looking at a matrix game, a saddle point is found by finding the simple pattern shown in Figure 9.2.

	$\geq$		$\geq$	
$\leq$	L^*	$\leq$	L^*	$\leq$
	$\geq$		$\geq$	
$\leq$	L^*	$\leq$	L^*	$\leq$
	$\geq$		$\geq$	

Figure 9.3: A matrix could have more than one saddle point, which may seem to lead to a coordination problem between the players. Fortunately, there is no problem, because the same value will be received regardless of which saddle point is selected by each player.

Example 9.14 (A Deterministic Saddle Point) Here is a matrix game that has a saddle point:

$$\begin{array}{c|ccc|} & & V & \\ \hline & 3 & 3 & 5 \\ U & 1 & -1 & 7 \\ & 0 & -2 & 4 \\ \hline \end{array} . \tag{9.53}$$

By applying (9.52) (or using Figure 9.2), the saddle point is obtained when $u = 3$ and $v = 3$. The result is that $L^* = 4$. In this case, neither player has regret after the game is finished. P$_1$ is satisfied because 4 is the lowest cost it could have received, given that P$_2$ chose the third column. Likewise, 4 is the highest cost that P$_2$ could have received, given that P$_1$ chose the bottom row. ■

What if there are multiple saddle points in the same game? This may appear to be a problem because the players have no way to coordinate their decisions. What if P$_1$ tries to achieve one saddle point while P$_2$ tries to achieve another? It turns out that if there is more than one saddle point, then there must at least be four, as shown in Figure 9.3. As soon as we try to make two "+" patterns like the one shown in Figure 9.2, they intersect, and four saddle points are created. Similar behavior occurs as more saddle points are added.

Example 9.15 (Multiple Saddle Points) This game has multiple saddle points and follows the pattern in Figure 9.3:

$$\begin{array}{c|ccccc|} & & & V & & \\ \hline & 4 & 3 & 5 & 1 & 2 \\ & -1 & 0 & -2 & 0 & -1 \\ U & -4 & 1 & 4 & 3 & 5 \\ & -3 & 0 & -1 & 0 & -2 \\ & 3 & 2 & -7 & 3 & 8 \\ \hline \end{array} . \tag{9.54}$$

Let (i,j) denote the pair of choices for P$_1$ and P$_2$, respectively. Both $(2,2)$ and $(4,4)$ are saddle points with value $V = 0$. What if P$_1$ chooses $u = 2$ and P$_2$ chooses

$v = 4$? This is not a problem because $(2, 4)$ is also a saddle point. Likewise, $(4, 2)$ is another saddle point. In general, no problems are caused by the existence of multiple saddle points because the resulting cost is independent of which saddle point is attempted by each player. ■

9.3.3 Randomized Strategies

The fact that some zero-sum games do not have a saddle point is disappointing because regret is unavoidable in these cases. Suppose we slightly change the rules. Assume that the same game is repeatedly played by P_1 and P_2 over numerous trials. If they use a deterministic strategy, they will choose the same actions every time, resulting in the same costs. They may instead switch between alternative security strategies, which causes fluctuations in the costs. What happens if they each implement a randomized strategy? Using the idea from Section 9.1.3, each strategy is specified as a probability distribution over the actions. In the limit, as the number of times the game is played tends to infinity, an expected cost is obtained. One of the most famous results in game theory is that on the space of randomized strategies, a saddle point always exists for any zero-sum matrix game; however, expected costs must be used. Thus, if randomization is used, there will be no regrets. In an individual trial, regret may be possible; however, as the costs are averaged over all trials, both players will be satisfied.

Extending the formulation

Since a game under Formulation 9.7 can be nicely expressed as a matrix, it is tempting to use linear algebra to conveniently express expected costs. Let $|U| = m$ and $|V| = n$. As in Section 9.1.3, a randomized strategy for P_1 can be represented as an m-dimensional vector,

$$w = [w_1 \;\; w_2 \;\; \ldots \;\; w_m]. \tag{9.55}$$

The probability axioms of Section 9.1.2 must be satisfied: 1) $w_i \geq 0$ for all $i \in \{1, \ldots, m\}$, and 2) $w_1 + \cdots + w_m = 1$. If w is considered as a point in $\mathbb{R}^m$, then the two constraints imply that it must lie on an $(m-1)$-dimensional simplex (recall Section 6.3.1). If $m = 3$, this means that w lies in a triangular subset of $\mathbb{R}^3$. Similarly, let z represent a randomized strategy for P_2 as an n-dimensional vector,

$$z = [z_1 \;\; z_2 \;\; \ldots \;\; z_n]^T, \tag{9.56}$$

that also satisfies the probability axioms. In (9.56), ^{T} denotes *transpose*, which yields a column vector that satisfies the dimensional constraints required for an upcoming matrix multiplication.

Let $\bar{L}(w, z)$ denote the expected cost that will be received if P_1 plays w and P_2 plays z. This can be computed as

$$\bar{L}(w, z) = \sum_{i=1}^{m} \sum_{j=1}^{n} L(i, j) w_i z_j. \tag{9.57}$$

Note that the cost, $L(i, j)$, makes use of the assumption in Formulation 9.7 that the actions are consecutive integers. The expected cost can be alternatively expressed using the cost matrix, A. In this case

$$\bar{L}(w, z) = wAz, \tag{9.58}$$

in which the product wAz yields a scalar value that is precisely (9.57). To see this, first consider the product Az. This yields an m-dimensional vector in which the ith element is the expected cost that P_1 would receive if it tries $u = i$. Thus, it appears that P_1 views P_2 as a nature player under the probabilistic model. Once w and Az are multiplied, a scalar value is obtained, which averages the costs in the vector Az according the probabilities of w.

Let W and Z denote the set of all randomized strategies for P_1 and P_2, respectively. These spaces include strategies that are equivalent to the deterministic strategies considered in Section 9.3.2 by assigning probability one to a single action. Thus, W and Z can be considered as expansions of the set of possible strategies in comparison to what was available in the deterministic setting. Using W and Z, *randomized security strategies* for P_1 and P_2 are defined as

$$w^* = \underset{w \in W}{\operatorname{argmin}} \Big\{ \max_{z \in Z} \Big\{ \bar{L}(w, z) \Big\} \Big\} \tag{9.59}$$

and

$$z^* = \underset{z \in Z}{\operatorname{argmax}} \Big\{ \min_{w \in W} \Big\{ \bar{L}(w, z) \Big\} \Big\}, \tag{9.60}$$

respectively. These should be compared to (9.44) and (9.46). The differences are that the space of strategies has been expanded, and expected cost is now used.

The *randomized upper value* is defined as

$$\overline{\mathcal{L}}^* = \max_{z \in Z} \Big\{ \bar{L}(w^*, z) \Big\}, \tag{9.61}$$

and the *randomized lower value* is

$$\underline{\mathcal{L}}^* = \min_{w \in W} \Big\{ \bar{L}(w, z^*) \Big\}. \tag{9.62}$$

Since W and Z include the deterministic security strategies, $\overline{\mathcal{L}}^* \leq \overline{L}^*$ and $\underline{\mathcal{L}}^* \geq \underline{L}^*$. These inequalities imply that the randomized security strategies may have some hope in closing the gap between the two values in general.

The most fundamental result in zero-sum game theory was shown by von Neumann [284, 285], and it states that $\underline{\mathcal{L}}^* = \overline{\mathcal{L}}^*$ for any game in Formulation 9.7. This yields the *randomized value* $\mathcal{L}^* = \underline{\mathcal{L}}^* = \overline{\mathcal{L}}^*$ for the game. This means that there will never be expected regret if the players stay with their security strategies. If the players apply their randomized security strategies, then a *randomized saddle point* is obtained. This saddle point cannot be seen as a simple pattern in the matrix A because it instead exists over W and Z.

The guaranteed existence of a randomized saddle point is an important result because it demonstrates the value of randomization when making decisions against an intelligent opponent. In Example 9.7, it was intuitively argued that randomization seems to help when playing against an intelligent adversary. When playing the game repeatedly with a deterministic strategy, the other player could learn the strategy and win every time. Once a randomized strategy is used, the players will not experience regret.

Computation of randomized saddle points

So far it has been established that a randomized saddle point always exists, but how can one be found? Two key observations enable a combinatorial solution to the problem:

1. The security strategy for each player can be found by considering only deterministic strategies for the opposing player.

2. If the strategy for the other player is fixed, then the expected cost is a linear function of the undetermined probabilities.

First consider the problem of determining the security strategy for P_1. The first observation means that (9.59) does not need to consider randomized strategies for P_2. Inside of the argmin, w is fixed. What randomized strategy, $z \in Z$, maximizes $\bar{L}(w,z) = wAz$? If w is fixed, then wA can be treated as a constant n-dimensional vector, s. This means $\bar{L}(w,z) = s \cdot z$, in which $\cdot$ is the inner (dot) product. Now the task is to select z to maximize $s \cdot z$. This involves selecting the largest element of s; suppose this is s_i. The maximum cost over all $z \in Z$ is obtained by placing all of the probability mass at action i. Thus, the strategy $z_i = 1$ and $z_j = 0$ for $i \neq j$ gives the highest cost, and it is deterministic.

Using the first observation, for each $w \in W$, only n possible responses by P_2 need to be considered. These are the n deterministic strategies, each of which assigns $z_i = 1$ for a unique $i \in \{1, \ldots, n\}$.

Now consider the second observation. The expected cost, $\bar{L}(w,z) = wAz$, is a linear function of w, if z is fixed. Since z only needs to be fixed at n different values due to the first observation, w is selected at the point at which the smallest maximum value among the n linear functions occurs. This is the minimum value of the *upper envelope* of the collection of linear functions. Such envelopes were mentioned in Section 6.5.2. Example 9.16 will illustrate this. The domain for this optimization can conveniently be set as a triangle in $\mathbb{R}^{m-1}$. Even though $W \subset \mathbb{R}^m$, the last coordinate, w_m, is not needed because it is always $w_m = 1-(w_1 + \cdots + w_{m-1})$. The resulting optimization falls under *linear programming*, for which many combinatorial algorithms exist [65, 68, 178, 201].

In the explanation above, there is nothing particular to P_1 when trying to find its security strategy. The same method can be applied to determine the security strategy for P_2; however, every minimization is replaced by a maximization, and vice versa. In summary, the min in (9.60) needs only to consider the deterministic strategies in W. If w becomes fixed, then $\bar{L}(w,z) = wAz$ is once again a linear function, but this time it is linear in z. The best randomized action is chosen by finding the point $z \in Z$ that gives the highest minimum value among m linear functions. This is the minimum value of the *lower envelope* of the collection of linear functions. The optimization occurs over $\mathbb{R}^{n-1}$ because the last coordinate, z_n, is obtained directly from $z_n = 1 - (z_1 + \cdots + z_{n-1})$.

This computation method is best understood through an example.

Example 9.16 (Computing a Randomized Saddle Point) The simplest case is when both players have only two actions. Let the cost matrix be defined as

$$\begin{array}{c|c|c|} & \multicolumn{2}{c}{V} \\ \cline{2-3} U & 3 & 0 \\ \cline{2-3} & -1 & 1 \\ \cline{2-3} \end{array} \; . \tag{9.63}$$

Consider computing the security strategy for P_1. Note that W and Z are only one-dimensional subsets of $\mathbb{R}^2$. A randomized strategy for P_1 is $w = [w_1 \;\; w_2]$, with $w_1 \geq 0$, $w_2 \geq 0$, and $w_1 + w_2 = 1$. Therefore, the domain over which the optimization is performed is $w_1 \in [0,1]$ because w_2 can always be derived as $w_2 = 1 - w_1$. Using the first observation above, only the two deterministic strategies for P_2 need to be considered. When considered as linear functions of w, these are

$$(3)w_1 + (-1)(1-w_1) = 4w_1 - 1 \tag{9.64}$$

for $z_1 = 1$ and

$$(0)w_1 + (1)(1-w_1) = 1 - w_1 \tag{9.65}$$

for $z_2 = 1$. The lines are plotted in Figure 9.4a. The security strategy is determined by the minimum point along the upper envelope shown in the figure. This is indicated by the thickened line, and it is always a piecewise-linear function in general. The lowest point occurs at $w_1 = 2/5$, and the resulting value is $\mathcal{L}^* = 3/5$. Therefore, $w^* = [2/5 \;\; 3/5]$.

A similar procedure can be used to obtain z^*. The lines that correspond to the deterministic strategies of P_1 are shown in Figure 9.4b. The security strategy is obtained by finding the maximum value along the lower envelope of the lines, which is shown as the thickened line in the figure. This results in $z^* = [1/5 \;\; 4/5]^T$, and once again, the value is observed as $\mathcal{L}^* = 3/5$ (this must coincide with the

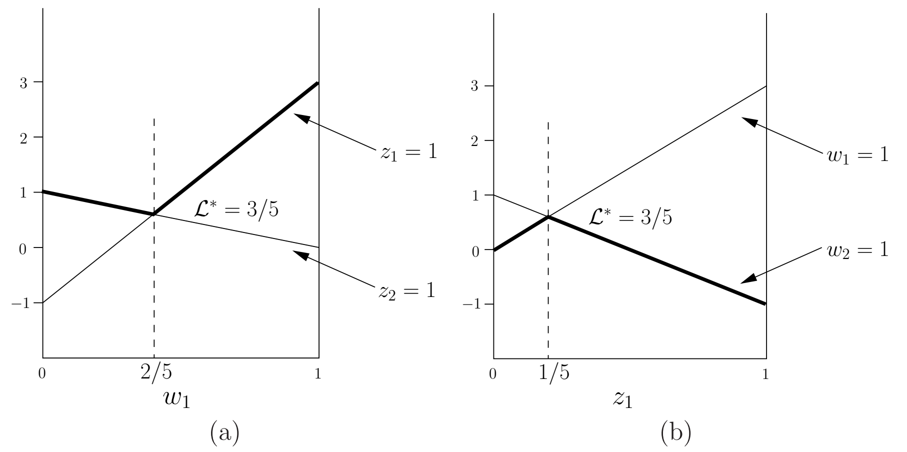


Figure 9.4: (a) Computing the randomized security strategy, w^*, for P$_1$. (b) Computing the randomized security strategy, z^*, for P$_2$.

previous one because the randomized upper and lower values are the same!). ■

This procedure appears quite simple if there are only two actions per player. If $n = m = 100$, then the upper and lower envelopes are piecewise-linear functions in $\mathbb{R}^{99}$. This may be computationally impractical because all existing linear programming algorithms have running time at least exponential in dimension [68].

9.4 Nonzero-Sum Games

This section parallels the development of Section 9.3, except that the more general case of nonzero-sum games is considered. This enables games with any desired degree of conflict to be modeled. Some decisions may even benefit all players. One of the main applications of the subject is in economics, where it helps to explain the behavior of businesses in competition.

The saddle-point solution will be replaced by the *Nash equilibrium*, which again is based on eliminating regret. Since the players do not necessarily oppose each other, it is possible to model a game that involves any number of players. For nonzero games, new difficulties arise, such as the nonuniqueness of Nash equilibria and the computation of randomized Nash equilibria does not generally fit into linear programming.

9.4.1 Two-Player Games

To help make the connection to Section 9.3 smoother, two-player games will be considered first. This case is also easier to understand because the notation is simpler. The ideas are then extended without difficulty from two players to many players. The game is formulated as follows.

Formulation 9.8 (A Two-Player Nonzero-Sum Game)

1. The same components as in Formulation 9.7, except the cost function.
2. A function, $L_1 : U \times V \to \mathbb{R} \cup \{\infty\}$, called the *cost function for* P$_1$.
3. A function, $L_2 : U \times V \to \mathbb{R} \cup \{\infty\}$, called the *cost function for* P$_2$.

The only difference with respect to Formulation 9.7 is that now there are two, independent cost functions, L_1 and L_2, one for each player. Each player would like to minimize its cost. There is no maximization in this problem; that appeared in zero-sum games because P$_2$ had opposing interests from P$_1$. A zero-sum game can be modeled under Formulation 9.7 by setting $L_1 = L$ and $L_2 = -L$.

Paralleling Section 9.3, first consider applying deterministic strategies to solve the game. As before, one possibility is that a player can apply its security strategy. To accomplish this, it does not even need to look at the cost function of the other player. It seems somewhat inappropriate, however, to neglect the consideration of both cost functions when making a decision. In most cases, the security strategy results in regret, which makes it inappropriate for nonzero-sum games.

A strategy that avoids regret will now be given. A pair (u^*, v^*) of actions is defined to be a *Nash equilibrium* if

$$L_1(u^*, v^*) = \min_{u \in U} \Big\{ L_1(u, v^*) \Big\} \tag{9.66}$$

and

$$L_2(u^*, v^*) = \min_{v \in V} \Big\{ L_2(u^*, v) \Big\}. \tag{9.67}$$

These expressions imply that neither P$_1$ nor P$_2$ has regret. Equation (9.66) indicates that P$_1$ is satisfied with its action, u^*, given the action, v^*, chosen by P$_2$. P$_1$ cannot reduce its cost any further by changing its action. Likewise, (9.67) indicates that P$_2$ is satisfied with its action v^*.

The game in Formulation 9.8 can be completely represented using two cost matrices. Let A and B denote the cost matrices for P$_1$ and P$_2$, respectively. Recall that Figure 9.2 showed a pattern for detecting a saddle point. A Nash equilibrium can be detected as shown in Figure 9.5. Think about the relationship between the two. If $A = -B$, then B can be negated and superimposed on top of A. This will yield the pattern in Figure 9.2 (each $\geq$ becomes $\leq$ because of negation). The values L_a^* and L_b^* coincide in this case. This observation implies that if $A = -B$, then the Nash equilibrium is actually the same concept as a saddle point. It applies, however, to much more general games.

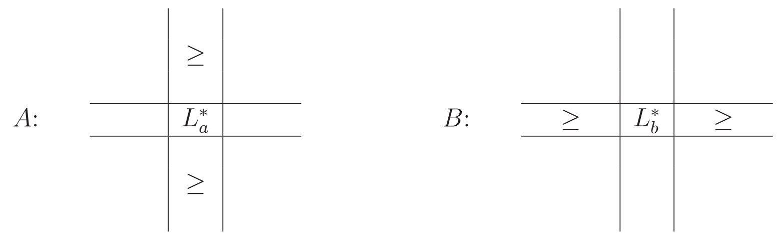


Figure 9.5: A Nash equilibrium can be detected in a pair of matrices by finding some (i, j) such that $L_a^* = L_1(i, j)$ is the lowest among all elements in column j of A, and $L_b^* = L_2(i, j)$ is the lowest among all elements in row i of B. Compare this with Figure 9.2.

Example 9.17 (A Deterministic Nash Equlibrium) Consider the game specified by the cost matrices A and B:

$$A: \quad \begin{array}{c|ccc} & & V & \\ \hline & 9 & 4 & 7 \\ U & 6 & -1 & 5 \\ & 1 & 4 & 2 \end{array} \qquad B: \quad \begin{array}{c|ccc} & & V & \\ \hline & 2 & 1 & 6 \\ U & 5 & 0 & 2 \\ & 2 & 2 & 5 \end{array} \; . \tag{9.68}$$

By applying (9.66) and (9.67), or by using the patterns in Figure 9.5, it can be seen that $u = 3$ and $v = 1$ is a Nash equilibrium. The resulting costs are $L_1 = 1$ and $L_2 = 2$. Another Nash equilibrium appears at $u = 2$ and $v = 2$. This yields costs $L_1 = -1$ and $L_2 = 0$, which is better for both players.

For zero-sum games, the existence of multiple saddle points did not cause any problem; however, for nonzero-sum games, there are great troubles. In the example shown here, one Nash equilibrium is clearly better than the other for both players. Therefore, it may seem reasonable that a rational DM would choose the better one. The issue of multiple Nash equilibria will be discussed next. ■

Dealing with multiple Nash equilibria

Example 9.17 was somewhat disheartening due to the existence of multiple Nash equilibria. In general, there could be any number of equilibria. How can each player know which one to play? If they each choose a different one, they are not guaranteed to fall into another equilibrium as in the case of saddle points of zero-sum games. Many of the equilibria can be eliminated by using Pareto optimality, which was explained in Section 9.1.1 and also appeared in Section 7.7.2 as a way to optimally coordinate multiple robots. The idea is to formulate the selection as a multi-objective optimization problem, which fits into Formulation 9.2.

Consider two-dimensional vectors of the form (x_i, y_i), in which x and y represent the costs L_1 and L_2 obtained under the implementation of a Nash equilibrium denoted by π_i. For two different equilibria π_1 and π_2, the cost vectors (x_1, y_1) and (x_2, y_2) are obtained. In Example 9.17, these were $(1, 2)$ and $(-1, 0)$. In general, π_1 is said to be *better* than π_2 if $x_1 \leq x_2$, $y_1 \leq y_2$, and at least one of the inequalities is strict. In Example 9.17, the equilibrium that produces $(-1, 0)$ is clearly better than obtaining $(1, 2)$ because both players benefit.

The definition of "better" induces a partial ordering on the space of Nash equilibria. It is only partial because some vectors are incomparable. Consider, for example, $(-1, 1)$ and $(1, -1)$. The first one is preferable to P_1, and the second is preferred by P_2. In game theory, the Nash equilibria that are minimal with respect to this partial ordering are called *admissible*. They could alternatively be called *Pareto optimal*.

The best situation is when a game has one Nash equilibrium. If there are multiple Nash equilibria, then there is some hope that only one of them is admissible. In this case, it is hoped that the rational players are intelligent enough to figure out that any nonadmissible equilibria should be discarded. Unfortunately, there are many games that have multiple admissible Nash equilibria. In this case, analysis of the game indicates that the players must communicate or collaborate in some way to eliminate the possibility of regret. Otherwise, regret is unavoidable in the worst case. It is also possible that there are no Nash equilibria, but, fortunately, by allowing randomized strategies, a randomized Nash equilibrium is always guaranteed to exist. This will be covered after the following two examples.

Example 9.18 (The Battle of the Sexes) Consider a game specified by the cost matrices A and B:

$$A: \quad \begin{array}{c|cc} & \multicolumn{2}{c}{V} \\ \hline U & -2 & 0 \\ & 0 & -1 \end{array} \qquad B: \quad \begin{array}{c|cc} & \multicolumn{2}{c}{V} \\ \hline U & -1 & 0 \\ & 0 & -2 \end{array} \; . \tag{9.69}$$

This is a famous game called the "Battle of the Sexes." Suppose that a man and a woman have a relationship, and they each have different preferences on how to spend the evening. The man prefers to go shopping, and the woman prefers to watch a football match. The game involves selecting one of these two activities. The best case for either one is to do what they prefer while still remaining together. The worst case is to select different activities, which separates the couple. This game is somewhat unrealistic because in most situations some cooperation between them is expected.

Both $u = v = 1$ and $u = v = 2$ are Nash equilibria, which yield cost vectors $(-2, -1)$ and $(-1, -2)$, respectively. Neither solution is better than the other; therefore, they are both admissible. There is no way to avoid the possibility of regret unless the players cooperate (you probably already knew this). ■

The following is one of the most famous nonzero-sum games.

Example 9.19 (The Prisoner's Dilemma) The following game is very simple to express, yet it illustrates many interesting issues. Imagine that a heinous crime has been committed by two people. The authorities know they are guilty, but they do not have enough evidence to convict them. Therefore, they develop a plan to try to trick the suspects. Each suspect (or player) is placed in an isolated prison cell and given two choices. Each player can cooperate with the authorities, $u = 1$ or $v = 1$, or refuse, $u = 2$ or $v = 2$. By cooperating, the player admits guilt and turns over evidence to the authorities. By refusing, the player claims innocence and refuses to help the authorities.

The cost L_i represents the number of years that the player will be sentenced to prison. The cost matrices are assigned as

$$A : \quad \begin{array}{cc} & V \\ U & \begin{array}{|c|c|}\hline 8 & 0 \\ \hline 30 & 2 \\ \hline \end{array} \end{array} \qquad B : \quad \begin{array}{cc} & V \\ U & \begin{array}{|c|c|}\hline 8 & 30 \\ \hline 0 & 2 \\ \hline \end{array} \end{array} . \tag{9.70}$$

The motivation is that both players receive 8 years if they both cooperate, which is the sentence for being convicted of the crime and being rewarded for cooperating with the authorities. If they both refuse, then they receive 2 years because the authorities have insufficient evidence for a longer term. The interesting cases occur if one refuses and the other cooperates. The one who refuses is in big trouble because the evidence provided by the other will be used against him. The one who cooperates gets to go free (the cost is 0); however, the other is convicted on the evidence and spends 30 years in prison.

What should the players do? What would you do? If they could make a coordinated decision, then it seems that a good choice would be for both to refuse, which results in costs $(2, 2)$. In this case, however, there would be regret because each player would think that he had a chance to go free (receiving cost 0 by refusing). If they were to play the game a second time, they might be inclined to change their decisions.

The Nash equilibrium for this problem is for both of them to cooperate, which results in $(8, 8)$. Thus, they pay a price for not being able to communicate and coordinate their strategy. This solution is also a security strategy for the players, because it achieves the lowest cost using worst-case analysis. ■

Randomized Nash equilibria

What happens if a game has no Nash equilibrium over the space of deterministic strategies? Once again the problem can be alleviated by expanding the strategy space to include randomized strategies. In Section 9.3.3 it was explained that every zero-sum game under Formulation 9.7 has a randomized saddle point on the space of randomized strategies. It was shown by Nash that every nonzero-sum game under Formulation 9.8 has a randomized Nash equilibrium [200]. This is a nice result; however, there are a couple of concerns. There may still exist other admissible equilibria, which means that there is no reliable way to avoid regret unless the players collaborate. The other concern is that randomized Nash equilibria unfortunately cannot be computed using the linear programming approach of Section 9.3.3. The required optimization is instead a form of nonlinear programming [24, 178, 201], which does not necessarily admit a nice, combinatorial solution.

Recall the definition of randomized strategies from Section 9.3.3. For a pair (w, z) of randomized strategies, the expected costs, $\bar{L}^1(w, z)$ and $\bar{L}^2(w, z)$, can be computed using (9.57). A pair (w^*, z^*) of strategies is said to be a *randomized Nash equilibrium* if

$$\bar{L}^1(w^*, z^*) = \min_{w \in W} \Big\{ \bar{L}^1(w, z^*) \Big\} \tag{9.71}$$

and

$$\bar{L}^2(w^*, z^*) = \min_{z \in Z} \Big\{ \bar{L}^2(w^*, z) \Big\}. \tag{9.72}$$

In game-theory literature, this is usually referred to as a *mixed Nash equilibrium*. Note that (9.71) and (9.72) are just generalizations of (9.66) and (9.67) from the space of deterministic strategies to the space of randomized strategies.

Using the cost matrices A and B, the Nash equilibrium conditions can be written as

$$w^* A z^* = \min_{w \in W} \Big\{ w A z^* \Big\} \tag{9.73}$$

and

$$w^* B z^* = \min_{z \in Z} \Big\{ w^* B z \Big\}. \tag{9.74}$$

Unfortunately, the computation of randomized Nash equilibria is considerably more challenging than computing saddle points. The main difficulty is that Nash equilibria are not necessarily security strategies. By using security strategies, it is possible to decouple the decisions of the players into separate linear programming problems, as was seen in Example 9.16. For the randomized Nash equilibrium, the optimization between the players remains coupled. The resulting optimization is often referred to as the *linear complementarity problem.* This can be formulated as a nonlinear programming problem [178, 201], which means that it is a nonlinear optimization that involves both equality and inequality constraints on the domain (in this particular case, a *bilinear programming* problem is obtained [9]).

Example 9.20 (Finding a Randomized Nash Equilibrium) To get an idea of the kind of optimization that is required, recall Example 9.18. A randomized Nash equilibrium that is distinct from the two deterministic equilibria can be found. Using the cost matrices from Example 9.18, the expected cost for P_1 given randomized strategies w and z is

$$\begin{aligned}\bar{L}^1(w,z) &= wAz \\ &= \begin{pmatrix} w_1 & w_2 \end{pmatrix} \begin{pmatrix} -2 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \\ &= -2w_1z_1 - w_2z_2 \\ &= -3w_1z_1 + w_1 + z_1, \end{aligned} \tag{9.75}$$

in which the final step uses the fact that $w_2 = 1 - w_1$ and $z_2 = 1 - z_1$. Similarly, the expected cost for P_2 is

$$\begin{aligned}\bar{L}^2(w,z) &= wBz \\ &= \begin{pmatrix} w_1 & w_2 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & -2 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \\ &= -w_1z_1 - 2w_2z_2 \\ &= -3w_1z_1 + 2w_1 + 2z_1. \end{aligned} \tag{9.76}$$

If z is fixed, then the final equation in (9.75) is linear in w; likewise, if w is fixed, then (9.76) is linear in z. In the case of computing saddle points for zero-sum games, we were allowed to make this assumption; however, it is not possible here. We must choose (w^*, z^*) to simultaneously optimize (9.75) while $z = z^*$ and (9.76) while $w = w^*$.

It turns out that this problem is simple enough to solve with calculus. Using the classical optimization method of taking derivatives, a candidate solution can be found by computing

$$\frac{\partial \bar{L}^1(w_1, z_1)}{\partial w_1} = 1 - 3z_1 \tag{9.77}$$

and

$$\frac{\partial \bar{L}^2(w_1, z_1)}{\partial z_1} = 2 - 3w_1. \tag{9.78}$$

Extrema occur when both of these simultaneously become 0. Solving $1 - 3z_1 = 0$ and $2 - 3w_1 = 0$ yields $(w^*, z^*) = (2/3, 1/3)$, which is a randomized Nash equilibrium. The deterministic Nash equilibria are not detected by this method because they occur on the boundary of W and Z, where the derivative is not defined. ■

The computation method in Example 9.20 did not appear too difficult because there were only two actions per player, and half of the matrix costs were 0. In general, two complicated equations must be solved simultaneously. The expressions, however, are always second-degree polynomials. Furthermore, they each become linear with respect to the other variables if w or z is held fixed.

Summary of possible solutions The solution possibilities to remember for a nonzero-sum game under Formulation 9.8 are as follows.

1. There may be multiple, admissible (deterministic) Nash equilibria.
2. There may be no (deterministic) Nash equilibria.
3. There is always at least one randomized Nash equilibrium.

9.4.2 More Than Two Players

The ideas of Section 9.4.1 easily generalize to any number of players. The main difficulty is that complicated notation makes the concepts appear more difficult. Keep in mind, however, that there are no fundamental differences. A nonzero-sum game with n players is formulated as follows.

Formulation 9.9 (An n-Player Nonzero-Sum Game)

1. A set of n players, P_1, P_2, …, P_n.
2. For each player P_i, a finite, nonempty set U^i called the *action space* for P_i. For convenience, assume that each U^i is a set of consecutive integers from 1 to $|U^i|$. Each $u^i \in U^i$ is referred to as an *action* of P_i.
3. For each player P_i, a function, $L_i : U^1 \times U^2 \times \cdots \times U^n \to \mathbb{R} \cup \{\infty\}$ called the *cost function for* P_i.

A matrix formulation of the costs is no longer possible because there are too many dimensions. For example, if $n = 3$ and $|U^i| = 2$ for each player, then $L_i(u^1, u^2, u^3)$ is specified by a $2 \times 2 \times 2$ cube of 8 entries. Three of these cubes are needed to specify the game. Thus, it may be helpful to just think of L_i as a multivariate function and avoid using matrices.[4]

The Nash equilibrium idea generalizes by requiring that each P_i experiences no regret, given the actions chosen by the other $n-1$ players. Formally, a set $(u^{1*}, \ldots, u^{n*})$ of actions is said to be a (deterministic) *Nash equilibrium* if

$$L_i(u^{1*}, \ldots, u^{i*}, \ldots, u^{n*}) = \min_{u^i \in U^i} \left\{ L_i(u^{1*}, \ldots, u^{(i-1)*}, u^i, u^{(i+1)*}, \ldots, u^{n*}) \right\} \tag{9.79}$$

for every $i \in \{1, \ldots, n\}$.

For $n > 2$, any of the situations summarized at the end of Section 9.4.1 can occur. There may be no deterministic Nash equilibria or multiple Nash equilibria. The definition of an admissible Nash equilibrium is extended by defining the notion of *better* over n-dimensional cost vectors. Once again, the minimal vectors

[4]If you enjoy working with tensors, these could be used to capture n-player cost functions [30].

with respect to the resulting partial ordering are considered *admissible* (or *Pareto optimal*). Unfortunately, multiple admissible Nash equilibria may still exist.

It turns out that for any game under Formulation 9.9, there exists a randomized Nash equilibrium. Let z^i denote a randomized strategy for P_i. The expected cost for each P_i can be expressed as

$$\bar{L}^i(z^1, z^2, \ldots, z^n) = \sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_n=1}^{m_n} L_i(i_1, i_2, \ldots, i_n) z^1_{i_1} z^2_{i_2} \cdots z^n_{i_n}. \tag{9.80}$$

Let Z^i denote the space of randomized strategies for P_i. An assignment, $(z^{1*}, \ldots, z^{n*})$, of randomized strategies to all of the players is called a *randomized Nash equilibrium* if

$$\bar{L}^i(z^{1*}, \ldots, z^{i*}, \ldots, z^{n*}) = \min_{z^i \in Z^i} \Big\{ \bar{L}^i(z^{1*}, \ldots, z^{(i-1)*}, z^i, z^{(i+1)*}, \ldots, z^{n*}) \Big\} \tag{9.81}$$

for all $i \in \{1, \ldots, n\}$.

As might be expected, computing a randomized Nash equilibrium for $n > 2$ is even more challenging than for $n = 2$. The method of Example 9.20 can be generalized to n-player games; however, the expressions become even more complicated. There are n equations, each of which appears linear if the randomized strategies are fixed for the other $n-1$ players. The result is a collection of n-degree polynomials over which n optimization problems must be solved simultaneously.

Example 9.21 (A Three-Player Nonzero-Sum Game) Suppose there are three players, P_1, P_2, and P_3, each of which has two actions, 1 and 2. A deterministic strategy is specified by a vector such as $(1, 2, 1)$, which indicates $u^1 = 1$, $u^2 = 2$, and $u^3 = 1$.

Now some costs will be defined. For convenience, let

$$L(i, j, k) = \Big(L_1(i, j, k),\ L_2(i, j, k),\ L_3(i, j, k) \Big) \tag{9.82}$$

for each $i, j, k \in \{1, 2\}$. Let the costs be

$$\begin{array}{ll} L(1,1,1) = (1,1,-2) & L(1,1,2) = (-4,3,1) \\ L(1,2,1) = (2,-4,2) & L(1,2,2) = (-5,-5,10) \\ L(2,1,1) = (3,-2,-1) & L(2,1,2) = (-6,-6,12) \\ L(2,2,1) = (2,2,-4) & L(2,2,2) = (-2,3,-1). \end{array} \tag{9.83}$$

There are two deterministic Nash equilibria, which yield the costs $(2, -4, 2)$ and $(3, -2, -1)$. These can be verified using (9.79). Each player is satisfied with the outcome given the actions chosen by the other players. Unfortunately, both Nash equilibria are both admissible. Therefore, some collaboration would be needed between the players to ensure that no regret will occur. ■

9.5 Decision Theory Under Scrutiny

Numerous models for decision making were introduced in this chapter. These provide a foundation for planning under uncertainty, which is the main focus of Part III. Before constructing planning models with this foundation, it is important to critically assess how appropriate it may be in applications. You may have had many questions while reading Sections 9.1 to 9.4. How are the costs determined? Why should we believe that optimizing the *expected* cost is the right thing to do? What happens if prior probability distributions are not available? Is worst-case analysis too conservative? Can we be sure that players in a game will follow the assumed rational behavior? Is it realistic that players know each other's cost functions? The purpose of this section is to help shed some light on these questions. A building is only as good as its foundation. Any mistakes made by misunderstanding the limitations of decision theory will ultimately work their way into planning formulations that are constructed from them.

9.5.1 Utility Theory and Rationality

This section provides some justification for using cost functions and then minimizing their expected value under Formulations 9.3 and 9.4. The resulting framework is called *utility theory*, which is usually formulated using rewards instead of costs. As stated in Section 9.1.1, a cost can be converted into a reward by multiplying by -1 and then swapping each maximization with minimization. We will therefore talk about a reward R with the intuition that a higher reward is better.

Comparing rewards

Imagine assigning reward values to various outcomes of a decision-making process. In some applications numerical values may come naturally. For example, the reward might be the amount of money earned in a financial investment. In robotics applications, one could negate time to execute a task or the amount of energy consumed. For example, the reward could indicate the amount of remaining battery life after a mobile robot builds a map.

In some applications the source of rewards may be subjective. For example, what is the reward for washing dishes, in comparison to sweeping the floor? Each person would probably assign different rewards, which may even vary from day to day. It may be based on their enjoyment or misery in performing the task, the amount of time each task would take, the perceptions of others, and so on. If decision theory is used to automate the decision process for a human "client," then it is best to consult carefully with the client to make sure you know their preferences. In this situation, it may be possible to sort their preferences and then assign rewards that are consistent with the ordering.

Once the rewards are assigned, consider making a decision under Formulation 9.1, which does not involve nature. Each outcome corresponds directly to an action, $u \in U$. If the rewards are given by $R : U \to \mathbb{R}$, then the cost, L, can be defined as $L(u) = -R(u)$ for every $u \in U$. Satisfying the client is then a matter of choosing u to minimize L.

Now consider a game against nature. The decision now involves comparing probability distributions over the outcomes. The space of all probability distributions may be enormous, but this is simplified by using expectation to map each probability distribution (or density) to a real value. The concern should be whether this projection of distributions onto real numbers will fail to reflect the true preferences of the client. The following example illustrates the effect of this.

Example 9.22 (Do You Like to Gamble?) Suppose you are given three choices:

1. You can have 1000 Euros.
2. We will toss an unbiased coin, and if the result is heads, then you will receive 2000 Euros. Otherwise, you receive nothing.
3. With probability 2/3, you can have 3000 Euros; however, with probability 1/3, you have to give me 3000 Euros.

The expected reward for each of these choices is 1000 Euros, but would you really consider these to be equivalent? Your love or disdain for gambling is not being taken into account by the expectation. How should such an issue be considered in games against nature? ■

To begin to fix this problem, it is helpful to consider another scenario. Many people would probably agree that having more money is preferable (if having too much worries you, then you can always give away the surplus to your favorite charities). What is interesting, however, is that being wealthy decreases the perceived value of money. This is illustrated in the next example.

Example 9.23 (Reality Television) Suppose you are lucky enough to appear on a popular reality television program. The point of the show is to test how far you will go in making a fool out of yourself, or perhaps even torturing yourself, to earn some money. You are asked to do some unpleasant task (such as eating cockroaches, or holding your head under water for a long time, and so on.). Let u_1 be the action to agree to do the task, and let u_2 mean that you decline the opportunity. The prizes are expressed in U.S. dollars. Imagine that you are a starving student on a tight budget.

Below are several possible scenarios that could be presented on the television program. Consider how you would react to each one.

1. Suppose that u_1 earns you \$1 and u_2 earns you nothing. Purely optimizing the reward would lead to choosing u_1, which means performing the unpleasant task. However, is this worth \$1? The problem so far is that we are not taking into account the amount of discomfort in completing a task. Perhaps it might make sense to make a reward function that shifts the dollar values by subtracting the amount for which you would be just barely willing to perform the task.

2. Suppose that u_1 earns you \$10,000 and u_2 earns you nothing. \$10,000 is assumed to be an enormous amount of money, clearly worth enduring any torture inflicted by the television program. Thus, u_1 is preferable.
3. Now imagine that the television host first gives you \$10 million just for appearing on the program. Are you still willing to perform the unpleasant task for an extra \$10,000? Probably not. What is happening here? Your sense of value assigned to money seems to decrease as you get more of it, right? It would not be too interesting to watch the program if the contestants were all wealthy oil executives.
4. Suppose that you have performed the task and are about to win the prize. Just to add to the drama, the host offers you a gambling opportunity. You can select action u_1 and receive \$10,000, or be a gambler by selecting u_2 and have probability 1/2 of winning \$25,000 by the tossing of a fair coin. In terms of the expected reward, the clear choice is u_2. However, you just completed the unpleasant task and expect to earn money. The risk of losing it all may be intolerable. Different people will have different preferences in this situation.
5. Now suppose once again that you performed the task. This time your choices are u_1, to receive \$100, or u_2, to have probability 1/2 of receiving \$250 by tossing a fair coin. The host is kind enough, though, to let you play 100 times. In this case, the expected totals for the two actions are \$10,000 and \$12,500, respectively. This time it seems clear that the best choice is to gamble. After 100 independent trials, we would expect that, with extremely high probability, over \$10,000 would be earned. Thus, reasoning by expected-case analysis seems valid if we are allowed numerous, independent trials. In this case, with high probability a value close to the expected reward should be received.

■

Based on these examples, it seems that the client or evaluator of the decision-making system must indicate preferences between probability distributions over outcomes. There is a formal way to ensure that once these preferences are assigned, a cost function can be designed for which its expectation faithfully reflects the preferences over distributions. This results in *utility theory*, which involves the following steps:

1. Require that the client is *rational* when assigning preferences. This notion is defined through axioms.

2. If the preferences are assigned in a way that is consistent with the axioms, then a utility function is guaranteed to exist. When expected utility is optimized, the preferences match exactly those of the client.

3. The cost function can be derived from the utility function.

The client must specify preferences among probability distributions of outcomes. Suppose that Formulation 9.2 is used. For convenience, assume that U and Θ are finite. Let X denote a *state space* based on outcomes.[5] Let $f : U \times \Theta \to X$ denote a mapping that assigns a state to every outcome. A simple example is to declare that $X = U \times \Theta$ and make f the identity map. This makes the outcome space and state space coincide. It may be convenient, though, to use f to collapse the space of outcomes down to a smaller set. If two outcomes map to the same state using f, then it means that the outcomes are indistinguishable as far as rewards or costs are concerned.

Let z denote a probability distribution over X, and let Z denote the set of all probability distributions over X. Every $z \in Z$ is represented as an n-dimensional vector of probabilities in which $n = |X|$; hence, it is considered as an element of $\mathbb{R}^n$. This makes it convenient to "blend" two probability distributions. For example, let $\alpha \in (0, 1)$ be a constant, and let z_1 and z_2 be any two probability distributions. Using scalar multiplication, a new probability distribution, $\alpha z_1 + (1 - \alpha)z_2$, is obtained, which is a *blend* of z_1 and z_2. Conveniently, there is no need to normalize the result. It is assumed that z_1 and z_2 initially have unit magnitude. The blend has magnitude $\alpha + (1 - \alpha) = 1$.

The modeler of the decision process must consult the client to represent preferences among elements of Z. Let $z_1 \prec z_2$ mean that z_2 is strictly preferred over z_1. Let $z_1 \approx z_2$ mean that z_1 and z_2 are equivalent in preference. Let $z_1 \preceq z_2$ mean that either $z_1 \prec z_2$ or $z_1 \approx z_2$. The following example illustrates the assignment of preferences.

Example 9.24 (Indicating Preferences) Suppose that $U = \Theta = \{1, 2\}$, which leads to four possible outcomes: $(1, 1)$, $(1, 2)$, $(2, 1)$, and $(2, 2)$. Imagine that nature represents a machine that generates 1 or 2 according to a probability distribution. The action is to guess the number that will be generated by the machine. If you pick the same number, then you win that number of gold pieces. If you do not pick the same number, then you win nothing, but also lose nothing.

Consider the construction of the state space X by using f. The outcomes $(2, 1)$ and $(1, 2)$ are identical concerning any conceivable reward. Therefore, these should map to the same state. The other two outcomes are distinct. The state space therefore needs only three elements and can be defined as $X = \{0, 1, 2\}$.

[5]In most utility theory literature, this is referred to as a *reward space*, $\mathcal{R}$ [21].

Let $f(2, 1) = f(1, 2) = 0$, $f(1, 1) = 1$, and $f(2, 2) = 2$. Thus, the last two states indicate that some gold will be earned.

The set Z of probability distributions over X is now considered. Each $z \in Z$ is a three-dimensional vector. As an example, $z_1 = [1/2 \;\; 1/4 \;\; 1/4]$ indicates that the state will be 0 with probability 1/2, 1 with probability 1/4, and 2 with probability 1/4. Suppose $z_2 = [1/3 \;\; 1/3 \;\; 1/3]$. Which distribution would you prefer? It seems in this case that z_2 is uniformly better than z_1 because there is a greater chance of winning gold. Thus, we declare $z_1 \prec z_2$. The distribution $z_3 = [1 \;\; 0 \;\; 0]$ seems to be the worst imaginable. Hence, we can safely declare $z_3 \prec z_1$ and $z_1 \prec z_2$.

The procedure of determining the preferences can become quite tedious for complicated problems. In the current example, Z is a 2D subset of $\mathbb{R}^3$. This subset can be partitioned into a finite set of regions over which the client may be able to clearly indicate preferences. One of the major criticisms of this framework is the impracticality of determining preferences over Z [237].

After the preferences are determined, is there a way to ensure that a real-value function on X exists for which the expected value exactly reflects the preferences? If the axioms of rationality are satisfied by the assignment of preferences, then the answer is *yes*. These axioms are covered next. ■

Axioms of rationality

To meet the goal of designing a utility function, it turns out that the preferences must follow rules called the *axioms of rationality*. They are sensible statements of consistency among the preferences. As long as these are followed, then a utility function is guaranteed to exist (detailed arguments appear in [70, 237]). The decision maker is considered *rational* if the following axioms are followed when defining $\prec$ and $\approx$:[6]

1. If $z_1, z_2 \in Z$, then either $z_1 \preceq z_2$ or $z_2 \preceq z_1$.
   "You must be able to make up your mind."

2. If $z_1 \preceq z_2$ and $z_2 \preceq z_3$, then $z_1 \preceq z_3$.
   "Preferences must be transitive."

3. If $z_1 \prec z_2$, then
$$\alpha z_1 + (1 - \alpha)z_3 \prec \alpha z_2 + (1 - \alpha)z_3, \tag{9.84}$$
   for any $z_3 \in Z$ and $\alpha \in (0, 1)$.
   "Evenly blending in a new distribution does not alter preference."

4. If $z_1 \prec z_2 \prec z_3$, then there exists some $\alpha \in (0, 1)$ and $\beta \in (0, 1)$ such that
$$\alpha z_1 + (1 - \alpha)z_3 \prec z_2 \tag{9.85}$$

[6]Alternative axiom systems exist [70, 240].

and

$$z_2 \prec \beta z_1 + (1-\beta) z_3. \tag{9.86}$$

"There is no heaven or hell."

Each axiom has an intuitive interpretation that makes practical sense. The first one simply indicates that the preference direction can always be inferred for a pair of distributions. The second axiom indicates that preferences must be transitive.[7] The last two axioms are somewhat more complicated. In the third axiom, z_2 is strictly preferred to z_1. An attempt is made to cause confusion by blending in a third distribution, z_3. If the same "amount" of z_3 is blended into both z_1 and z_2, then the preference should not be affected. The final axiom involves z_1, z_2, and z_3, each of which is strictly better than its predecessor. The first equation, (9.85), indicates that if z_2 is strictly better than z_1, then a tiny amount of z_3 can be blended into z_1, with z_2 remaining preferable. If z_3 had been like "heaven" (i.e., infinite reward), then this would not be possible. Similarly, (9.86) indicates that a tiny amount of z_1 can be blended into z_3, and the result remains better than z_2. This means that z_1 cannot be "hell," which would have infinite negative reward.[8]

Constructing a utility function

If the preferences have been determined in a way consistent with the axioms, then it can be shown that a *utility function* always exists. This means that there exists a function $\mathcal{U} : X \to \mathbb{R}$ such that, for all $z_1, z_2 \in Z$,

$$z_1 \prec z_2 \text{ if and only if } E_{z_1}[\mathcal{U}] < E_{z_2}[\mathcal{U}], \tag{9.87}$$

in which E_{z_i} denotes the expected value of $\mathcal{U}$, which is being treated as a random variable under the probability distribution z_i. The existence of $\mathcal{U}$ implies that it is safe to determine the best action by maximizing the expected utility.

A reward function can be defined using a utility function, $\mathcal{U}$, as $R(u,\theta) = \mathcal{U}(f(u,\theta))$. The utility function can be converted to a cost function as $L(u,\theta) = -R(u,\theta) = -\mathcal{U}(f(u,\theta))$. Minimizing the expected cost, as was recommended under Formulations 9.3 and 9.4 with probabilistic uncertainty, now seems justified under the assumption that $\mathcal{U}$ was constructed correctly to preserve preferences.

Unfortunately, establishing the existence of a utility function does not produce a systematic way to construct it. In most circumstances, one is forced to design $\mathcal{U}$ by a trial-and-error process that involves repeatedly checking the preferences. In the vast majority of applications, people create utility and cost functions without regard to the implications discussed in this section. Thus, undesirable conclusions may be reached in practice. Therefore, it is important not to be too confident about the quality of an optimal decision rule.

[7]For some reasonable problems, however, transitivity is not desirable. See the Candorcet and Simpson paradoxes in [237].

[8]Some axiom systems allow infinite rewards, which lead to utility and cost functions with infinite values, but this is not considered here.

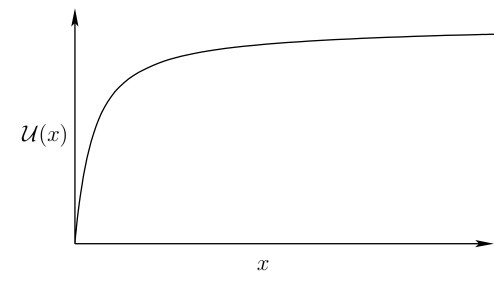


Figure 9.6: The utility of new amounts of money decreases as the total accumulation of wealth increases. The utility function may even bounded.

Note that if worst-case analysis had been used, then most of the problems discussed here could have been avoided. Worst-case analysis, however, has its weaknesses, which will be discussed in Section 9.5.3.

Example 9.25 (The Utility of Money) We conclude the section by depicting a utility function that is often applied to money. Suppose that the state space $X = \mathbb{R}$, which corresponds to the amount of U.S. dollars earned. The utility of money applied by most people indicates that the value of new increments of money decreases as the total accumulated wealth increases. The utility function may even be bounded. Imagine there is some maximum dollar amount, such as $\$10^{100}$, after which additional money has no value. A typical utility curve is shown in Figure 9.6 [21]. ■

9.5.2 Concerns Regarding the Probabilistic Model

Section 9.5.1 addressed the source of cost functions and the validity of taking their expectations. This section raises concerns over the validity of the probability distributions used in Section 9.2. The two main topics are criticisms of Bayesian methods in general and problems with constructing probability distributions.

Bayesians vs. frequentists

For the past century and a half, there has been a fundamental debate among statisticians on the *meaning* of probabilities. Virtually everyone is satisfied with the axioms of probability, but beyond this, what is their meaning when making inferences? The two main camps are the *frequentists* and the *Bayesians*. A form of Bayes' rule was published in 1763 after the death of Bayes [16]. During most of the nineteenth century Bayesian analysis tended to dominate literature; however, during the twentieth century, the frequentist philosophy became more popular as a more rigorous interpretation of probabilities. In recent years, the credibility of Bayesian methods has been on the rise again.

As seen so far, a Bayesian interprets probabilities as the degree of belief in a hypothesis. Under this philosophy, it is perfectly valid to begin with a prior distribution, gather a few observations, and then make decisions based on the resulting posterior distribution from applying Bayes' rule.

From a frequentist perspective, Bayesian analysis makes far too liberal use of probabilities. The frequentist believes that probabilities are only defined as the quantities obtained in the limit after the number of independent trials tends to infinity. For example, if an unbiased coin is tossed over numerous trials, the probability 1/2 represents the value to which the ratio between heads and the total number of trials will converge as the number of trials tends to infinity. On the other hand, a Bayesian might say that the probability that the *next* trial results in heads is 1/2. To a frequentist, this interpretation of probability is too strong.

Frequentists have developed a version of decision theory based on their philosophy; comparisons between the two appear in [237]. As an example, a frequentist would advocate optimizing the following *frequentist risk* to obtain a decision rule:

$$R(\theta, \pi) = \int_y L(\pi(y), \theta) P(y|\theta) dy, \tag{9.88}$$

in which π represents the strategy, $\pi : Y \rightarrow U$. The frequentist risk averages over all data, rather than making a decision based on a single observation, as advocated by Bayesians in (9.26). The probability $P(y|\theta)$ is assumed to be obtained in the limit as the number of independent data trials tends to infinity. The main drawback in using (9.88) is that the optimization depends on θ. The resulting best decision rule must depend on θ, which is unknown. In some limited cases, it may be possible to select some π that optimizes (9.88) for all θ, but this rarely occurs. Thus, the frequentist risk can be viewed as a constraint on the desirability of strategies, but it usually is not powerful enough to select a single one. This problem is reminiscent of Pareto optimality, which was discussed in Section 9.1.1. The frequentist approach attempts to be more conservative and rigorous, with the result being that weaker statements are made regarding decisions.

The source of prior distributions

Suppose that the Bayesian method has been adopted. The most widespread concern in all Bayesian analyses is the source of the prior distribution. In Section 9.2, this is represented as $P(\theta)$ (or $p(\theta)$), which represents a distribution (or density) over the nature action space. The best way to obtain $P(\theta)$ is by estimating the distribution over numerous independent trials. This brings its definition into alignment with frequentist views. This was possible with Example 9.11, in which $P(\theta)$ could be reliably estimated from the frequency of occurrence of letters across numerous pages of text. The distribution could even be adapted to a particular language or theme.

In most applications that use decision theory, however, it is impossible or too costly to perform such experiments. What should be done in this case? If a prior distribution is simply "made up," then the resulting posterior probabilities may be suspect. In fact, it may be invalid to call them probabilities at all. Sometimes the term *subjective probabilities* is used in this case. Nevertheless, this is commonly done because there are few other options. One of these options is to resort to frequentist decision theory, but, as mentioned, it does not work with single observations.

Fortunately, as the number of observations increases, the influence of the prior on the Bayesian posterior distributions diminishes. If there is only one observation, or even none as in Formulation 9.3, then the prior becomes very influential. If there is little or no information regarding $P(\theta)$, the distribution should be designed as carefully as possible. It should also be understood that whatever conclusions are made with this assumption, they are biased by the prior. Suppose this model is used as the basis of a planning approach. You might feel satisfied computing the "optimal" plan, but this notion of optimality could still depend on some arbitrary initial bias due to the assignment of prior values.

If there is no information available, then it seems reasonable that $P(\theta)$ should be as uniform as possible over Θ. This was referred to by Laplace as the "principle of insufficient reason" [156]. If there is no reason to believe that one element is more likely than another, then they should be assigned equal values. This can also be justified by using Shannon's entropy measure from information theory [6, 62, 246]. In the discrete case, this is

$$-\sum_{\theta \in \Theta} P(\theta) \lg P(\theta), \tag{9.89}$$

and in the continuous case it is

$$-\int_{\Theta} p(\theta) \lg p(\theta) d\theta. \tag{9.90}$$

This entropy measure was developed in the context of communication systems to estimate the minimum number of bits needed to encode messages delivered through a noisy medium. It generally indicates the amount of uncertainty associated with the distribution. A larger value of entropy implies a greater amount of uncertainty.

It turns out that the entropy function is maximized when $P(\theta)$ is a uniform distribution, which seems to justify the principle of insufficient reason. This can be considered as a *noninformative prior*. The idea is even applied quite frequently when $\Theta = \mathbb{R}$, which leads to an *improper prior*. The density function cannot maintain a constant, nonzero value over all of $\mathbb{R}$ because its integral would be infinite. Since the decisions made in Section 9.2 do not depend on any normalizing factors, a constant value can be assigned for $p(\theta)$ and the decisions are not affected by the fact that the prior is improper.

The main difficulty with applying the entropy argument in the selection of a prior is that Θ itself may be chosen in a number of arbitrary ways. Uniform assignments to different choices of Θ ultimately yield different information regarding the priors. Consider the following example.

Example 9.26 (A Problem with Noninformative Priors) Consider a decision about what activities to do based on the weather. Imagine that there is absolutely no information about what kind of weather is possible. One possible assignment is $\Theta = \{p, c\}$, in which p means "precipitation" and c means "clear." Maximizing (9.89) suggests assigning $P(p) = P(c) = 1/2$.

After thinking more carefully, perhaps we would like to distinguish between different kinds of precipitation. A better set of nature actions would be $\Theta = \{r, s, c\}$, in which c still means "clear," but precipitation p has been divided into r for "rain" and s for "snow." Now maximizing (9.89) assigns probability 1/3 to each nature action. This is clearly different from the original assignment. Now that we distinguish between different kinds of precipitation, it seems that precipitation is much more likely to occur. Does our preference to distinguish between different forms of precipitation really affect the weather? ■

Example 9.27 (Noninformitive Priors for Continuous Spaces) Similar troubles can result in continuous spaces. Recall the parameter estimation problem described in Example 9.12. Suppose instead that the task is to estimate a line based on some data points that were supposed to fall on the line but missed due to noise in the measurement process.

What initial probability density should be assigned to Θ, the set of all lines? Suppose that the line lives in $Z = \mathbb{R}^2$. The line equation can be expressed as

$$\theta_1 z_1 + \theta_2 z_2 + \theta_3 = 0. \tag{9.91}$$

The problem is that if the parameter vector, $\theta = [\theta_1 \quad \theta_2 \quad \theta_3]$, is multiplied by a scalar constant, then the same line is obtained. Thus, even though $\theta \in \mathbb{R}^3$, a constraint must be added. Suppose we require that

$$\theta_1^2 + \theta_2^2 + \theta_3^1 = 1 \tag{9.92}$$

and $\theta_1 \geq 0$. This mostly fixes the problem and ensures that each parameter value corresponds to a unique line (except for some duplicate cases at $\theta_1 = 0$, but these can be safely neglected here). Thus, the parameter space is the upper half of a sphere, $\mathbb{S}^2$. The maximum-entropy prior suggests assigning a uniform probability density to Θ. This may feel like the right thing to do, but this notion of uniformity is biased by the particular constraint applied to the parameter space to ensure uniqueness. There are many other choices. For example, we could replace (9.92) by constraints that force the points to lie on the upper half of the surface of cube, instead of a sphere. A uniform probability density assigned in this new parameter space certainly differs from one over the sphere.

In some settings, there is a natural representation of the parameter space that is invariant to certain transformations. Section 5.1.4 introduced the notion of Haar measure. If the Haar measure is used as a noninformative prior, then a meaningful notion of uniformity may be obtained. For example, suppose that the parameter space is $SO(3)$. Uniform probability mass over the space of unit quaternions, as suggested in Example 5.14, is an excellent choice for a noninformative prior because it is consistent with the Haar measure, which is invariant to group operations applied to the events. Unfortunately, a Haar measure does not exist for most spaces that arise in practice.[9] ■

Incorrect assumptions on conditional distributions

One final concern is that many times even the distribution $P(y|\theta)$ is incorrectly estimated because it is assumed arbitrarily to belong to a family of distributions. For example, it is often very easy to work with Gaussian densities. Therefore, it is tempting to assume that $p(y|\theta)$ is Gaussian. Experiments can be performed to estimate the mean and variance parameters. Even though some best fit will be found, it does not necessarily imply that a Gaussian is a good representation. Conclusions based on this model may be incorrect, especially if the true distribution has a different shape, such as having a larger tail or being multimodal. In many cases, *nonparametric* methods may be needed to avoid such biases. Such methods do not assume a particular family of distributions. For example, imagine estimating a probability distribution by making a histogram that records the frequency of y occurrences for a fixed value of θ. The histogram can then be normalized to contain a representation of the probability distribution without assuming an initial form.

9.5.3 Concerns Regarding the Nondeterministic Model

Given all of the problems with probabilistic modeling, it is tempting to abandon the whole framework and work strictly with the nondeterministic model. This only requires specifying Θ, without indicating anything about the relative likelihoods of various actions. Therefore, most of the complicated issues presented in Sections 9.5.1 and 9.5.2 vanish. Unfortunately, this advantage comes at a substantial price. Making decisions with worst-case analysis under the nondeterministic model has its own shortcomings. After considering the trade-offs, you can decide which is most appropriate for a particular application of interest.

[9]A locally compact topological group is required [103, 239].

The first difficulty is to ensure that Θ is sufficiently large to cover all possibilities. Consider Formulation 9.6, in which nature acts twice. Through a nature observation action space, $\Psi(\theta)$, interference is caused in the measurement process. Suppose that $\Theta = \mathbb{R}$ and $h(\theta, \psi) = \theta + \psi$. In this case, $\Psi(\theta)$ can be interpreted as the measurement error. What is the maximum amount of error that can occur? Perhaps a sonar is measuring the distance from the robot to a wall. Based on the sensor specifications, it may be possible to construct a nice bound on the error. Occasionally, however, the error may be larger than this bound. Sonars sometimes fail to hear the required echo to compute the distance. In this case the reported distance is ∞. Due to reflections, extremely large errors can sometimes occur. Although such errors may be infrequent, if we want *guaranteed* performance, then large or even infinite errors should be included in $\Psi(\theta)$. The problem is that worst-case reasoning could always conclude that the sensor is useless by reporting ∞. Any statistically valid information that could be gained from the sensor would be ignored. Under the probabilistic model, it is easy to make $\Psi(\theta)$ quite large and then assign very small probabilities to larger errors. The problem with nondeterministic uncertainty is that $\Psi(\theta)$ needs to be smaller to make appropriate decisions; however, theoretically "guaranteed" performance may not truly be guaranteed in practice.

Once a nondeterministic model is formulated, the optimal decision rule may produce results that seem absurd for the intended application. The problem is that the DM cannot tolerate any risk. An action is applied only if the result can be guaranteed. The hope of doing better than the worst case is not taken into account. Consider the following example:

Example 9.28 (A Problem with Conservative Decision Making) Suppose that a friend offers you the choice of either a check for 1000 Euros or 1 Euro in cash. With the check, you must take it to the bank, and there is a small chance that your friend will have insufficient funds in the account. In this case, you will receive nothing. If you select the 1 Euro in cash, then you are guaranteed to earn something.

The following cost matrix reflects the outcomes (ignoring utility theory):

$$\begin{array}{c} \\ \Theta \end{array} \begin{array}{c} U \\ \begin{array}{|c|c|} \hline 1 & 1000 \\ \hline 1 & 0 \\ \hline \end{array} \end{array} . \qquad (9.93)$$

Using probabilistic analysis, we might conclude that it is best to take the check. Perhaps the friend is even known to be very wealthy and responsible with banking accounts. This information, however, cannot be taken into account in the decision-making process. Using worst-case analysis, the optimal action is to take the 1 Euro in cash. You may not feel too good about it, though. Imagine the regret if you later learn that the account had sufficient funds to cash the check for 1000 Euros. ■

Thus, it is important to remember the price that one must pay for wanting results that are absolutely guaranteed. The probabilistic model offers the flexibility of incorporating statistical information. Sometimes the probabilistic model can be viewed as a generalization of the nondeterministic model. If it is assumed that nature acts after the robot, then the nature action can take this into account, as incorporated into Formulation 9.4. In the nondeterministic case, $\Theta(u)$ is specified, and in the probabilistic case, $P(\theta|u)$ is specified. The distribution $P(\theta|u)$ can be designed so that nature selects with very high probability the $\theta \in \Theta$ that maximizes $L(u, \theta)$. In Example 9.28, this would mean that the probability that the check would bounce (resulting in no earnings) would by very high, such as 0.999999. In this case, even the optimal action under the probabilistic model is to select the 1 Euro in cash. For virtually any decision problem that is modeled using worst-case analysis, it is possible to work backward and derive possible priors for which the same decision would be made using probabilistic analysis. In Example 9.4, it seemed as if the decision was based on assuming that with very high probability, the check would bounce, even though there were no probabilistic models.

This means that worst-case analysis under the nondeterministic model can be considered as a special case of a probabilistic model in which the prior distribution assigns high probabilities to the worst-case outcomes. The justification for this could be criticized in the same way that other prior assignments are criticized in Bayesian analysis. What is the basis of this particular assignment?

9.5.4 Concerns Regarding Game Theory

One of the most basic limitations of game theory is that each player must know the cost functions of the other players. As established in Section 9.5.1, it is even quite difficult to determine an appropriate cost function for a single decision maker. It is even more difficult to determine costs and motivations of other players. In most practical settings this information is not available. One possibility is to model uncertainty associated with knowledge of the cost function of another player. Bayesian analysis could be used to reason about the cost based on observations of actions chosen by the player. Issues of assigning priors once again arise. One of the greatest difficulties in allowing uncertainties in the cost functions is that a kind of "infinite reflection" occurs [111]. For example, if I am playing a game, does the other player know my cost function? I may be uncertain about this. Furthermore, does the other player know that I do not completely know its cost function? This kind of second-guessing can occur indefinitely, leading to a nightmare of nested reasoning and assignments of prior distributions.[10]

The existence of saddle points or Nash equilibria was assured by using ran-

[10]Readers familiar with the movie *The Princess Bride* may remember the humorous dialog between Vizzini and the Dread Pirate Roberts about which goblet contains the deadly Iocane powder.

domized strategies. Mathematically, this appears to be a clean solution to a frustrating problem; however, it also represents a substantial change to the model. Many games are played just once. For the expected-case results to converge, the game must be played an infinite number of times. If a game is played once, or only a few times, then the players are very likely to experience regret, even though the theory based on expected-case analysis indicates that regret is eliminated.

Another issue is that intelligent human players may fundamentally alter their strategies after playing a game several times. It is very difficult for humans to simulate a randomized strategy (assuming they even want to, which is unlikely). There are even international tournaments in which the players repeatedly engage in classic games such as Rock-Paper-Scissors or the Prisoner's Dilemma. For an interesting discussion of a tournament in which people designed programs that repeatedly compete on the Prisoner's Dilemma, see [268]. It was observed that even some cooperation often occurs after many iterations, which secures greater rewards for both players, even though they cannot communicate. A famous strategy arose in this context called Tit-for-Tat (written by Anatol Rapoport), which in each stage repeated the action chosen by the other player in the last stage. The approach is simple yet surprisingly successful.

In the case of nonzero-sum games, it is particularly disheartening that multiple Nash equilibria may exist. Suppose there is only one admissible equilibrium among several Nash equilibria. Does it really seem plausible that an adversary would think very carefully about the various Nash equilibria and pick the admissible one? Perhaps some players are conservative and even play security strategies, which completely destroys the assumptions of minimizing regret. If there are multiple admissible Nash equilibria, it appears that regret is unavoidable unless there is some collaboration between players. This result is unfortunate if such collaboration is impossible.

Further Reading

Section 9.1 covered very basic concepts, which can be found in numerous books and on the Internet. For more on Pareto optimality, see [243, 265, 283, 302]. Section 9.2 is inspired mainly by decision theory books. An excellent introduction is [21]. Other sources include [70, 73, 184, 237]. The "game against nature" view is based mainly on [31]. Pattern classification, which is an important form of decision theory, is covered in [3, 73, 82, 193]. Bayesian networks [222] are a popular representation in artificial intelligence research and often provide compact encodings of information for complicated decision-making problems.

Further reading on the game theory concepts of Sections 9.3 and 9.4 can be found in many books (e.g., [9, 210]). A fun book that has many examples and intuitions is [268]. For games that have infinite action sets, see [9]. The computation of randomized Nash equilibria remains a topic of active research. A survey of methods appears in [191]; see also [147, 192]. The coupled polynomial equations that appear in computing randomized Nash equilibria may seem to suggest applying algorithms from computational algebraic geometry, as were needed in Section 6.4 to solve this kind of problem in combinatorial motion planning. An approach that uses such tools is given in [67]. Contrary to the noncooperative games defined in Section 9.4, *cooperative game theory* investigates ways in which various players can form coalitions to improve their rewards [223].

Parts of Section 9.5 were inspired by [21]. Utility theory appears in most decision theory books (e.g., [21]) and in some artificial intelligence books (e.g., [240]). An in-depth discussion of Bayesian vs. frequentist issues appears in [237]. For a thorough introduction to constructing cost models for decision making, see [145].

Exercises

1. Suppose that a single-stage two-objective decision-making problem is defined in which there are two objectives and a continuous set of actions, $U = [-10, 10]$. The cost vector is $L = [u^2 \;\; u-1]$. Determine the set of Pareto-optimal actions.

2. Let

$$\begin{array}{cc} & \Theta \\ U & \begin{array}{|c|c|c|c|} \hline -1 & 3 & 2 & -1 \\ \hline -1 & 0 & 7 & -1 \\ \hline 1 & 5 & 5 & -2 \\ \hline \end{array} \end{array}$$

define the cost for each combination of choices by the decision maker and nature. Let nature's randomized strategy be $[1/5 \;\; 2/5 \;\; 1/10 \;\; 3/10]$.

   (a) Use nondeterministic reasoning to find the minimax decision and worst-case cost.

   (b) Use probabilistic reasoning to find the best expected-case decision and expected cost.

3. Many reasonable decision rules are possible, other than those considered in this chapter.

   (a) Exercise 2(a) reflects extreme pessimism. Suppose instead that extreme optimism is used. Select the choice that optimizes the best-case cost for the matrix in Exercise 2.

   (b) One approach is to develop a coefficient of optimism, $\alpha \in [0, 1]$, which allows one to interpolate between the two extreme scenarios. Thus, a decision, $u \in U$, is chosen by minimizing

$$\alpha \max_{\theta \in \Theta} \Big\{ L(u, \theta) \Big\} + (1 - \alpha) \min_{\theta \in \Theta} \Big\{ L(u, \theta) \Big\}. \qquad (9.94)$$

   Determine the optimal decision for this scenario under all possible choices for $\alpha \in [0, 1]$. Give your answer as a list of choices, each with a specified range of α.

4. Suppose that after making a decision, you observe the choice made by nature. How does the cost that you received compare with the best cost that could have been obtained if you chose something else, given this choice by nature? This difference in costs can be considered as *regret* or minimum "Doh!"[11] Psychologists have argued that some people make choices based on minimizing regret. It reflects how badly you wish you had done something else after making the decision.

   (a) Develop an expression for the worst-case regret, and use it to make a minimax regret decision using the matrix from Exercise 2.

   (b) Develop an expression for the expected regret, and use it to make a minimum expected regret decision.

5. Using the matrix from Exercise 2, consider the set of all probability distributions for nature. Characterize the set of all distributions for which the minimax decision and the best expected decision results in the same choice. This indicates how to provide reverse justification for priors.

6. Consider a Bayesian decision-theory scenario with cost function L. Show that the decision rule never changes if $L(u, \theta)$ is replaced by $aL(u, \theta) + b$, for any $a > 0$ and $b \in \mathbb{R}$.

7. Suppose that there are two classes, $\Omega = \{\omega_1, \omega_2\}$, with $P(\omega_1) = P(\omega_2) = \frac{1}{2}$. The observation space, Y, is $\mathbb{R}$. Recall from probability theory that the normal (or Gaussian) probability density function is

$$p(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(y-\mu)^2/2\sigma^2}, \tag{9.95}$$

   in which μ denotes the mean and σ^2 denotes the variance. Suppose that $p(y|\omega_1)$ is a normal density in which $\mu = 0$ and $\sigma^2 = 1$. Suppose that $p(y|\omega_2)$ is a normal density in which $\mu = 6$ and $\sigma^2 = 4$. Find the optimal classification rule, $\gamma : Y \to \Omega$. You are welcome to solve the problem numerically (by computer) or graphically (by careful function plotting). Carefully explain how you arrived at the answer in any case.

8. Let

$$\begin{array}{c|c|c|c|c|} & \multicolumn{4}{c}{\Theta} \\ \hline & 2 & -2 & -2 & 1 \\ U & -1 & -2 & -2 & 6 \\ & 4 & 0 & -3 & 4 \\ \hline \end{array}$$

   give the cost for each combination of choices by the decision maker and nature. Let nature's randomized strategy be [1/4 1/2 1/8 1/8].

   (a) Use nondeterministic reasoning to find the minimax decision and worst-case cost.

[11] In 2001, the Homer Simpson term "Doh!" was added to the Oxford English Dictionary as an expression of regret.

   (b) Use probabilistic reasoning to find the best expected-case decision and expected cost.

   (c) Characterize the set of all probability distributions for which the minimax decision and the best expected decision results in the same choice.

9. In a *constant-sum game*, the costs for any $u \in U$ and $v \in V$ add to yield

$$L_1(u, v) + L_2(u, v) = c \tag{9.96}$$

   for some constant c that is independent of u and v. Show that any constant-sum game can be transformed into a zero-sum game, and that saddle point solutions can be found using techniques for the zero-sum formulation.

10. Formalize Example 9.7 as a zero-sum game, and compute security strategies for the players. What is the expected value of the game?

11. Suppose that for two zero-sum games, there exists some nonzero $c \in \mathbb{R}$ for which the cost matrix of one game is obtained by multiplying all entries by c in the cost matrix of the other. Prove that these two games must have the same deterministic and randomized saddle points.

12. In the same spirit as Exercise 11, prove that two zero-sum games have the same deterministic and randomized saddle points if c is added to all matrix entries.

13. Prove that multiple Nash equilibria of a nonzero-sum game specified by matrices A and B are interchangeable if (A, B) as a game yields the same Nash equilibria as the game $(A, -A)$.

14. Analyze the game of Rock-Paper-Scissors for three players. For each player, assign a cost of 1 for losing, 0 for a tie, and -1 for winning. Specify the cost functions. Is it possible to avoid regret? Does it have a deterministic Nash equilibrium? Can you find a randomized Nash equilibrium?

15. Compute the randomized equilibrium point for the following zero-sum game:

$$\begin{array}{c|c|c|} & \multicolumn{2}{c}{V} \\ \hline U & 0 & -1 \\ & -1 & 2 \\ \hline \end{array} \; . \tag{9.97}$$

   Indicate the randomized strategies for the players and the resulting expected value of the game.

Implementations

16. Consider estimating the value of an unknown parameter, $\theta \in \mathbb{R}$. The prior probability density is a normal,

$$p(\theta) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(\theta-\mu)^2/2\sigma^2}, \tag{9.98}$$

with $\mu = 0$ and $\sigma = 4$. Suppose that a sequence, y_1, y_2, ..., y_k, of k observations is made and that each $p(y_i|\theta)$ is a normal density with $\mu = \theta$ and $\sigma = 9$. Suppose that u represents your guess of the parameter value. The task is select u to minimize the expectation of the cost, $L(u, \theta) = (u - \theta)^2$. Suppose that the "true" value of θ is 4. Determine the u^*, the minimal action with respect to the expected cost after observing: $y_i = 4$ for every $i \in \{1, \ldots, k\}$.

(a) Determine u^* for $k = 1$.

(b) Determine u^* for $k = 10$.

(c) Determine u^* for $k = 1000$.

This experiment is not very realistic because the observations should be generated by sampling from the normal density, $p(y_i|\theta)$. Repeat the exercise using values drawn with the normal density, instead of $y_k = 4$, for each k.

17. Implement an algorithm that computes a randomized saddle point for zero-sum games. Assume that one player has no more than two actions and the other may have any finite number of actions.

18. Suppose that a K-stage decision-making problem is defined using multiple objectives. There is a finite state space X and a finite action set $U(x)$ for each $x \in X$. A state transition equation, $x_{k+1} = f(x_k, u_k)$, gives the next state from a current state and input. There are N cost functionals of the form

$$L_i(u_1, \ldots, u_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F), \tag{9.99}$$

in which $F = K+1$. Assume that $l_F(x_F) = \infty$ if $x_F \in X_{goal}$ (for some goal region $X_{goal} \subset X$) and $l_F(x_F) = 0$ otherwise. Assume that there is no termination action (which simplifies the problem). Develop a value-iteration approach that finds the complete set of Pareto-optimal plans efficiently as possible. If two or more plans produce the same cost vector, then only one representative needs to be returned.

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 [23, 151, 153]. The concepts can be useful for modeling problems in mobile robotics because future states are usually unpredictable and can sometimes be modeled probabilistically [301] or using worst-case analysis [159]. 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* [59, 109].

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.

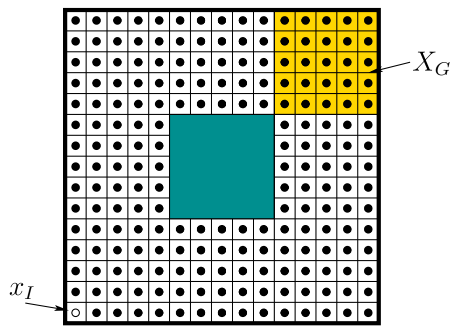


Figure 10.1: A grid-based shortest path problem with interference from nature.

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 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

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

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

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

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

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

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

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

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

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

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* [101], 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 $\pi : X \times \mathcal{K} \to U$. 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

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

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

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

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

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

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))

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

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

$$G^*_F(x_F) = l_F(x_F), \tag{10.34}$$

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))

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

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))

$$G^*_K(x_K) = \min_{u_K} \max_{\theta_K} \Big\{ l(x_K, u_K, \theta_K) + G^*_F(f(x_K, u_K, \theta_K)) \Big\}, \tag{10.36}$$

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))

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

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))

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

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

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

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))

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

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 $-\infty$.

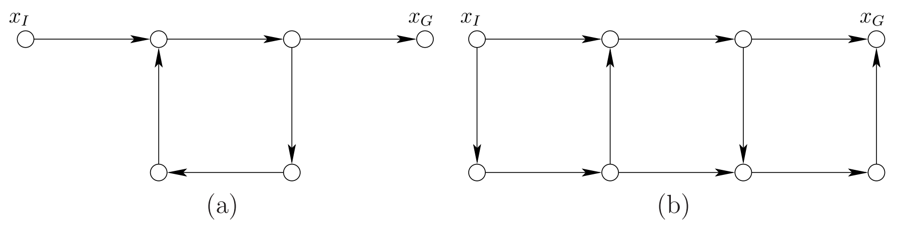


Figure 10.2: Plan-based state transition graphs. (a) The goal is possibly reachable, but not guaranteed reachable because an infinite cycle could occur. (b) The goal is guaranteed reachable because all flows lead to the goal.

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 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

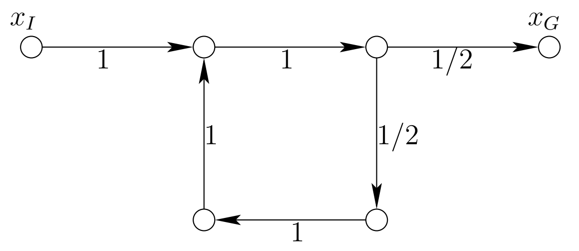


Figure 10.3: A plan-based state transition graph that causes asymptotic convergence. The probabilities of the transitions are shown on the edges. Longer and longer paths exist by traversing the cycle, but the probabilities become smaller.

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 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* [18]; 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

$$G_\pi(x) = l(x,\pi(x)) + G_\pi(f(x,\pi(x))). \qquad (10.50)$$

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

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

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

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

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

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

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)* [115, 287].

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

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.

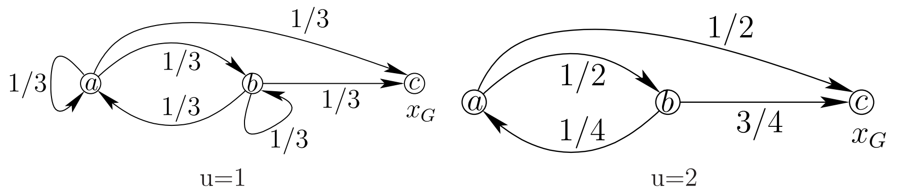


Figure 10.5: The probabilistic state transition graphs for $u = 1$ and $u = 2$. Transitions out of c are not shown because it is assumed that a termination action is always applied from x_g.

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 $c \in X_G$.

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 $\pi^* = \pi$. ■

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.

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

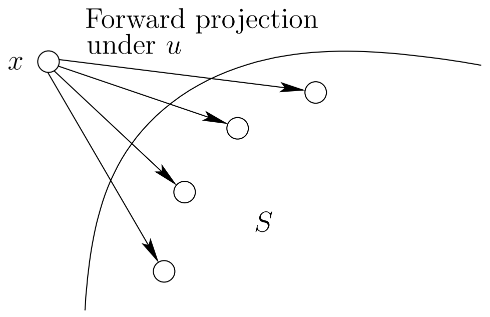


Figure 10.7: A state x can be added to S if there exists an action $u \in U(x)$ such that the one-stage forward projection is contained in S.

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 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 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

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.

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

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

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.

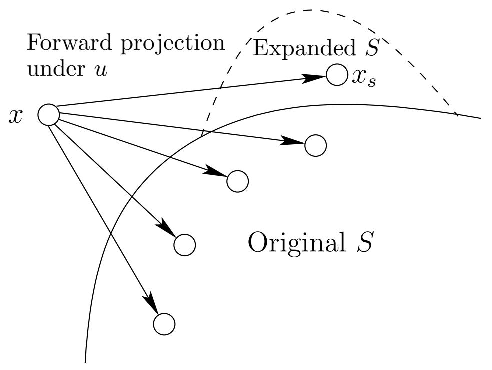


Figure 10.9: The worst-case cost-to-go is computed for any state x such that there exists a $u \in U(x)$ for which the one-stage forward projection is contained in the updated S and one state in the forward projection is x_s.

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

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

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."

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

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.

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 [26]. 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 [26], 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 [3, 271], 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 [27] 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 [25], 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:

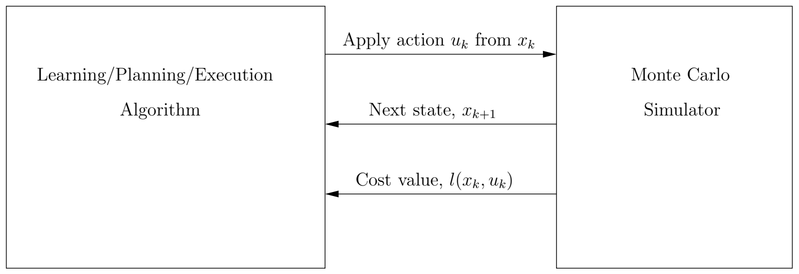

Learning/Planning/Execution Algorithm
Apply action u_k from x_k
Next state, x_{k+1}
Cost value, $l(x_k, u_k)$
Monte Carlo Simulator

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* [271]. 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, ..., 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

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

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 [271, 26]. Suppose that a trial is performed from x that results in the sequence $x_1 = x$, x_2, ..., x_k, ..., 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:

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

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* [270]. 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, ..., x_K, x_F generated by a trial. Let d_k denote a *temporal difference*, which is defined as

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

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 $1 \le k \le K$, 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

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

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 $\hat{G}_\pi$ [26, 27].

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 [27].

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* [20, 27, 152]. 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 [27] 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

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

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

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

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

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

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

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

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):

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

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:

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

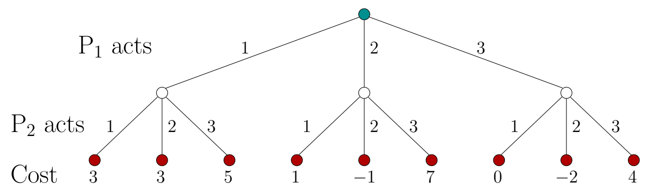


Figure 10.12: A 3 × 3 matrix game expressed using a game tree.

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

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

which is based on (10.97).

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* [9]. If there is a continuum of stages, which is briefly considered in Section 13.5, then *differential game theory* is obtained [9, 129, 225, 297].

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?

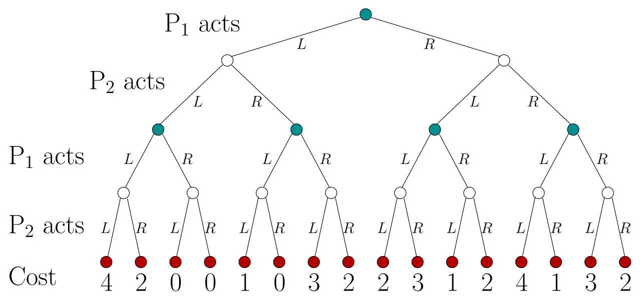


Figure 10.13: A two-player, two-stage game expressed using a game tree.

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 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.

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

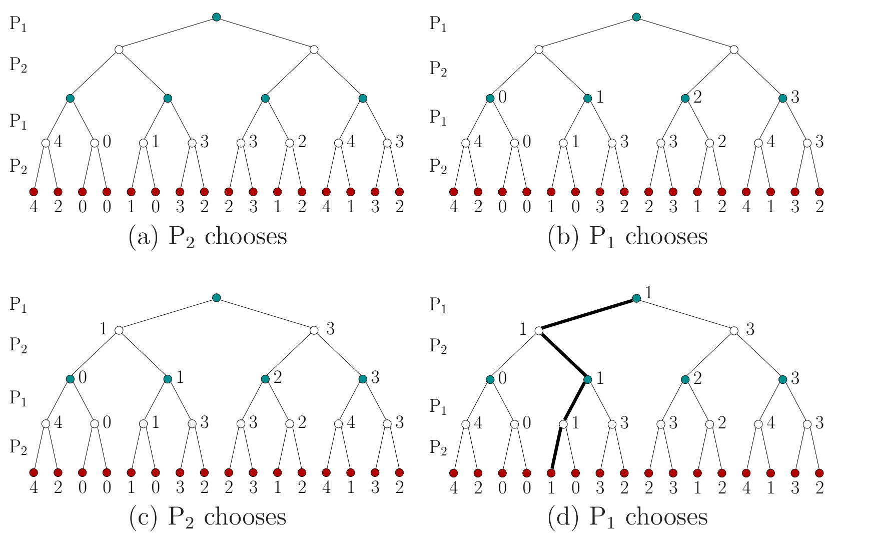


Figure 10.14: The security plan for P$_1$ is determined by propagating costs upward from the leaves. The choices involved in the security plan are shown in the last picture. An upper value of 1 is obtained for the game.

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 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

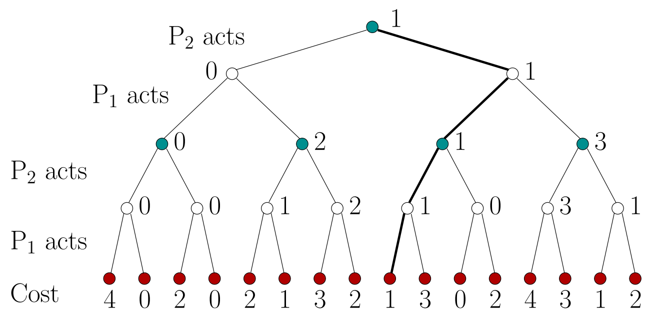


Figure 10.15: The security plan can be found for P$_2$ by swapping the order of P$_1$ and P$_2$ (the order of the costs on the leaves also become reshuffled).

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 [240], 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 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.

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$_1$

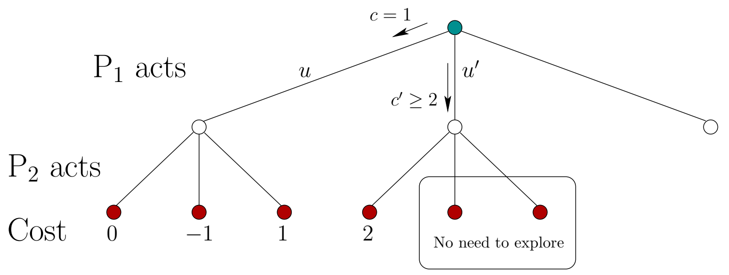


Figure 10.16: If the tree is explored in depth-first order, there are situations in which some children (and hence whole subtrees) do not need to be explored. This is an example that eliminates children of P$_2$. Another case exists, which eliminates children of P$_1$.

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

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

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.

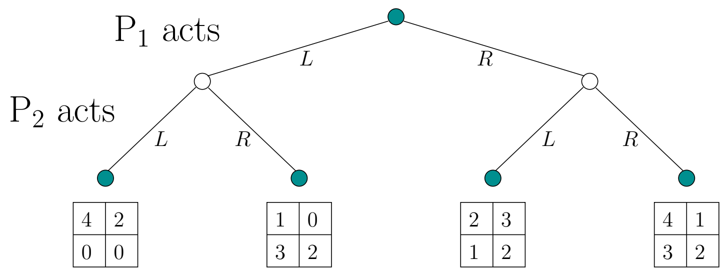


Figure 10.17: Under the stage-by-stage model, the game in Figure 10.13 can instead be represented as a tree in which each player acts once, and then they play a matrix game to determine the cost.

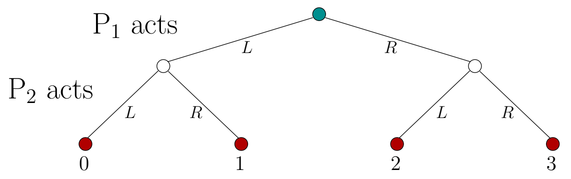


Figure 10.18: Each matrix in Figure 10.17 can be replaced by its randomized value. This clips one level from the original tree. For this particular example, the randomized value is also a deterministic value. Note that these are exactly the costs that appeared in Figure 10.14c. This occurred because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

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:

$$\begin{array}{cc} & \Pi_2 \\ \Pi_1 & \begin{array}{|c|c|c|c|} \hline 4 & 2 & 1 & 0 \\ \hline 0 & 0 & 3 & 2 \\ \hline 2 & 3 & 4 & 1 \\ \hline 1 & 2 & 3 & 2 \\ \hline \end{array} \end{array} . \qquad (10.106)$$

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 $\pi_1 : X \times \mathcal{K} \rightarrow U$ and $\pi_2 : X \times \mathcal{K} \rightarrow V$ 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 $\pi_1 : X \times \mathcal{K} \rightarrow U$, 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 $\pi_2 : X \times \mathcal{K} \rightarrow V$, 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

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

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.

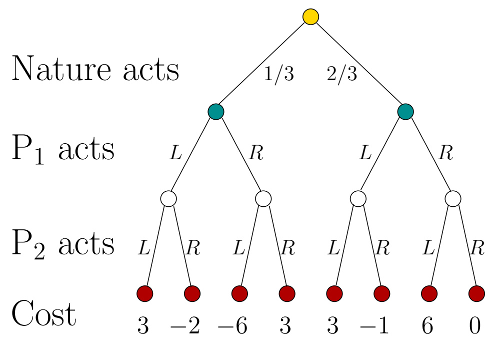


Figure 10.19: This is a single-stage, zero-sum game that involves nature. It is assumed that all players act at the same time.

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 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 [9].

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

$$\begin{array}{cc} & V \\ U & \begin{array}{|c|c|} \hline 3 & -2 \\ \hline -6 & 3 \\ \hline \end{array} \end{array}, \qquad (10.117)$$

and with probability 2/3 it will be

$$\begin{array}{cc} & V \\ U & \begin{array}{|c|c|} \hline 3 & -1 \\ \hline 6 & 0 \\ \hline \end{array} \end{array}. \qquad (10.118)$$

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

$$\begin{array}{cc} & V \\ U & \begin{array}{|c|c|} \hline 3 & 0 \\ \hline 2 & 1 \\ \hline \end{array} \end{array}. \qquad (10.119)$$

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

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

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* [220].

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, ..., 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_k^i denote the action chosen by P_i at stage k. The state transition becomes

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

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.

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 $\theta \in \Theta(x,u)$.

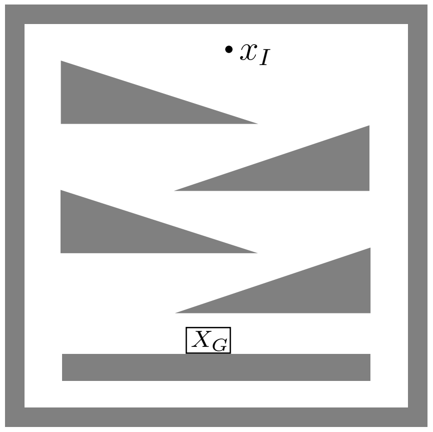
4. A *state transition function* f that is derived from f_m by changing the mode and holding the configuration fixed. Thus, $f((q,m),u,\theta) = (q, f_m(q,m,\theta))$ (the only difference with respect to Formulation 7.3 is that θ has been included).

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

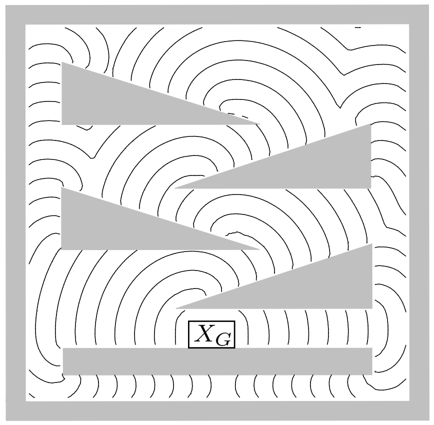
6. A continuous-time cost-functional,

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

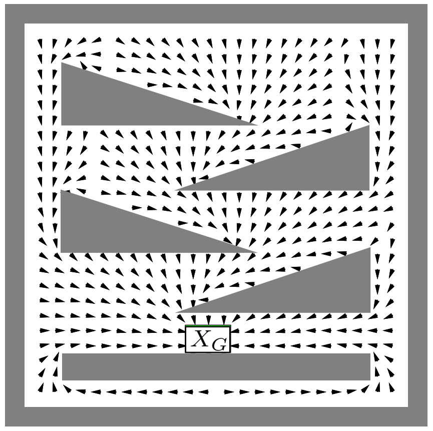
Value iteration proceeds in the same way for such hybrid problems. Interpolation only needs to be performed over the configuration space. Along the mode "axis" no interpolation is needed because the mode set is already finite. The resulting computation time grows linearly in the number of modes. A 2D motion planning



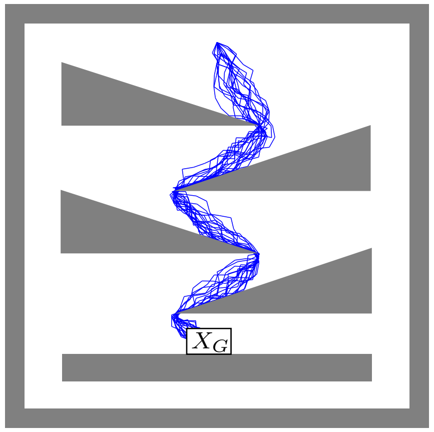
(a) Motion planning game against nature



(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 [166].

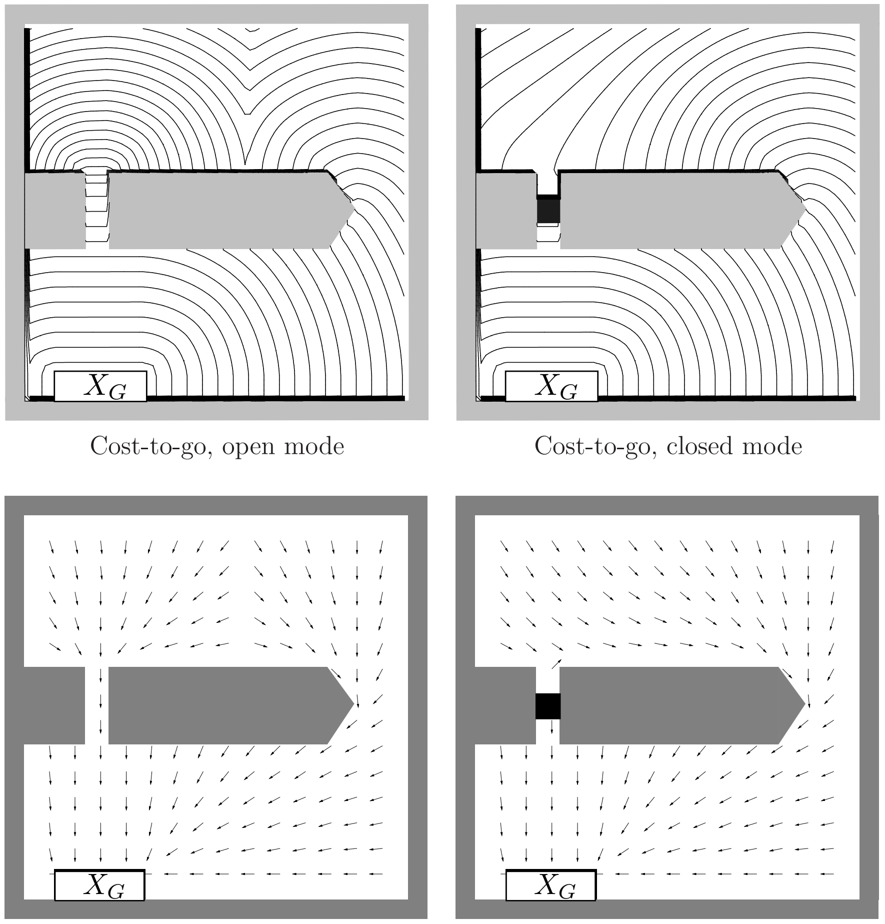


Figure 10.21: Level sets of the optimal navigation function and resulting vector field are shown for a stochastic, hybrid motion planning problem. There are two modes, which correspond to whether a door is closed. The goal is to reach the rectangle at the bottom left [168]

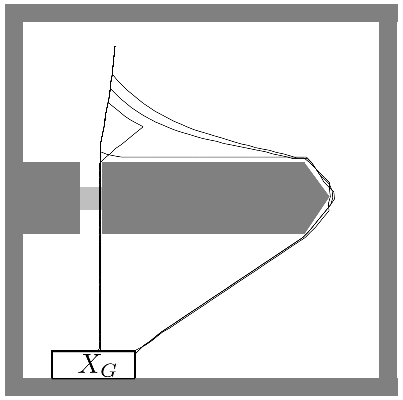


Figure 10.22: Several executions from the same initial state are shown. A different trajectory results each time because of the different times when the door is open or closed.

example for a point robot, taken from [168], 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 [168, 250, 251].

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 [25, 151]. The framework is also popular in artificial intelligence [15, 69, 128, 240]. For an early, influential work on stochastic control, see [31], 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 [75, 92, 177]. The nondeterministic formulation is obtained by eliminating probabilities from the formulation; worst-case analysis also appears extensively in control theory [7, 8, 86]. A case for using

randomized strategies in robotics is made in [93].

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 [25]. Section 10.2.3 is based on extending Dijkstra's algorithm. For convergence issues due to approximations of continuous problems, see [22, 153, 197]. For complexity results for games against nature, see [212, 213].

Section 10.3 was inspired by coverage in [25]. For further reading on reinforcement learning, the subject of Section 10.4, see [3, 13, 27, 262].

Section 10.5 was based on material in [9], but with an emphasis on unifying concepts from previous sections. Also contained in [9] 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 [61].

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 [162, 166, 168, 247, 248].

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.

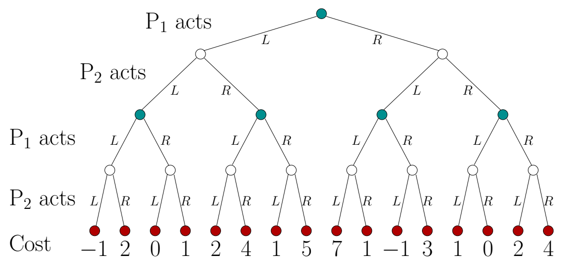


Figure 10.23: A two-player, two-stage game expressed using a game tree.

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 [25].]

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).

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.

Chapter 11

Sensors and Information Spaces

Up until now it has been assumed everywhere that the current state is known. What if the state is not known? In this case, information regarding the state is obtained from sensors during the execution of a plan. This situation arises in most applications that involve interaction with the physical world. For example, in robotics it is virtually impossible for a robot to precisely know its state, except in some limited cases. What should be done if there is limited information regarding the state? A classical approach is to take all of the information available and try to estimate the state. In robotics, the state may include both the map of the robot's environment and the robot configuration. If the estimates are sufficiently reliable, then we may safely pretend that there is no uncertainty in state information. This enables many of the planning methods introduced so far to be applied with little or no adaptation.

The more interesting case occurs when state estimation is altogether avoided. It may be surprising, but many important tasks can be defined and solved without ever requiring that specific states are sensed, even though a state space is defined for the planning problem. To achieve this, the planning problem will be expressed in terms of an *information space*. Information spaces serve the same purpose for sensing problems as the configuration spaces of Chapter 4 did for problems that involve geometric transformations. Each information space represents the place where a problem that involves sensing uncertainty naturally lives. Successfully formulating and solving such problems depends on our ability to manipulate, simplify, and control the information space. In some cases elegant solutions exist, and in others there appears to be no hope at present of efficiently solving them. There are many exciting open research problems associated with information spaces and sensing uncertainty in general.

Recall the situation depicted in Figure 11.1, which was also shown in Section 1.4. It is assumed that the state of the environment is not known. There are three general sources of information regarding the state:

1. The *initial conditions* can provide powerful information before any actions are applied. It might even be the case that the initial state is given. At the

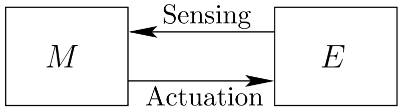


Figure 11.1: The state of the environment is not known. The only information available to make inferences is the history of sensor observations, actions that have been applied, and the initial conditions. This history becomes the *information state.*

other extreme, the initial conditions might contain no information.

2. The *sensor observations* provide measurements related to the state during execution. These measurements are usually incomplete or involve disturbances that distort their values.

3. The *actions* already executed in the plan provide valuable information regarding the state. For example, if a robot is commanded to move east (with no other uncertainties except an unknown state), then it is expected that the state is further east than it was previously. Thus, the applied actions provide important clues for deducing possible states.

Keep in mind that there are generally two ways to use the information space:

1. *Take all of the information available, and try to estimate the state.* This is the classical approach. Pretend that there is no longer any uncertainty in state, but prove (or hope) that the resulting plan works under reasonable estimation error. A plan is generally expressed as $\pi : X \rightarrow U$.

2. *Solve the task entirely in terms of an information space.* Many tasks may be achieved without ever knowing the exact state. The goals and analysis are formulated in the information space, without the need to achieve particular states. For many problems this results in dramatic simplifications. A plan is generally expressed as $\pi : \mathcal{I} \rightarrow U$ for an information space, $\mathcal{I}$.

The first approach may be considered somewhat traditional and can be handled by the concepts of Chapter 8 once a good estimation technique is defined. Most of the focus of the chapter is on the second approach, which represents a powerful way to express and solve planning problems.

For brevity, "information" will be replaced by "I" in many terms. Hence, information spaces and information states become I-spaces and I-states, respectively. This is similar to the shortening of configuration spaces to C-spaces.

Sections 11.1 to 11.3 first cover information spaces for discrete state spaces. This case is much easier to formulate than information spaces for continuous spaces. In Sections 11.4 to 11.6, the ideas are extended from discrete state spaces to continuous state spaces. It is helpful to have a good understanding of the discrete case before proceeding to the continuous case. Section 11.7 extends the formulation of information spaces to game theory, in which multiple players interact over the same state space. In this case, each player in the game has its own information space over which it makes decisions.

11.1 Discrete State Spaces

11.1.1 Sensors

As the name suggests, *sensors* are designed to sense the state. Throughout all of this section it is assumed that the state space, X, is finite or countably infinite, as in Formulations 2.1 and 2.3. A *sensor* is defined in terms of two components: 1) an *observation space*, which is the set of possible readings for the sensor, and 2) a *sensor mapping*, which characterizes the readings that can be expected if the current state or other information is given. Be aware that in the planning model, the state is not really given; it is only assumed to be given when modeling a sensor. The sensing model given here generalizes the one given in Section 9.2.3. In that case, the sensor provided information regarding θ instead of x because state spaces were not needed in Chapter 9.

Let Y denote an *observation space*, which is a finite or countably infinite set. Let h denote the *sensor mapping*. Three different kinds of sensor mappings will be considered, each of which is more complicated and general than the previous one:

1. **State sensor mapping:** In this case, $h : X \to Y$, which means that given the state, the observation is completely determined.

2. **State-nature sensor mapping:** In this case, a finite set, $\Psi(x)$, of *nature sensing actions* is defined for each $x \in X$. Each nature sensing action, $\psi \in \Psi(x)$, interferes with the sensor observation. Therefore, the state-nature mapping, h, produces an observation, $y = h(x, \psi) \in Y$, for every $x \in X$ and $\psi \in \Psi(x)$. The particular ψ chosen by nature is assumed to be unknown during planning and execution. However, it is specified as part of the sensing model.

3. **History-based sensor mapping:** In this case, the observation could be based on the current state or any previous states. Furthermore, a nature sensing action could be applied. Suppose that the current stage is k. The set of nature sensing actions is denoted by $\Psi_k(x)$, and the particular nature sensing action is $\psi_k \in \Psi_k(x)$. This yields a very general sensor mapping,

$$y_k = h_k(x_1, \ldots, x_k, \psi_k), \tag{11.1}$$

in which y_k is the observation obtained in stage k. Note that the mapping is denoted as h_k because the domain is different for each k. In general, any of the sensor mappings may be stage-dependent, if desired.

Many examples of sensors will now be given. These are provided to illustrate the definitions and to provide building blocks that will be used in later examples of I-spaces. Examples 11.1 to 11.6 all involve state sensor mappings.

Example 11.1 (Odd/Even Sensor) Let $X = \mathbb{Z}$, the set of integers, and let $Y = \{0, 1\}$. The sensor mapping is

$$y = h(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ 1 & \text{if } x \text{ is odd.} \end{cases} \tag{11.2}$$

The limitation of this sensor is that it only tells whether $x \in X$ is odd or even. When combined with other information, this might be enough to infer the state, but in general it provides incomplete information. ■

Example 11.2 (Mod Sensor) Example 11.1 can be easily generalized to yield the remainder when x is divided by k for some fixed integer k. Let $X = \mathbb{Z}$, and let $Y = \{0, 1, \ldots, k-1\}$. The sensor mapping is

$$y = h(x) = x \bmod k. \tag{11.3}$$

■

Example 11.3 (Sign Sensor) Let $X = \mathbb{Z}$, and let $Y = \{-1, 0, 1\}$. The sensor mapping is

$$y = h(x) = \operatorname{sgn} x. \tag{11.4}$$

This sensor provides very limited information because it only indicates on which side of the boundary $x = 0$ the state may lie. It can, however, precisely determine whether $x = 0$. ■

Example 11.4 (Selective Sensor) Let $X = \mathbb{Z} \times \mathbb{Z}$, and let $(i, j) \in X$ denote a state in which $i, j \in \mathbb{Z}$. Suppose that only the first component of (i, j) can be observed. This yields the sensor mapping

$$y = h(i, j) = i. \tag{11.5}$$

An obvious generalization can be made for any state space that is formed from Cartesian products. The sensor may reveal the values of one or more components, and the rest remain hidden. ■

Example 11.5 (Bijective Sensor) Let X be any state space, and let $Y = X$. Let the sensor mapping be any bijective function $h : X \to Y$. This sensor provides information that is equivalent to knowing the state. Since h is bijective, it can be inverted to obtain $h^{-1} : Y \rightarrow X$. For any $y \in Y$, the state can be determined as $x = h^{-1}(y)$.

A special case of the *bijective sensor* is the *identity sensor*, for which h is the identity function. This was essentially assumed to exist for all planning problems covered before this chapter because it immediately yields the state. However, any bijective sensor could serve the same purpose. ■

Example 11.6 (Null Sensor) Let X be any state space, and let $Y = \{0\}$. The *null sensor* is obtained by defining the sensor mapping as $h(x) = 0$. The sensor reading remains fixed and hence provides no information regarding the state. ■

From the examples so far, it is tempting to think about partitioning X based on sensor observations. Suppose that in general a state mapping, h, is not bijective, and let $H(y)$ denote the following subset of X:

$$H(y) = \{x \in X \mid y = h(x)\}, \tag{11.6}$$

which is the *preimage* of y. The set of preimages, one for each $y \in Y$, forms a partition of X. In some sense, this indicates the "resolution" of the sensor. A bijective sensor partitions X into singleton sets because it contains perfect information. At the other extreme, the null sensor partitions X into a single set, X itself. The sign sensor appears slightly more useful because it partitions X into three sets: $H(1) = \{1, 2, \ldots\}$, $H(-1) = \{\ldots, -2, -1\}$, and $H(0) = \{0\}$. The preimages of the selective sensor are particularly interesting. For each $i \in \mathbb{Z}$, $H(i) = \mathbb{Z}$. The partitions induced by the preimages may remind those with an algebra background of the construction of quotient groups via homomorphisms [215].

Next consider some examples that involve a state-action sensor mapping. There are two different possibilities regarding the model for the nature sensing action:

1. **Nondeterministic:** In this case, there is no additional information regarding which $\psi \in \Psi(x)$ will be chosen.

2. **Probabilistic:** A probability distribution is known. In this case, the probability, $P(\psi|x)$, that ψ will be chosen is known for each $\psi \in \Psi(x)$.

These two possibilities also appeared in Section 10.1.1, for nature actions that interfere with the state transition equation.

It is sometimes useful to consider the state-action sensor model as a probability distribution over Y for a given state. Recall the conversion from $P(\psi|\theta)$ to $P(y|\theta)$ in (9.28). By replacing Θ by X, the same idea can be applied here. Assume that

if the domain of h is restricted to some $x \in X$, it forms an injective (one-to-one) mapping from Ψ to Y. In this case,

$$P(y|x) = \begin{cases} P(\psi|x) & \text{for the unique } \psi \text{ such that } y = h(x,\psi). \\ 0 & \text{if no such } \psi \text{ exists.} \end{cases} \tag{11.7}$$

If the injective assumption is lifted, then $P(\psi|x)$ is replaced by a sum over all ψ for which $y = h(x, \psi)$.

Example 11.7 (Sensor Disturbance) Let $X = \mathbb{Z}$, $Y = \mathbb{Z}$, and $\Psi = \{-1, 0, 1\}$. The idea is to construct a sensor that would be the identity sensor if it were not for the interference of nature. The sensor mapping is

$$y = h(x, \psi) = x + \psi. \tag{11.8}$$

It is always known that $|x - y| \leq 1$. Therefore, if y is received as a sensor reading, one of the following must be true: $x = y - 1$, $x = y$, or $x = y + 1$. ■

Example 11.8 (Disturbed Sign Sensor) Let $X = \mathbb{Z}$, $Y = \{-1, 0, 1\}$, and $\Psi = \{-1, 0, 1\}$. Let the sensor mapping be

$$y = h(x, \psi) = \operatorname{sgn}(x + \psi). \tag{11.9}$$

In this case, if $y = 0$, it is no longer known for certain whether $x = 0$. It is possible that $x = -1$ or $x = 1$. If $x = 0$, then it is possible for the sensor to read -1, 0, or 1. ■

Example 11.9 (Disturbed Odd/Even Sensor) It is not hard to construct examples for which some mild interference from nature destroys all of the information. Let $X = \mathbb{Z}$, $Y = \{0, 1\}$, and $\Psi = \{0, 1\}$. Let the sensor mapping be

$$y = h(x, \psi) = \begin{cases} 0 & \text{if } x + \psi \text{ is even.} \\ 1 & \text{if } x + \psi \text{ is odd.} \end{cases} \tag{11.10}$$

Under the nondeterministic model for the nature sensing action, the sensor provides no useful information regarding the state. Regardless of the observation, it is never known whether x is even or odd. Under a probabilistic model, however, this sensor may provide some useful information. ■

It is once again informative to consider preimages. For a state-action sensor mapping, the preimage is

$$H(y) = \{x \in X \mid \exists \psi \in \Psi(x) \text{ for which } y = h(x, \psi)\}. \tag{11.11}$$

$$\begin{tikzcd}
y_1 & & y_2 & & y_3 & \\
x_1 \arrow[u] \arrow[r] & u_1 \arrow[r] & x_2 \arrow[u] \arrow[r] & u_2 \arrow[r] & x_3 \arrow[u] \arrow[r] & \cdots
\end{tikzcd}$$

Figure 11.2: In each stage, k, an observation, $y_k \in Y$, is received and an action $u_k \in U$ is applied. The state, x_k, however, is hidden from the decision maker.

In comparison to state sensor mappings, the preimage sets are larger for state-action sensor mappings. Also, they do not generally form a partition of X. For example, the preimages of Example 11.8 are $H(1) = \{0, 1, \ldots\}$, $H(0) = \{-1, 0, 1\}$, and $H(-1) = \{\ldots, -2, -1, 0\}$. This is not a partition because every preimage contains 0. If desired, $H(y)$ can be directly defined for each $y \in Y$, instead of explicitly defining nature sensing actions.

Finally, one example of a history-based sensor mapping is given.

Example 11.10 (Delayed-Observation Sensor) Let $X = Y = \mathbb{Z}$. A *delayed-observation sensor* can be defined for some fixed positive integer i as $y_k = x_{k-i}$. It indicates what the state was i stages ago. In this case, it gives a perfect measurement of the old state value. Many other variants are possible. For example, it might only give the sign of the state from i stages ago. ■

11.1.2 Defining the History Information Space

This section defines the most basic and natural I-space. Many others will be derived from it, which is the topic of Section 11.2. Suppose that X, U, and f have been defined as in Formulation 10.1, and the notion of stages has been defined as in Formulation 2.2. This yields a state sequence x_1, x_2, ..., and an action sequence u_1, u_2, ..., during the execution of a plan. However, in the current setting, the state sequence is not known. Instead, at every stage, an observation, y_k, is obtained. The process depicted in Figure 11.2.

In previous formulations, the action space, $U(x)$, was generally allowed to depend on x. Since x is unknown in the current setting, it would seem strange to allow the actions to depend on x. This would mean that inferences could be made regarding the state by simply noticing which actions are available.[1] Instead, it will be assumed by default that U is fixed for all $x \in X$. In some special contexts, however, $U(x)$ may be allowed to vary.

Initial conditions As stated at the beginning of the chapter, the initial conditions provide one of the three general sources of information regarding the state. Therefore, three alternative types of initial conditions will be allowed:

[1]Such a problem could be quite interesting to study, but it will not be considered here.

1. **Known State:** The initial state, $x_1 \in X$, is given. This initializes the problem with perfect state information. Assuming nature actions interfere with the state transition function, f, uncertainty in the current state will generally develop.

2. **Nondeterministic:** A set of states, $X_1 \subset X$, is given. In this case, the initial state is only known to lie within a particular subset of X. This can be considered as a generalization of the first type, which only allowed singleton subsets.

3. **Probabilistic:** A probability distribution, $P(x_1)$, over X is given.

In general, let η_0 denote the initial condition, which may be any one of the three alternative types.

History Suppose that the kth stage has passed. What information is available? It is assumed that at every stage, a sensor observation is made. This results in a *sensing history*,

$$\tilde{y}_k = (y_1, y_2, \ldots, y_k). \tag{11.12}$$

At every stage an action can also be applied, which yields an *action history*,

$$\tilde{u}_{k-1} = (u_1, u_2, \ldots, u_{k-1}). \tag{11.13}$$

Note that the action history only runs to u_{k-1}; if u_k is applied, the state x_{k+1} and stage $k+1$ are obtained, which lie beyond the current stage, k. By combining the sensing and action histories, the *history* at stage k is $(\tilde{u}_{k-1}, \tilde{y}_k)$.

History information states The history, $(\tilde{u}_{k-1}, \tilde{y}_k)$, in combination with the initial condition, η_0, yields the *history I-state*, which is denoted by η_k. This corresponds to all information that is known up to stage k. In spite of the fact that the states, x_1, ..., x_k, might not be known, the history I-states are always known because they are defined directly in terms of available information. Thus, the history I-state is

$$\eta_k = (\eta_0, \tilde{u}_{k-1}, \tilde{y}_k). \tag{11.14}$$

When representing I-spaces, we will generally ignore the problem of nesting parentheses. For example, (11.14) is treated a single sequence, instead of a sequence that contains two sequences. This distinction is insignificant for the purposes of decision making.

The history I-state, η_k, can also be expressed as

$$\eta_k = (\eta_{k-1}, u_{k-1}, y_k), \tag{11.15}$$

by noticing that the history I-state at stage k contains all of the information from the history I-state at stage $k-1$. The only new information is the most recently applied action, u_{k-1}, and the current sensor observation, y_k.

The history information space The history I-space is simply the set of all possible history I-states. Although the history I-states appear to be quite complicated, it is helpful to think of them abstractly as points in a new space. To define the set of all possible history I-states, the sets of all initial conditions, actions, and observations must be precisely defined.

The set of all observation histories is denoted as $\tilde{Y}_k$ and is obtained by a Cartesian product of k copies of the observation space:

$$\tilde{Y}_k = \underbrace{Y \times Y \ldots \times Y}_{k}. \tag{11.16}$$

Similarly, the set of all action histories is $\tilde{U}_{k-1}$, the Cartesian product of $k-1$ copies of the action space U.

It is slightly more complicated to define the set of all possible initial conditions because three different types of initial conditions are possible. Let $\mathcal{I}_0$ denote the *initial condition space*. Depending on which of the three types of initial conditions are used, one of the following three definitions of $\mathcal{I}_0$ is used:

1. **Known State:** If the initial state, x_1, is given, then $\mathcal{I}_0 \subseteq X$. Typically, $\mathcal{I}_0 = X$; however, it might be known in some instances that certain initial states are impossible. Therefore, it is generally written that $\mathcal{I}_0 \subseteq X$.

2. **Nondeterministic:** If X_1 is given, then $\mathcal{I}_0 \subseteq \text{pow}(X)$ (the power set of X). Again, a typical situation is $\mathcal{I}_0 = \text{pow}(x)$; however, it might be known that certain subsets of X are impossible as initial conditions.

3. **Probabilistic:** Finally, if $P(x)$ is given, then $\mathcal{I}_0 \subseteq \mathcal{P}(X)$, in which $\mathcal{P}(x)$ is the set of all probability distributions over X.

The *history I-space at stage* k is expressed as

$$\mathcal{I}_k = \mathcal{I}_0 \times \tilde{U}_{k-1} \times \tilde{Y}_k. \tag{11.17}$$

Each $\eta_k \in \mathcal{I}_k$ yields an initial condition, an action history, and an observation history. It will be convenient to consider I-spaces that do not depend on k. This will be defined by taking a union (be careful not to mistakenly think of this construction as a Cartesian product). If there are K stages, then the *history I-space* is

$$\mathcal{I}_{hist} = \mathcal{I}_0 \cup \mathcal{I}_1 \cup \mathcal{I}_2 \cup \cdots \cup \mathcal{I}_K. \tag{11.18}$$

Most often, the number of stages is not fixed. In this case, $\mathcal{I}_{hist}$ is defined to be the union of $\mathcal{I}_k$ over all $k \in \{0\} \cup \mathbb{N}$:

$$\mathcal{I}_{hist} = \mathcal{I}_0 \cup \mathcal{I}_1 \cup \mathcal{I}_2 \cup \cdots . \tag{11.19}$$

This construction is related to the state space obtained for time-varying motion planning in Section 7.1. The history I-space is stage-dependent because information accumulates over time. In the discrete model, the reference to time is

only implicit through the use of stages. Therefore, stage-dependent I-spaces are defined. Taking the union of all of these is similar to the state space that was formed in Section 7.1 by making time be one axis of the state space. For the history I-space, $\mathcal{I}_{hist}$, the stage index k can be imagined as an "axis."

One immediate concern regarding the history I-space $\mathcal{I}_{hist}$ is that its I-states may be arbitrarily long because the history grows linearly with the number of stages. For now, it is helpful to imagine $\mathcal{I}_{hist}$ abstractly as another kind of state space, without paying close attention to how complicated each $\eta \in \mathcal{I}_{hist}$ may be to represent. In many contexts, there are ways to simplify the I-space. This is the topic of Section 11.2.

11.1.3 Defining a Planning Problem

Planning problems will be defined directly on the history I-space, which makes it appear as an ordinary state space in many ways. Keep in mind, however, that it was derived from another state space for which perfect state observations could not be obtained. In Section 10.1, a feedback plan was defined as a function of the state. Here, a feedback plan is instead a function of the I-state. Decisions cannot be based on the state because it will be generally unknown during the execution of the plan. However, the I-state is always known; thus, it is logical to base decisions on it.

Let π_K denote a K*-step information-feedback plan*, which is a sequence $(\pi_1, \pi_2, \ldots, \pi_K)$ of K functions, $\pi_k : \mathcal{I}_k \rightarrow U$. Thus, at every stage k, the I-state $\eta_k \in \mathcal{I}_k$ is used as a basis for choosing the action $u_k = \pi_k(\eta_k)$. Due to interference of nature through both the state transition equation and the sensor mapping, the action sequence $(u_1, \ldots, u_K)$ produced by a plan, π_K, will not be known until the plan terminates.

As in Formulation 2.3, it will be convenient to assume that U contains a *termination action*, u_T. If u_T is applied at stage k, then it is repeatedly applied forever. It is assumed once again that the state x_k remains fixed after the termination condition is applied. Remember, however, x_k is still unknown in general; it becomes fixed but unknown. Technically, based on the definition of the history I-space, the I-state must change after u_T is applied because the history grows. These changes can be ignored, however, because no new decisions are made after u_T is applied. A plan that uses a termination condition can be specified as $\pi = (\pi_1, \pi_2, \ldots)$ because the number of stages may vary each time the plan is executed. Using the history I-space definition in (11.19), an *information-feedback plan* is expressed as

$$\pi : \mathcal{I}_{hist} \rightarrow U. \tag{11.20}$$

We are almost ready to define the planning problem. This will require the specification of a cost functional. The cost depends on the histories of states $\tilde{x}$ and actions $\tilde{u}$ as in Section 10.1. The planning formulation involves the following components, summarizing most of the concepts introduced so far in Section 11.1 (see Formulation 10.1 for similarities):

Formulation 11.1 (Discrete Information Space Planning)

1. A nonempty *state space* X that is either finite or countably infinite.
2. A nonempty, finite *action space* U. It is assumed that U contains a special *termination action*, which has the same effect as defined in Formulation 2.3.
3. A finite *nature action space* $\Theta(x,u)$ for each $x \in X$ and $u \in U$.
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 finite or countably infinite *observation space* Y.
6. A finite *nature sensing action space* $\Psi(x)$ for each $x \in X$.
7. A *sensor mapping* h which produces an observation, $y = h(x,\psi)$, for each $x \in X$ and $\psi \in \Psi(x)$. This definition assumes a state-nature sensor mappings. A state sensor mapping or history-based sensor mapping, as defined in Section 11.1.1, could alternatively be used.
8. A set of *stages*, each denoted by k, which begins at $k = 1$ and continues indefinitely.
9. An *initial condition* η_0, which is an element of an *initial condition space*, $\mathcal{I}_0$.
10. A *history I-space* $\mathcal{I}_{hist}$ which is the union of $\mathcal{I}_0$ and $\mathcal{I}_k = \mathcal{I}_0 \times \tilde{U}_{k-1} \times \tilde{Y}_k$ for every stage $k \in \mathbb{N}$.
11. Let L denote a stage-additive cost functional, which may be applied to any pair $(\tilde{x}_{K+1}, \tilde{u}_K)$ of state and action histories to yield
$$L(\tilde{x}_{K+1}, \tilde{u}_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_{K+1}). \qquad (11.21)$$
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 $l(x_i, u_T) = 0$. Either a feasible or optimal planning problem can be defined, as in Formulation 10.1; however, the plan here is specified as $\pi : \mathcal{I} \rightarrow U$.

A *goal set* may be defined as $X_G \subset X$. Alternatively, the goal could be expressed as a desirable set of history I-states. After Section 11.2, it will be seen that the goal can be expressed in terms of I-states that are derived from histories.

Some immediate extensions of Formulation 11.1 are possible, but we avoid them here simplify notation in the coming concepts. One extension is to allow different action sets, $U(x)$, for each $x \in X$. Be careful, however, because information regarding the current state can be inferred if the action set $U(x)$ is given, and it varies depending on x. Another extension is to allow the costs to depend on nature, to obtain $l(x_k, u_k, \theta_k)$, instead of $l(x_k, u_k)$ in (11.21).

The cost of a plan The next task is to extend the definition of the cost-to-go under a fixed plan, which was given in Section 10.1.3, to the case of imperfect state information. Consider evaluating the quality of a plan, so that the "best" one might be selected. Suppose that the nondeterministic uncertainty is used to model nature and that a nondeterministic initial condition is given. If a plan π is fixed, some state and action trajectories are possible, and others are not. It is impossible to know in general what histories will occur; however, the plan constrains the choices substantially. Let $\mathcal{H}(\pi, \eta_0)$ denote the set of state-action histories that could arise from π applied to the initial condition η_0.

The cost of a plan π from an initial condition η_0 is measured using *worst-case analysis* as
$$G_\pi(\eta_0) = \max_{(\tilde{x},\tilde{u}) \in \mathcal{H}(\pi,\eta_0)} \Big\{ L(\tilde{x}, \tilde{u}) \Big\}. \qquad (11.22)$$

Note that $\tilde{x}$ includes x_1, which is usually not known. It may be known only to lie in X_1, as specified by η_0. Let Π denote the set of all possible plans. An optimal plan using worst-case analysis is any plan for which (11.22) is minimized over all $\pi \in \Pi$ and $\eta_0 \in \mathcal{I}_0$. In the case of feasible planning, there are usually numerous equivalent alternatives.

Under probabilistic uncertainty, the cost of a plan can be measured using *expected-case analysis* as
$$G_\pi(\eta_0) = E_{\mathcal{H}(\pi,\eta_0)} \Big[L(\tilde{x}, \tilde{u}) \Big], \qquad (11.23)$$
in which E denotes the mathematical expectation of the cost, with the probability distribution taken over $\mathcal{H}(\pi, \eta_0)$. The task is to find a plan $\pi \in \Pi$ that minimizes (11.23).

The information space is just another state space It will become important throughout this chapter and Chapter 12 to view the I-space as an ordinary state space. It only seems special because it is derived from another state space, but once this is forgotten, it exhibits many properties of an ordinary state space in planning. One nice feature is that the state in this special space is always known. Thus, by converting from an original state space to its I-space, we also convert from having imperfect state information to always knowing the state, albeit in a larger state space.

One important consequence of this interpretation is that the state transition equation can be lifted into the I-space to obtain an *information transition function*, $f_{\mathcal{I}}$. Suppose that there are no sensors, and therefore no observations. In this case, future I-states are predictable, which leads to
$$\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k). \qquad (11.24)$$
The function $f_{\mathcal{I}}$ generates η_{k+1} by concatenating u_k onto η_k.

Now suppose that there are observations, which are generally unpredictable. In Section 10.1, the nature action $\theta_k \in \Theta(x, u)$ was used to model the unpredictability. In terms of the information transition equation, y_{k+1} serves the same purpose. When the decision is made to apply u_k, the observation y_{k+1} is not yet known (just as θ_k is unknown in Section 10.1). In a sequential game against nature with perfect state information, x_{k+1} is directly observed at the next stage. For the information transition equation, y_{k+1} is instead observed, and η_{k+1} can be determined. Using the history I-state representation, (11.14), simply concatenate u_k and y_{k+1} onto the histories in η_k to obtain η_{k+1}. The information transition equation is expressed as

$$\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k, y_{k+1}), \tag{11.25}$$

with the understanding that y_{k+1} plays the same role as θ_k in the case of perfect state information and unpredictable future states. Even though nature causes future I-states to be unpredictable, the current I-state is always known. A plan, $\pi : \mathcal{I} \rightarrow U$, now seems like a state-feedback plan, if the I-space is viewed as a state space. The transitions are all specified by $f_{\mathcal{I}}$.

The costs in this new state space can be derived from the original cost functional, but a maximization or expectation is needed over all possible states given the current information. This will be covered in Section 12.1.

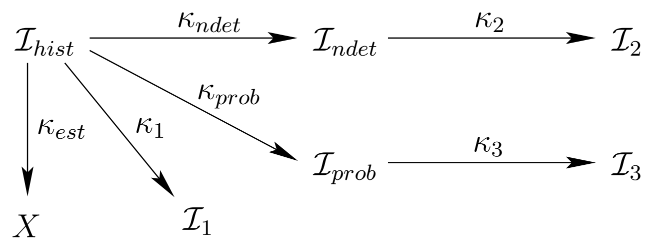
11.2 Derived Information Spaces

The history I-space appears to be quite complicated. Every I-state corresponds to a history of actions and observations. Unfortunately, the length of the I-state sequence grows linearly with the number of stages. To overcome this difficultly, it is common to map history I-states to some simpler space. In many applications, the ability to perform this simplification is critical to finding a practical solution. In some cases, the simplification fully preserves the history I-space, meaning that completeness, and optimality if applicable, is not lost. In other cases, we are willing to tolerate a simplification that destroys much of the structure of the history I-space. This may be necessary to obtain a dramatic reduction in the size of the I-space.

11.2.1 Information Mappings

Consider a function that maps the history I-space into a space that is simpler to manage. Formally, let $\kappa : \mathcal{I}_{hist} \rightarrow \mathcal{I}_{der}$ denote a function from a history I-space, $\mathcal{I}_{hist}$, to a *derived I-space*, $\mathcal{I}_{der}$. The function, κ, is called an *information mapping*, or *I-map*. The derived I-space may be any set; hence, there is great flexibility in defining an I-map.[2] Figure 11.3 illustrates the idea. The starting place is $\mathcal{I}_{hist}$,

[2]Ideally, the mapping should be *onto* $\mathcal{I}_{der}$; however, to facilitate some definitions, this will not be required.



$$\begin{tikzcd}
\mathcal{I}_{hist} \arrow[rr, "\kappa_{ndet}"] \arrow[dd, "\kappa_{est}"'] \arrow[ddr, "\kappa_1"] \arrow[drr, "\kappa_{prob}"] & & \mathcal{I}_{ndet} \arrow[r, "\kappa_2"] & \mathcal{I}_2 \\
& & \mathcal{I}_{prob} \arrow[r, "\kappa_3"] & \mathcal{I}_3 \\
X & \mathcal{I}_1 & &
\end{tikzcd}$$

Figure 11.3: Many alternative information mappings may be proposed. Each leads to a derived information space.

and mappings are made to various derived I-spaces. Some generic mappings, κ_1, κ_2, and κ_3, are shown, along with some very important kinds, $\mathcal{I}_{est}$, $\mathcal{I}_{ndet}$ and $\mathcal{I}_{prob}$. The last two are the subjects of Sections 11.2.2 and 11.2.3, respectively. The other important I-map is κ_{est}, which uses the history to estimate the state; hence, the derived I-space is X (see Example 11.11). In general, an I-map can even map any derived I-space to another, yielding $\kappa : \mathcal{I}_{der} \rightarrow \mathcal{I}'_{der}$, for any I-spaces $\mathcal{I}_{der}$ and $\mathcal{I}'_{der}$. Note that any composition of I-maps yields an I-map. The derived I-spaces $\mathcal{I}_2$ and $\mathcal{I}_3$ from Figure 11.3 are obtained via compositions.

Making smaller information-feedback plans The primary use of an I-map is to simplify the description of a plan. In Section 11.1.3, a plan was defined as a function on the history I-space, $\mathcal{I}_{hist}$. Suppose that an I-map, κ, is introduced that maps from $\mathcal{I}_{hist}$ to $\mathcal{I}_{der}$. A feedback plan on $\mathcal{I}_{der}$ is defined as $\pi : \mathcal{I}_{der} \rightarrow U$. To execute a plan defined on $\mathcal{I}_{der}$, the derived I-state is computed at each stage k by applying κ to η_k to obtain $\kappa(\eta_k) \in \mathcal{I}_{der}$. The action selected by π is $\pi(\kappa(\eta_k)) \in U$.

To understand the effect of using $\mathcal{I}_{der}$ instead of $\mathcal{I}_{hist}$ as the domain of π, consider the set of possible plans that can be represented over $\mathcal{I}_{der}$. Let Π_{hist} and Π_{der} be the sets of all plans over $\mathcal{I}_{hist}$ and $\mathcal{I}_{der}$, respectively. Any $\pi \in \Pi_{der}$ can be converted into an equivalent plan, $\pi' \in \Pi_{hist}$, as follows: For each $\eta \in \mathcal{I}_{hist}$, define $\pi'(\eta) = \pi(\kappa(\eta))$.

It is not always possible, however, to construct a plan, $\pi \in \Pi_{der}$, from some $\pi' \in \mathcal{I}_{hist}$. The problem is that there may exist some $\eta_1, \eta_2 \in \mathcal{I}_{hist}$ for which $\pi'(\eta_1) \neq \pi'(\eta_2)$ and $\kappa(\eta_1) = \kappa(\eta_2)$. In words, this means that the plan in Π_{hist} requires that two histories cause different actions, but in the derived I-space the histories cannot be distinguished. For a plan in Π_{der}, both histories must yield the same action.

An I-map κ has the potential to collapse $\mathcal{I}_{hist}$ down to a smaller I-space by inducing a partition of $\mathcal{I}_{hist}$. For each $\eta_{der} \in \mathcal{I}_{der}$, let the preimage $\kappa^{-1}(\eta_{der})$ be defined as

$$\kappa^{-1}(\eta_{der}) = \{\eta \in \mathcal{I}_{hist} \mid \eta_{der} = \kappa(\eta)\}. \tag{11.26}$$

This yields the set of history I-states that map to η_{der}. The induced partition can intuitively be considered as the "resolution" at which the history I-space is characterized. If the sets in (11.26) are large, then the I-space is substantially reduced. The goal is to select κ to make the sets in the partition as large as possible; however, one must be careful to avoid collapsing the I-space so much that the problem can no longer be solved.

Example 11.11 (State Estimation) In this example, the I-map is the classical approach that is conveniently taken in numerous applications. Suppose that a technique has been developed that uses the history I-state $\eta \in \mathcal{I}_{hist}$ to compute an estimate of the current state. In this case, the I-map is $\kappa_{est} : \mathcal{I}_{hist} \rightarrow X$. The derived I-space happens to be X in this case! This means that a plan is specified as $\pi : X \rightarrow U$, which is just a state-feedback plan.

Consider the partition of $\mathcal{I}_{hist}$ that is induced by κ_{est}. For each $x \in X$, the set $\kappa_{est}^{-1}(x)$, as defined in (11.26), is the set of all histories that lead to the same state estimate. A plan on X can no longer distinguish between various histories that led to the same state estimate. One implication is that the ability to encode the amount of uncertainty in the state estimate has been lost. For example, it might be wise to make the action depend on the covariance in the estimate of x; however, this is not possible because decisions are based only on the estimate itself. ■

Example 11.12 (Stage Indices) Consider an I-map, κ_{stage}, that returns only the current stage index. Thus, $\kappa_{stage}(\eta_k) = k$. The derived I-space is the set of stages, which is $\mathbb{N}$. A feedback plan on the derived I-space is specified as $\pi : \mathbb{N} \rightarrow U$. This is equivalent to specifying a plan as an action sequence, $(u_1, u_2, \ldots,)$, as in Section 2.3.2. Since the feedback is trivial, this is precisely the original case of planning without feedback, which is also refereed to as an open-loop plan. ■

Constructing a derived information transition equation As presented so far, the full history I-state is needed to determine a derived I-state. It may be preferable, however, to discard histories and work entirely in the derived I-space. Without storing the histories on the machine or robot, a derived information transition equation needs to be developed. The important requirement in this case is as follows:

If η_k is replaced by $\kappa(\eta_k)$, then $\kappa(\eta_{k+1})$ must be correctly determined using only $\kappa(\eta_k)$, u_k, and y_{k+1}.

Whether this requirement can be met depends on the particular I-map. Another way to express the requirement is that if $\kappa(\eta_k)$ is given, then the full history η does not contain any information that could further constrain $\kappa(\eta_{k+1})$. The information provided by κ is *sufficient* for determining the next derived I-states.

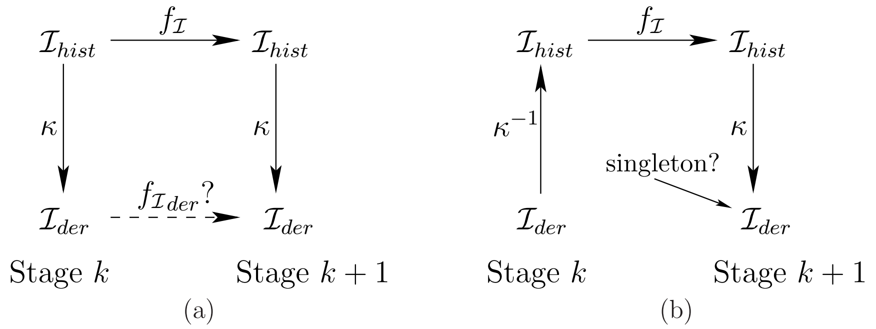


Figure 11.4: (a) For an I-map to be sufficient, the same result must be reached in the lower right, regardless of the path taken from the upper left. (b) The problem is that κ images may contain many histories, which eventually map to multiple derived I-states.

This is similar to the concept of a *sufficient statistic*, which arises in decision theory [21]. If the requirement is met, then κ is called a *sufficient I-map*. One peculiarity is that the sufficiency is relative to $\mathcal{I}_{der}$, as opposed to being absolute in some sense. For example, any I-map that maps onto $\mathcal{I}_{der} = \{0\}$ is sufficient because $\kappa(\eta_{k+1})$ is always known (it remains fixed at 0). Thus, the requirement for sufficiency depends strongly on the particular derived I-space.

For a sufficient I-map, a *derived information transition equation* is determined as

$$\kappa(\eta_{k+1}) = f_{\mathcal{I}_{der}}(\kappa(\eta_k), u_k, y_{k+1}). \tag{11.27}$$

The implication is that $\mathcal{I}_{der}$ is the new I-space in which the problem "lives." There is no reason for the decision maker to consider histories. This idea is crucial to the success of many planning algorithms. Sections 11.2.2 and 11.2.3 introduce nondeterministic I-spaces and probabilistic I-spaces, which are two of the most important derived I-spaces and are obtained from sufficient I-maps. The I-map κ_{stage} from Example 11.12 is also sufficient. The estimation I-map from Example 11.11 is usually not sufficient because some history is needed to provide a better estimate.

The diagram in Figure 11.4a indicates the problem of obtaining a sufficient I-map. The top of the diagram shows the history I-state transitions before the I-map was introduced. The bottom of the diagram shows the attempted derived information transition equation, $f_{\mathcal{I}_{der}}$. The requirement is that the derived I-state obtained in the lower right must be the same regardless of which path is followed from the upper left. Either $f_{\mathcal{I}}$ can be applied to η, followed by κ, or κ can be applied to η, followed by some $f_{\mathcal{I}_{der}}$. The problem with the existence of $f_{\mathcal{I}_{der}}$ is that κ is usually not invertible. The preimage $\kappa^{-1}(\eta_{der})$ of some derived I-state $\eta_{der} \in \mathcal{I}_{der}$ yields a set of histories in $\mathcal{I}_{hist}$. Applying $f_{\mathcal{I}}$ to all of these yields a set of possible next-stage history I-states. Applying κ to these may yield a set of derived I-states because of the ambiguity introduced by κ^{-1}. This chain of mappings is shown in Figure 11.4b. If a singleton is obtained under all circumstances, then this yields the required values of $f_{\mathcal{I}_{der}}$. Otherwise, new uncertainty arises about the current derived I-state. This could be handled by defining an information space over the information space, but this nastiness will be avoided here.

Since I-maps can be defined from any derived I-space to another, the concepts presented in this section do not necessarily require $\mathcal{I}_{hist}$ as the starting point. For example, an I-map, $\kappa : \mathcal{I}_{der} \to \mathcal{I}'_{der}$, may be called *sufficient with respect to* $\mathcal{I}_{der}$ rather than with respect to $\mathcal{I}_{hist}$.

11.2.2 Nondeterministic Information Spaces

This section defines the I-map κ_{ndet} from Figure 11.3, which converts each history I-state into a subset of X that corresponds to all possible current states. Nature is modeled nondeterministically, which means that there is no information about what actions nature will choose, other than that they will be chosen from Θ and Ψ. Assume that the state-action sensor mapping from Section 11.1.1 is used. Consider what inferences may be drawn from a history I-state, $\eta_k = (\eta_0, \tilde{u}_{k-1}, \tilde{y}_k)$. Since the model does not involve probabilities, let η_0 represent a set $X_1 \subseteq X$. Let $\kappa_{ndet}(\eta_k)$ be the minimal subset of X in which x_k is known to lie given η_k. This subset is referred to as a *nondeterministic I-state*. To remind you that $\kappa_{ndet}(\eta_k)$ is a subset of X, it will now be denoted as $X_k(\eta_k)$. It is important that $X_k(\eta_k)$ be as small as possible while consistent with η_k.

Recall from (11.6) that for every observation y_k, a set $H(y_k) \subseteq X$ of possible values for x_k can be inferred. This could serve as a crude estimate of the nondeterministic I-state. It is certainly known that $X_k(\eta_k) \subseteq H(y_k)$; otherwise, x_k, would not be consistent with the current sensor observation. If we carefully progress from the initial conditions while applying constraints due to the state transition equation, the appropriate subset of $H(y_k)$ will be obtained.

From the state transition function f, define a set-valued function F that yields a subset of X for every $x \in X$ and $u \in U$ as

$$F(x,u) = \{x' \in X \mid \exists \theta \in \Theta(x,u) \text{ for which } x' = f(x,u,\theta)\}. \tag{11.28}$$

Note that both F and H are set-valued functions that eliminate the direct appearance of nature actions. The effect of nature is taken into account in the set that is obtained when these functions are applied. This will be very convenient for computing the nondeterministic I-state.

An inductive process will now be described that results in computing the nondeterministic I-state, $X_k(\eta_k)$, for any stage k. The base case, $k = 1$, of the induction proceeds as

$$X_1(\eta_1) = X_1(\eta_0, y_1) = X_1 \cap H(y_1). \tag{11.29}$$

The first part of the equation replaces η_1 with (η_0, y_1), which is a longer way to write the history I-state. There are not yet any actions in the history. The second part applies set intersection to make consistent the two pieces of information: 1) The initial state lies in X_1, which is the initial condition, and 2) the states in $H(y_1)$ are possible given the observation y_1.

Now assume inductively that $X_k(\eta_k) \subseteq X$ has been computed and the task is to compute $X_{k+1}(\eta_{k+1})$. From (11.15), $\eta_{k+1} = (\eta_k, u_k, y_{k+1})$. Thus, the only new pieces of information are that u_k was applied and y_{k+1} was observed. These will be considered one at a time.

Consider computing $X_{k+1}(\eta_k, u_k)$. If x_k was known, then after applying u_k, the state could lie anywhere within $F(x_k, u_k)$, using (11.28). Although x_k is actually not known, it is at least known that $x_k \in X_k(\eta_k)$. Therefore,

$$X_{k+1}(\eta_k, u_k) = \bigcup_{x_k \in X_k(\eta_k)} F(x_k, u_k). \tag{11.30}$$

This can be considered as the set of all states that can be reached by starting from some state in $X_k(\eta_k)$ and applying any actions $u_k \in U$ and $\theta_k \in \Theta(x_k, u_k)$. See Figure 11.5.

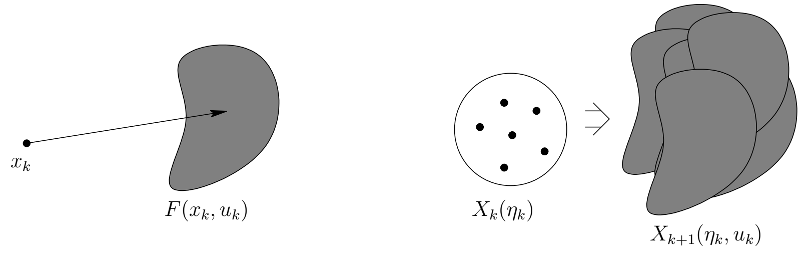


Figure 11.5: The first step in computing the nondeterministic I-state is to take the union of $F(x_k, u_k)$ over all possible $x_k \in X_k(\eta_k)$.

The next step is to take into account the observation y_{k+1}. This information alone indicates that x_{k+1} lies in $H(y_{k+1})$. Therefore, an intersection is performed to obtain the nondeterministic I-state,

$$X_{k+1}(\eta_{k+1}) = X_{k+1}(\eta_k, u_k, y_{k+1}) = X_{k+1}(\eta_k, u_k) \cap H(y_{k+1}). \tag{11.31}$$

Thus, it has been shown how to compute $X_{k+1}(\eta_{k+1})$ from $X_k(\eta_k)$. After starting with (11.29), the nondeterministic I-states at any stage can be computed by iterating (11.30) and (11.31) as many times as necessary.

Since the nondeterministic I-state is always a subset of X, the *nondeterministic I-space*, $\mathcal{I}_{ndet} = \text{pow}(X)$, is obtained (shown in Figure 11.3). If X is finite, then $\mathcal{I}_{ndet}$ is also finite, which was not the case with $\mathcal{I}_{hist}$ because the histories continued to grow with the number of stages. Thus, if the number of stages is unbounded or large in comparison to the size of X, then nondeterministic I-states seem preferable. It is also convenient that κ_{ndet} is a sufficient I-map, as defined in Section 11.2.1. This implies that a planning problem can be completely expressed in terms of $\mathcal{I}_{ndet}$ without maintaining the histories. The goal region, X_G, can be expressed directly as a nondeterministic I-state. In this way, the planning task is to terminate in a nondeterministic I-state, $X_k(\eta_k)$, for which $X_k(\eta_k) \subseteq X_G$.

The sufficiency of κ_{ndet} is obtained because (11.30) and (11.31) show that $X_{k+1}(\eta_{k+1})$ can be computed from $X_k(\eta_k)$, u_k, and y_{k+1}. This implies that a derived information transition equation can be formed. The nondeterministic I-space can also be treated as "just another state space." Although many history I-states may map to the same nondeterministic I-state, it has been assumed for decision-making purposes that particular history is irrelevant, once $X_k(\eta_k)$ is given.

The following example is not very interesting in itself, but it is simple enough to illustrate the concepts.

Example 11.13 (Three-State Example) Let $X = \{0,1,2\}$, $U = \{-1,0,1\}$, and $\Theta(x,u) = \{0,1\}$ for all $x \in X$ and $u \in U$. The state transitions are given by $f(x,u,\theta) = (x+u+\theta) \mod 3$. Regarding sensing, $Y = \{0,1,2,3,4\}$ and $\Psi(x) = \{0,1,2\}$ for all $x \in X$. The sensor mapping is $y = h(x,\psi) = x + \psi$.

The history I-space appears very cumbersome for this example, which only involves three states. The nondeterministic I-space for this example is

$$\mathcal{I}_{ndet} = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{1,2\}, \{0,2\}, \{0,1,2\}\}, \tag{11.32}$$

which is the power set of $X = \{0,1,2\}$. Note, however, that the empty set, $\emptyset$, can usually be deleted from $\mathcal{I}_{ndet}$.[3] Suppose that the initial condition is $X_1 = \{0,2\}$ and that the initial state is $x_1 = 0$. The initial state is unknown to the decision maker, but it is needed to ensure that valid observations are made in the example.

Now consider the execution over a number of stages. Suppose that the first observation is $y_1 = 2$. Based on the sensor mapping, $H(y_1) = H(2) = \{0,1,2\}$, which is not very helpful because $H(2) = X$. Applying (11.29) yields $X_1(\eta_1) = \{0,2\}$. Now suppose that the decision maker applies the action $u_1 = 1$ and nature applies $\theta_1 = 1$. Using f, this yields $x_2 = 2$. The decision maker does not know θ_1 and must therefore take into account any nature action that could have been applied. It uses (11.30) to infer that

$$X_2(\eta_1, u_1) = F(2,1) \cup F(0,1) = \{0,1\} \cup \{1,2\} = \{0,1,2\}. \tag{11.33}$$

Now suppose that $y_2 = 3$. From the sensor mapping, $H(3) = \{1,2\}$. Applying (11.31) yields

$$X_2(\eta_2) = X_2(\eta_1, u_1) \cap H(y_2) = \{0,1,2\} \cap \{1,2\} = \{1,2\}. \tag{11.34}$$

[3]One notable exception is in the theory of nondeterministic finite automata, in which it is possible that all copies of the machine die and there is no possible current state [260].

This process may be repeated for as many stages as desired. A path is generated through $\mathcal{I}_{ndet}$ by visiting a sequence of nondeterministic I-states. If the observation $y_k = 4$ is ever received, the state, x_k, becomes immediately known because $H(4) = \{2\}$. ■

11.2.3 Probabilistic Information Spaces

This section defines the I-map κ_{prob} from Figure 11.3, which converts each history I-state into a probability distribution over X. A Markov, probabilistic model is assumed in the sense that the actions of nature only depend on the current state and action, as opposed to state or action histories. The set union and intersection of (11.30) and (11.31) are replaced in this section by marginalization and Bayes' rule, respectively. In a sense, these are the probabilistic equivalents of union and intersection. It will be very helpful to compare the expressions from this section to those of Section 11.2.2.

Rather than write $\kappa_{prob}(\eta)$, standard probability notation will be applied to obtain $P(x|\eta)$. Most expressions in this section of the form $P(x_k|\cdot)$ have an analogous expression in Section 11.2.2 of the form $X_k(\cdot)$. It is helpful to recognize the similarities.

The first step is to construct probabilistic versions of H and F. These are $P(x_k|y_k)$ and $P(x_{k+1}|x_k,u_k)$, respectively. The latter term was given in Section 10.1.1. To obtain $P(x_k|y_k)$, recall from Section 11.1.1 that $P(y_k|x_k)$ is easily derived from $P(\psi_k|x_k)$. To obtain $P(x_k|y_k)$, Bayes' rule is applied:

$$P(x_k|y_k) = \frac{P(y_k|x_k)P(x_k)}{P(y_k)} = \frac{P(y_k|x_k)P(x_k)}{\displaystyle\sum_{x_k \in X} P(y_k|x_k)P(x_k)}. \tag{11.35}$$

In the last step, $P(y_k)$ was rewritten using marginalization, (9.8). In this case x_k appears as the sum index; therefore, the denominator is only a function of y_k, as required. Bayes' rule requires knowing the prior, $P(x_k)$. In the coming expressions, this will be replaced by a probabilistic I-state.

Now consider defining probabilistic I-states. Each is a probability distribution over X and is written as $P(x_k|\eta_k)$. The initial condition produces $P(x_1)$. As for the nondeterministic case, probabilistic I-states can be computed inductively. For the base case, the only new piece of information is y_1. Thus, the probabilistic I-state, $P(x_1|\eta_1)$, is $P(x_1|y_1)$. This is computed by letting $k = 1$ in (11.35) to yield

$$P(x_1|\eta_1) = P(x_1|y_1) = \frac{P(y_1|x_1)P(x_1)}{\displaystyle\sum_{x_1 \in X} P(y_1|x_1)P(x_1)}. \tag{11.36}$$

Now consider the inductive step by assuming that $P(x_k|\eta_k)$ is given. The task is to determine $P(x_{k+1}|\eta_{k+1})$, which is equivalent to $P(x_{k+1}|\eta_k, u_k, y_{k+1})$. As in

Section 11.2.2, this will proceed in two parts by first considering the effect of u_k, followed by y_{k+1}. The first step is to determine $P(x_{k+1}|\eta_k, u_k)$ from $P(x_k|\eta_k)$. First, note that

$$P(x_{k+1}|\eta_k, x_k, u_k) = P(x_{k+1}|x_k, u_k) \tag{11.37}$$

because η_k contains no additional information regarding the prediction of x_{k+1} once x_k is given. Marginalization, (9.8), can be used to eliminate x_k from $P(x_{k+1}|x_k, u_k)$. This must be eliminated because it is not given. Putting these steps together yields

$$\begin{aligned} P(x_{k+1}|\eta_k, u_k) &= \sum_{x_k \in X} P(x_{k+1}|x_k, u_k, \eta_k) P(x_k|\eta_k) \\ &= \sum_{x_k \in X} P(x_{k+1}|x_k, u_k) P(x_k|\eta_k), \end{aligned} \tag{11.38}$$

which expresses $P(x_{k+1}|\eta_k, u_k)$ in terms of given quantities. Equation (11.38) can be considered as the probabilistic counterpart of (11.30).

The next step is to take into account the observation y_{k+1}. This is accomplished by making a version of (11.35) that is conditioned on the information accumulated so far: η_k and u_k. Also, k is replaced with $k+1$. The result is

$$P(x_{k+1}|y_{k+1}, \eta_k, u_k) = \frac{P(y_{k+1}|x_{k+1}, \eta_k, u_k) P(x_{k+1}|\eta_k, u_k)}{\displaystyle\sum_{x_{k+1} \in X} P(y_{k+1}|x_{k+1}, \eta_k, u_k) P(x_{k+1}|\eta_k, u_k)}. \tag{11.39}$$

This can be considered as the probabilistic counterpart of (11.31). The left side of (11.39) is equivalent to $P(x_{k+1}|\eta_{k+1})$, which is the probabilistic I-state for stage $k+1$, as desired. There are two different kinds of terms on the right. The expression for $P(x_{k+1}|\eta_k, u_k)$ is given in (11.38). Therefore, the only remaining term to calculate is $P(y_{k+1}|x_{k+1}, \eta_k, u_k)$. Note that

$$P(y_{k+1}|x_{k+1}, \eta_k, u_k) = P(y_{k+1}|x_{k+1}) \tag{11.40}$$

because the sensor mapping depends only on the state (and the probability model for the nature sensing action, which also depends only on the state). Since $P(y_{k+1}|x_{k+1})$ is specified as part of the sensor model, we have now determined how to obtain $P(x_{k+1}|\eta_{k+1})$ from $P(x_k|\eta_k)$, u_k, and y_{k+1}. Thus, $\mathcal{I}_{prob}$ is another I-space that can be treated as just another state space.

The probabilistic I-space $\mathcal{I}_{prob}$ (shown in Figure 11.3) is the set of all probability distributions over X. The update expressions, (11.38) and (11.39), establish that the I-map κ_{prob} is sufficient, which means that the planning problem can be expressed entirely in terms of $\mathcal{I}_{prob}$, instead of maintaining histories. A goal region can be specified as constraints on the probabilities. For example, from some particular $x \in X$, the goal might be to reach any probabilistic I-state for which $P(x_k|\eta_k) > 1/2$.

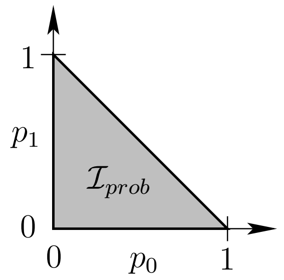


Figure 11.6: The probabilistic I-space for the three-state example is a 2-simplex embedded in $\mathbb{R}^3$. This simplex can be projected into $\mathbb{R}^2$ to yield the depicted triangular region in $\mathbb{R}^2$.

Example 11.14 (Three-State Example Revisited) Now return to Example 11.13, but this time use probabilistic models. For a probabilistic I-state, let p_i denote the probability that the current state is $i \in X$. Any probabilistic I-state can be expressed as $(p_0, p_1, p_2) \in \mathbb{R}^3$. This implies that the I-space can be nicely embedded in $\mathbb{R}^3$. By the axioms of probability (given in Section 9.1.2), $p_0 + p_1 + p_2 = 1$, which can be interpreted as a plane equation in $\mathbb{R}^3$ that restricts $\mathcal{I}_{prob}$ to a 2D set. Also following the axioms of probability, for each $i \in \{0, 1, 2\}$, $0 \leq p_i \leq 1$. This means that $\mathcal{I}_{prob}$ is restricted to a triangular region in $\mathbb{R}^3$. The vertices of this triangular region are $(0, 0, 1)$, $(0, 1, 0)$, and $(1, 0, 0)$; these correspond to the three different ways to have perfect state information. In a sense, the distance away from these points corresponds to the amount of uncertainty in the state. The uniform probability distribution $(1/3, 1/3, 1/3)$ is equidistant from the three vertices. A projection of the triangular region into $\mathbb{R}^2$ is shown in Figure 11.6. The interpretation in this case is that p_0 and p_1 specify a point in $\mathbb{R}^2$, and p_2 is automatically determined from $p_2 = 1 - p_0 - p_1$.

The triangular region in $\mathbb{R}^3$ is an uncountably infinite set, even though the history I-space is countably infinite for a fixed initial condition. This may seem strange, but there is no mistake because for a fixed initial condition, it is generally impossible to reach all of the points in $\mathcal{I}_{prob}$. If the initial condition can be any point in $\mathcal{I}_{prob}$, then all of the probabilistic I-space is covered because $\mathcal{I}_0 = \mathcal{I}_{prob}$, in which $\mathcal{I}_0$ is the initial condition space.. ■

11.2.4 Limited-Memory Information Spaces

Limiting the amount of memory provides one way to reduce the sizes of history I-states. Except in special cases, this usually does not preserve the feasibility or optimality of the original problem. Nevertheless, such I-maps are very useful in practice when there appears to be no other way to reduce the size of the I-space. Furthermore, they occasionally do preserve the desired properties of feasibility, and sometimes even optimality.

Previous *i* stages Under this model, the history I-state is truncated. Any actions or observations received earlier than i stages ago are dropped from memory. An I-map, κ_i, is defined as

$$\kappa_i(\eta_k) = (u_{k-i}, \ldots, u_{k-1}, y_{k-i+1}, \ldots, y_k), \tag{11.41}$$

for any integer $i > 0$ and $k > i$. If $i \leq k$, then the derived I-state is the full history I-state, (11.14). The advantage of this approach, if it leads to a solution, is that the length of the I-state no longer grows with the number of stages. If X and U are finite, then the derived I-space is also finite. Note that κ_i is sufficient in the sense defined in Section 11.2.1 because enough history is passed from stage to stage to determine the derived I-states.

Sensor feedback An interesting I-map is obtained by removing all but the last sensor observation from the history I-state. This yields an I-map, $\kappa_{sf} : \mathcal{I}_{hist} \rightarrow Y$, which is defined as $\kappa_{sf}(\eta_k) = y_k$. The model is referred to as *sensor feedback*. In this case, all decisions are made directly in terms of the current sensor observation. The derived I-space is Y, and a plan on the derived I-space is $\pi : Y \rightarrow U$, which is called a *sensor-feedback plan*. In some literature, this may be referred to as a purely *reactive plan*. Many problems for which solutions exist in the history I-space cannot be solved using sensor feedback. Neglecting history prevents the complicated deductions that are often needed regarding the state. In some sense, sensor feedback causes short-sightedness that could unavoidably lead to repeating the same mistakes indefinitely. However, it may be worth determining whether such a sensor-feedback solution plan exists for some particular problem. Such plans tend to be simpler to implement in practice because the actions can be connected directly to the sensor output. Certainly, if a sensor-feedback solution plan exists for a problem, and feasibility is the only concern, then it is pointless to design and implement a plan in terms of the history I-space or some larger derived I-space. Note that this I-map is sufficient, even though it ignores the entire history.

11.3 Examples for Discrete State Spaces

11.3.1 Basic Nondeterministic Examples

First, we consider a simple example that uses the sign sensor of Example 11.3.

Example 11.15 (Using the Sign Sensor) This example is similar to Example 10.1, except that it involves sensing uncertainty instead of prediction uncertainty. Let $X = \mathbb{Z}$, $U = \{-1, 1, u_T\}$, $Y = \{-1, 0, 1\}$, and $y = h(x) = \text{sgn} x$. For the state transition equation, $x_{k+1} = f(x_k, u_k) = x_k + u_k$. No nature actions interfere with

the state transition equation or the sensor mapping. Therefore, future history I-states are predictable. The information transition equation is $\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k)$. Suppose that initially, $\eta_0 = X$, which means that any initial state is possible. The goal is to terminate at $0 \in X$.

The general expression for a history I-state at stage k is

$$\eta_k = (X, u_1, \ldots, u_{k-1}, y_1, \ldots, y_k). \tag{11.42}$$

A possible I-state is $\eta_5 = (X, -1, 1, 1, -1, 1, 1, 1, 1, 0)$. Using the nondeterministic I-space from Section 11.2.2, $\mathcal{I}_{ndet} = \text{pow}(X)$, which is uncountably infinite. By looking carefully at the problem, however, it can be seen that most of the nondeterministic I-states are not reachable. If $y_k = 0$, it is known that $x_k = 0$; hence, $X_k(\eta_k) = \{0\}$. If $y_k = 1$, it will always be the case that $X_k(\eta_k) = \{1, 2, \ldots\}$ unless 0 is observed. If $y_k = -1$, then $X_k(\eta_k) = \{\ldots, -2, -1\}$. From this a plan, π, can be specified over the three nondeterministic I-states mentioned above. For the first one, $\pi(X_k(\eta_k)) = u_T$. For the other two, $\pi(X_k(\eta_k)) = -1$ and $\pi(X_k(\eta_k)) = 1$, respectively. Based on the sign, the plan tries to move toward 0. If different initial conditions are allowed, then more nondeterministic I-states can be reached, but this was not required as the problem was defined. Note that optimal-length solutions are produced by the plan.

The solution can even be implemented with sensor feedback because the action depends only on the current sensor value. Let $\pi : Y \rightarrow U$ be defined as

$$\pi(y) = \begin{cases} -1 & \text{if } y = 1 \\ 1 & \text{if } y = -1 \\ u_T & \text{if } y = 0. \end{cases} \tag{11.43}$$

This provides dramatic memory savings over defining a plan on $\mathcal{I}_{hist}$. ■

The next example provides a simple illustration of solving a problem without ever knowing the current state. This leads to the *goal recognizability* problem [177] (see Section 12.5.1).

Example 11.16 (Goal Recognizability) Let $X = \mathbb{Z}$, $U = \{-1, 1, u_T\}$, and $Y = \mathbb{Z}$. For the state transition equation, $x_{k+1} = f(x_k, u_k) = x_k + u_k$. Now suppose that a variant of Example 11.7 is used to model sensing: $y = h(x, \psi) = x + \psi$ and $\Psi = \{-5, -4, \ldots, 5\}$. Suppose that once again, $\eta_0 = X$. In this case, it is impossible to guarantee that a goal, $X_G = \{0\}$, is reached because of the goal recognizability problem. The disturbance in the sensor mapping does not allow precise enough state measurements to deduce the precise achievement of the state. If the goal region, X_G, is enlarged to $\{-5, -4, \ldots, 5\}$, then the problem can be solved. Due to the disturbance, the nondeterministic I-state is always a subset of a consecutive sequence of 11 states. It is simple to derive a plan that moves this interval until the nondeterministic I-state becomes a subset of X_G. When this

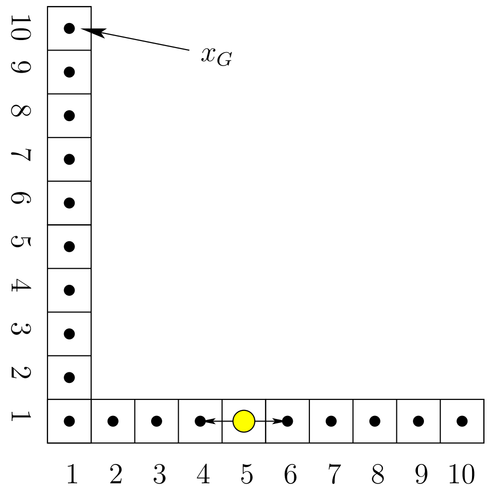


Figure 11.7: An example that involves 19 states. There are no sensor observations; however, actions can be chosen that enable the state to be estimated. The example provides an illustration of reducing the I-space via I-maps.

occurs, then the plan applies u_T. In solving this problem, the exact state never had to be known. ■

The problem shown in Figure 11.7 serves two purposes. First, it is an example of *sensorless planning* [97, 112], which means that there are no observations (see Sections 11.5.4 and 12.5.2). This is an interesting class of problems because it appears that no information can be gained regarding the state. Contrary to intuition, it turns out for this example and many others that plans can be designed that estimate the state. The second purpose is to illustrate how the I-space can be dramatically collapsed using the I-map concepts of Section 11.2.1. The standard nondeterministic I-space for this example contains 2^{19} I-states, but it can be mapped to a much smaller derived I-space that contains only a few elements.

Example 11.17 (Moving in an L-shaped Corridor) The state space X for the example shown in Figure 11.7 has 19 states, each of which corresponds to a location on one of the white tiles. For convenience, let each state be denoted by (i, j). There are 10 *bottom states*, denoted by $(1, 1)$, $(2, 1)$, $\ldots$, $(10, 1)$, and 10 *left states*, denoted by $(1, 1)$, $(1, 2)$, $\ldots$, $(1, 10)$. Since $(1, 1)$ is both a bottom state and a left state, it is called the *corner state*.

There are no sensor observations for this problem. However, nature interferes with the state transitions, which leads to a form of nondeterministic uncertainty. If an action is applied that tries to take one step, nature may cause two or three

steps to be taken. This can be modeled as follows. Let

$$U = \{(1, 0), (-1, 0), (0, 1), (0, -1)\} \tag{11.44}$$

and let $\Theta = \{1, 2, 3\}$. The state transition equation is defined as $f(x, u, \theta) = x+\theta u$ whenever such motion is not blocked (by hitting a dead end). For example, if $x = (5, 1)$, $u = (-1, 0)$, and $\theta = 2$, then the resulting next state is $(5, 1) + 2(-1, 0) = (3, 1)$. If blocking is possible, the state changes as much as possible until it becomes blocked. Due to blocking, it is even possible that $f(x, u, \theta) = x$.

Since there are no sensor observations, the history I-state at stage k is

$$\eta_k = (\eta_0, u_1, \ldots, u_{k-1}). \tag{11.45}$$

Now use the nondeterministic I-space, $\mathcal{I}_{ndet} = \text{pow}(X)$. The initial state, $x_1 = (10, 1)$, is given, which means that the initial I-state, η_0, is $\{(10, 1)\}$. The goal is to arrive at the I-state, $\{(1, 10)\}$, which means that the task is to design a plan that moves from the lower right to the upper left.

With perfect information, this would be trivial; however, without sensors the uncertainty may grow very quickly. For example, after applying the action $u_1 = (-1, 0)$ from the initial state, the nondeterministic I-state becomes $\{(7, 1), (8, 1), (9, 1)\}$. After $u_2 = (-1, 0)$ it becomes $\{(4, 1), \ldots, (8, 1)\}$. A nice feature of this problem, however, is that uncertainty can be reduced without sensing. Suppose that for 100 stages, we repeatedly apply $u_k = (-1, 0)$. What is the resulting I-state? As the corner state is approached, the uncertainty is reduced because the state cannot be further changed by nature. It is known that each action, $u_k = (-1, 0)$, decreases the X coordinate by at least one each time. Therefore, after nine or more stages, it is known that $\eta_k = \{(1, 1)\}$. Once this is known, then the action $(0, 1)$ can be applied. This will again increase uncertainty as the state moves through the set of left states. If $(0, 1)$ is applied nine or more times, then it is known for certain that $x_k = (1, 10)$, which is the required goal state.

A successful plan has now been obtained: 1) Apply $(-1, 0)$ for nine stages, 2) then apply $(0, 1)$ for nine stages. This plan could be defined over $\mathcal{I}_{ndet}$; however, it is simpler to use the I-map κ_{stage} from Example 11.12 to define a plan as $\pi : \mathbb{N} \rightarrow U$. For k such that $1 \leq k \leq 9$, $\pi(k) = (-1, 0)$. For k such that $10 \leq k \leq 18$, $\pi(k) = (0, 1)$. For $k > 18$, $\pi(k) = u_T$. Note that the plan works even if the initial condition is any subset of X. From this point onward, assume that any subset may be given as the initial condition.

Some alternative plans will now be considered by making other derived I-spaces from $\mathcal{I}_{ndet}$. Let κ_3 be an I-map from $\mathcal{I}_{ndet}$ to a set $\mathcal{I}_3$ of three derived I-states. Let $\mathcal{I}_3 = \{g, l, a\}$, in which g denotes "goal," l denotes "left," and a denotes "any." The I-map, κ_3, is

$$X(\eta) = \begin{cases} g & \text{if } X(\eta) = \{(1, 10)\} \\ l & \text{if } X(\eta) \text{ is a subset of the set of left states} \\ a & \text{otherwise.} \end{cases} \tag{11.46}$$

Based on the successful plan described so far, a solution on $\mathcal{I}_3$ is defined as $\pi(g) = u_T$, $\pi(l) = (0, 1)$, and $\pi(a) = (-1, 0)$. This plan is simpler to represent than the one on $\mathbb{N}$; however, there is one drawback. The I-map κ_3 is not sufficient. This implies that more of the nondeterministic I-state needs to be maintained during execution. Otherwise, there is no way to know when certain transitions occur. For example, if $(-1, 0)$ is applied from a, how can the robot determine whether l or a is reached in the next stage? This can be easily determined from the nondeterministic I-state.

To address this problem, consider a new I-map, $\kappa_{19} : \mathcal{I}_{ndet} \to \mathcal{I}_{19}$, which is sufficient. There are 19 derived I-states, which include g as defined previously, l_i for $1 \leq j \leq 9$, and a_i for $2 \leq i \leq 10$. The I-map is defined as $\kappa_{19}(X(\eta)) = g$ if $X(\eta) = \{(1, 10)\}$. Otherwise, $\kappa_{19}(X(\eta)) = l_i$ for the smallest value of i such that $X(\eta)$ is a subset of $\{(1, i), \ldots, (1, 10)\}$. If there is no such value for i, then $\kappa_{19}(X(\eta)) = a_i$, for the smallest value of i such that $X(\eta)$ is a subset of $\{(1, 1), \ldots, (1, 10), (2, 1), \ldots, (i, 1)\}$. Now the plan is defined as $\pi(g) = u_T$, $\pi(l_i) = (0, 1)$, and $\pi(a_i) = (-1, 0)$. Although the plan is larger, the robot does not need to represent the full nondeterministic I-state during execution. The correct transitions occur. For example, if $u_k = (-1, 0)$ is applied at a_5, then a_4 is obtained. If $u = (-1, 0)$ is applied at a_2, then l_1 is obtained. From there, $u = (0, 1)$ is applied to yield l_2. These actions can be repeated until eventually l_9 and g are reached. The resulting plan, however, is not an improvement over the original open-loop one. ■

11.3.2 Nondeterministic Finite Automata

An interesting connection lies between the ideas of this chapter and the theory of finite automata, which is part of the theory of computation (see [126, 260]). In Section 2.1, it was mentioned that determining whether there exists some string that is accepted by a DFA is equivalent to a discrete feasible planning problem. If unpredictability is introduced into the model, then a *nondeterministic finite automaton* (NFA) is obtained, as depicted in Figure 11.8. This represents one of the simplest examples of nondeterminism in theoretical computer science. Such nondeterministic models serve as a powerful tool for defining models of computation and their associated complexity classes. It turns out that these models give rise to interesting examples of information spaces.

An NFA is typically described using a directed graph as shown in Figure 11.8b, and is considered as a special kind of finite state machine. Each vertex of the graph represents a state, and edges represent possible transitions. An *input string* of finite length is read by the machine. Typically, the input string is a binary sequence of 0's and 1's. The initial state is designated by an inward arrow that has no source vertex, as shown pointing into state a in Figure 11.8b. The machine starts in this state and reads the first symbol of the input string. Based

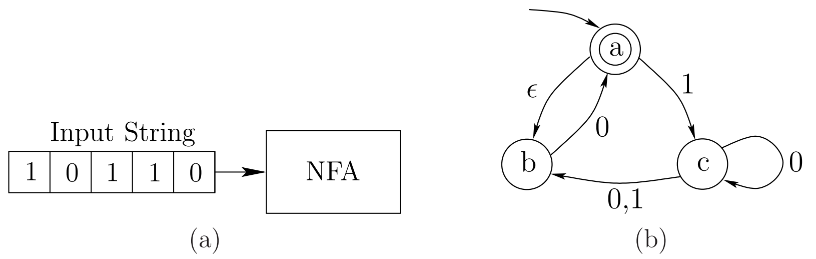


Figure 11.8: (a) An nondeterministic finite automaton (NFA) is a state machine that reads an input string and decides whether to accept it. (b) A graphical depiction of an NFA.

on its value, it makes appropriate transitions. For a DFA, the next state must be specified for each of the two inputs 0 and 1 from each state. From a state in an NFA, there may be any number of outgoing edges (including zero) that represent the response to a single symbol. For example, there are two outgoing edges if 0 is read from state c (the arrow from c to b actually corresponds to two directed edges, one for 0 and the other for 1). There are also edges designated with a special ϵ symbol. If a state has an outgoing ϵ, the state may immediately transition along the edge without reading another symbol. This may be iterated any number of times, for any outgoing ϵ edges that may be encountered, without reading the next input symbol. The nondeterminism arises from the fact that there are multiple choices for possible next states due to multiple edges for the same input and ϵ transitions. There is no sensor that indicates which state is actually chosen.

The interpretation often given in the theory of computation is that when there are multiple choices, the machine clones itself and one copy runs each choice. It is like having multiple universes in which each different possible action of nature is occurring simultaneously. If there are no outgoing edges for a certain combination of state and input, then the clone dies. Any states that are depicted with a double boundary, such as state a in Figure 11.8, are called *accept states*. When the input string ends, the NFA is said to *accept* the input string if there exists at least one alternate universe in which the final machine state is an accept state.

The formulation usually given for NFAs seems very close to Formulation 2.1 for discrete feasible planning. Here is a typical NFA formulation [260], which formalizes the ideas depicted in Figure 11.8:

Formulation 11.2 (Nondeterministic Finite Automaton)

1. A finite state space X.

2. A finite *alphabet* Σ which represents the possible input symbols. Let $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

3. A *transition function*, $\delta : X \times \Sigma_\epsilon \to \text{pow}(X)$. For each state and symbol, a set of outgoing edges is specified by indicating the states that are reached.

4. A *start state* $x_0 \in X$.

5. A set $A \subseteq X$ of *accept states*.

Example 11.18 (Three-State NFA) The example in Figure 11.8 can be expressed using Formulation 11.2. The components are $X = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $\Sigma_\epsilon = \{0, 1, \epsilon\}$, $x_0 = a$, and $A = \{a\}$. The state transition equation requires the specification of a state for every $x \in X$ and symbol in Σ_ϵ:

$$\begin{array}{c|ccc} & 0 & 1 & \epsilon \\ \hline a & \emptyset & \{c\} & \{b\} \\ b & \{a\} & \emptyset & \emptyset \\ c & \{b, c\} & \{b\} & \emptyset \end{array} . \tag{11.47}$$

■

Now consider reformulating the NFA and its acceptance of strings as a kind of planning problem. An input string can be considered as a plan that uses no form of feedback; it is a fixed sequence of actions. The feasible planning problem is to determine whether any string exists that is accepted by the NFA. Since there is no feedback, there is no sensing model. The initial state is known, but subsequent states cannot be measured. The history I-state η_k at stage k reduces to $\eta_k = \tilde{u}_{k-1} = (u_1, \ldots, u_{k-1})$, the action history. The nondeterminism can be accounted for by defining nature actions that interfere with the state transitions. This results in the following formulation, which is described in terms of Formulation 11.2.

Formulation 11.3 (An NFA Planning Problem)

1. A finite state space X.

2. An action space $U = \Sigma \cup \{u_T\}$.

3. A *state transition function*, $F : X \times U \to \text{pow}(X)$. For each state and symbol, a set of outgoing edges is specified by indicating the states that are reached.

4. An *initial state* $x_0 = x_1$.

5. A set $X_G = A$ of *goal states.*

The history I-space $\mathcal{I}_{hist}$ is defined using

$$\mathcal{I}_k = \tilde{U}_{k-1} \tag{11.48}$$

for each $k \in \mathbb{N}$ and taking the union as defined in (11.19). Assume that the initial state of the NFA is always fixed; therefore, it does not appear in the definition of $\mathcal{I}_{hist}$.

For expressing the planning task, it is best to use the nondeterministic I-space $\mathcal{I}_{ndet} = \text{pow}(X)$ from Section 11.2.2. Thus, each nondeterministic I-state, $X(\eta) \in \mathcal{I}_{ndet}$, is the subset of X that corresponds to the possible current states of the machine. The initial condition could be any subset of X because ϵ transitions can occur from x_1. Subsequent nondeterministic I-states follow directly from F. The task is to compute a plan of the form

$$\pi = (u_1, u_2, \ldots, u_K, u_T), \tag{11.49}$$

which results in $X_{K+1}(\eta_{K+1}) \in \mathcal{I}_{ndet}$ with $X_{K+1}(\eta_{K+1}) \cap X_G \neq \emptyset$. This means that at least one possible state of the NFA must lie in X_G after the termination action is applied. This condition is much weaker than a typical planning requirement. Using worst-case analysis, a typical requirement would instead be that *every* possible NFA state lies in X_G.

The problem given in Formulation 11.3 is not precisely a specialization of Formulation 11.1 because of the state transition function. For convenience, F was directly defined, instead of explicitly requiring that f be defined in terms of nature actions, $\Theta(x, u)$, which in this context depend on both x and u for an NFA. There is one other small issue regarding this formulation. In the planning problems considered in this book, it is always assumed that there is a current state. For an NFA, it was already mentioned that if there are no outgoing edges for a certain input, then the clone of the machine dies. This means that potential current states cease to exist. It is even possible that every clone dies, which leaves no current state for the machine. This can be easily enabled by directly defining F; however, planning problems must always have a current state. To resolve this issue, we could augment X in Formulation 11.3 to include an extra *dead* state, which signifies the death of a clone when there are no outgoing edges. A dead state can never lie in X_G, and once a transition to a dead state occurs, the state remains dead for all time. In this section, the state space will not be augmented in this way; however, it is important to note that the NFA formulation can easily be made consistent with Formulation 11.3.

The planning model can now be compared to the standard use of NFAs in the theory of computation. A *language* of an NFA is defined to be the set of all input strings that it accepts. The planning problem formulated here determines whether there exists a string (which is a plan that ends with termination actions) that is accepted by the NFA. Equivalently, a planning algorithm determines whether the language of an NFA is empty. Constructing the set of all successful plans is equivalent to determining the language of the NFA.

Example 11.19 (Planning for the Three-State NFA) The example in Figure 11.8 can be expressed using Formulation 11.2. The components are $X =$

$\{a, b, c\}$, $\Sigma = \{0, 1\}$, $\Sigma_\epsilon = \{0, 1, \epsilon\}$, $x_0 = a$, and $F = \{a\}$. The function $F(x, u)$ is defined as

$$\begin{array}{c|cc} & 0 & 1 \\ \hline a & \emptyset & \{c\} \\ b & \{a,b\} & \emptyset \\ c & \{b,c\} & \{b\}. \end{array} \qquad (11.50)$$

The nondeterministic I-space is

$$X(\eta) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}, \qquad (11.51)$$

in which the initial condition is $\eta_0 = \{a, b\}$ because an ϵ transition occurs immediately from a. An example plan that solves the problem is $(1, 0, 0, u_T, \ldots)$. This corresponds to sending an input string "100" through the NFA depicted in Figure 11.8. The sequence of nondeterministic I-states obtained during the execution of the plan is

$$\{a,b\} \xrightarrow{1} \{c\} \xrightarrow{0} \{b,c\} \xrightarrow{0} \{a,b,c\} \xrightarrow{u_T} \{a,b,c\}. \qquad (11.52)$$

■

A basic theorem from the theory of finite automata states that for the set of strings accepted by an NFA, there exists a DFA (deterministic) that accepts the same set [260]. This is proved by constructing a DFA directly from the nondeterministic I-space. Each nondeterministic I-state can be considered as a state of a DFA. Thus, the DFA has 2^n states, if the original NFA has n states. The state transitions of the DFA are derived directly from the transitions between nondeterministic I-states. When an input (or action) is given, then a transition occurs from one subset of X to another. A transition is made between the two corresponding states in the DFA. This construction is an interesting example of how the I-space is a new state space that arises when the states of the original state space are unknown. Even though the I-space is usually larger than the original state space, its states are always known. Therefore, the behavior appears the same as in the case of perfect state information. This idea is very general and may be applied to many problems beyond DFAs and NFAs; see Section 12.1.2

11.3.3 The Probabilistic Case: POMDPs

Example 11.14 generalizes nicely to the case of n states. In operations research and artificial intelligence literature, these are generally referred to as *partially observable Markov decision processes* or *POMDPs* (pronounced "pom dee peez"). For the case of three states, the probabilistic I-space, $\mathcal{I}_{prob}$, is a 2-simplex embedded in $\mathbb{R}^3$. In general, if $|X| = n$, then $\mathcal{I}_{prob}$ is an $(n-1)$-simplex embedded in $\mathbb{R}^n$. The coordinates of a point are expressed as $(p_0, p_1, \ldots, p_{n-1}) \in \mathbb{R}^n$. By the axioms of probability, $p_0 + \cdots + p_{n-1} = 1$, which implies that $\mathcal{I}_{prob}$ is an $(n-1)$-dimensional subspace of $\mathbb{R}^n$. The vertices of the simplex correspond to the

n cases in which the state is known; hence, their coordinates are $(0, 0, \ldots, 0, 1)$, $(0, 0, \ldots, 0, 1, 0)$, ..., $(1, 0, \ldots, 0)$. For convenience, the simplex can be projected into $\mathbb{R}^{n-1}$ by specifying a point in $\mathbb{R}^{n-1}$ for which $p_1 + \cdots + p_{n-2} \leq 1$ and then choosing the final coordinate as $p_{n-1} = 1 - p_1 + \cdots + p_{n-2}$. Section 12.1.3 presents algorithms for planning for POMDPs.

11.4 Continuous State Spaces

This section takes many of the concepts that have been developed in Sections 11.1 and 11.2 and generalizes them to continuous state spaces. This represents an important generalization because the configuration space concepts, on which motion planning was defined in Part II, are all based on continuous state spaces. In this section, the state space might be a configuration space, $X = \mathcal{C}$, as defined in Chapter 4 or any other continuous state space. Since it may be a configuration space, many interesting problems can be drawn from robotics.

During the presentation of the concepts of this section, it will be helpful to recall analogous concepts that were already developed for discrete state spaces. In many cases, the formulations appear identical. In others, the continuous case is more complicated, but it usually maintains some of the properties from the discrete case. It will be seen after introducing continuous sensing models in Section 11.5.1 that some problems formulated in continuous spaces are even more elegant and easy to understand than their discrete counterparts.

11.4.1 Discrete-Stage Information Spaces

Assume here that there are discrete stages. Let $X \subseteq \mathbb{R}^m$ be an n-dimensional manifold for $n \leq m$ called the *state space*.[4] Let $Y \subseteq \mathbb{R}^m$ be an n_y-dimensional manifold for $n_y \leq m$ called the *observation space*. For each $x \in X$, let $\Psi(x) \subseteq \mathbb{R}^m$ be an n_n-dimensional manifold for $n_n \leq m$ called the set of *nature sensing actions*. The three kinds of sensors mappings, h, defined in Section 11.1.1 are possible, to yield either a *state mapping*, $y = h(x)$, a *state-nature mapping* $y = h(x, \psi)$, or a *history-based*, $y = h_k(x_1, \ldots, x_k, y)$. For the case of a state mapping, the preimages, $H(y)$, once again induce a partition of X. Preimages can also be defined for state-action mappings, but they do not necessarily induce a partition of X.

Many interesting sensing models can be formulated in continuous state spaces. Section 11.5.1 provides a kind of sensor catalog. There is once again the choice of nondeterministic or probabilistic uncertainty if nature sensing actions are used. If nondeterministic uncertainty is used, the expressions are the same as the discrete case. Probabilistic models are defined in terms of a probability density function,

[4]If you did not read Chapter 4 and are not familiar with manifold concepts, then assume $X = \mathbb{R}^n$; it will not make much difference. Make similar assumptions for Y, $\Psi(x)$, U, and $\Theta(x, u)$.

$p : \Psi \rightarrow [0, \infty)$,[5] in which $p(\psi)$ denotes the continuous-time replacement for $P(\psi)$. The model can also be expressed as $p(y|x)$, in that same manner that $P(y|x)$ was obtained for discrete state spaces.

The usual three choices exist for the initial conditions: 1) Either $x_1 \in X$ is given; 2) a subset $X_1 \in X$ is given; or 3) a probability density function, $p(x)$, is given. The initial condition spaces in the last two cases can be enormous. For example, if $X = [0, 1]$ and any subset is possible as an initial condition, then $\mathcal{I}_0 = \text{pow}(\mathbb{R})$, which has higher cardinality than $\mathbb{R}$. If any probability density function is possible, then $\mathcal{I}_0$ is a space of functions.[6]

The I-space definitions from Section 11.1.2 remain the same, with the understanding that all of the variables are continuous. Thus, (11.17) and (11.19) serve as the definitions of $\mathcal{I}_k$ and $\mathcal{I}$. Let $U \subseteq \mathbb{R}^m$ be an n_u-dimensional manifold for $n_u \leq m$. For each $x \in X$ and $u \in U$, let $\Theta(x, u)$ be an n_θ-dimensional manifold for $n_\theta \leq m$. A discrete-stage I-space planning problem over continuous state spaces can be easily formulated by replacing each discrete variable in Formulation 11.1 by its continuous counterpart that uses the same notation. Therefore, the full formulation is not given.

11.4.2 Continuous-Time Information Spaces

Now assume that there is a continuum of stages. Most of the components of Section 11.4.1 remain the same. The spaces X, Y, $\Psi(x)$, U, and $\Theta(x, u)$ remain the same. The sensor mapping also remains the same. The main difference occurs in the state transition equation because the effect of nature must be expressed in terms of velocities. This was already introduced in Section 10.6. In that context, there was only uncertainty in predictability. In the current context there may be uncertainties in both predictability and in sensing the current state.

For the discrete-stage case, the history I-states were based on action and observation sequences. For the continuous-time case, the history instead becomes a function of time. As defined in Section 7.1.1, let T denote a *time interval*, which may be bounded or unbounded. Let $\tilde{y}_t : [0, t] \rightarrow Y$ be called the *observation history* up to time $t \in T$. Similarly, let $\tilde{u}_t : [0, t) \rightarrow U$ and $\tilde{x}_t : [0, t] \rightarrow X$ be called the *action history* and *state history*, respectively, up to time $t \in T$.

Thus, the three kinds of sensor mappings in the continuous-time case are as follows:

[5]Assume that all continuous spaces are measure spaces and all probability density functions are measurable functions over these spaces.

[6]To appreciate of the size of this space, it can generally be viewed as an infinite-dimensional vector space (recall Example 8.5). Consider, for example, representing each function with a series expansion. To represent any analytic function exactly over $[0, 1]$, an infinite sequence of real-valued coefficients may be needed. Each sequence can be considered as an infinitely long vector, and the set of all such sequences forms an infinite-dimensional vector space. See [103, 239] for more background on function spaces and functional analysis.

1. A *state-sensor mapping* is expressed as $y(t) = h(x(t))$, in which $x(t)$ and $y(t)$ are the state and observation, respectively, at time $t \in T$.

2. A *state-nature mapping* is expressed as $y(t) = h(x(t), \psi(t))$, which implies that nature chooses some $\psi(t) \in \Psi(x(t))$ for each $t \in T$.

3. A *history-based sensor mapping*, which could depend on all of the states obtained so far. Thus, it depends on the entire function $\tilde{x}_t$. This could be denoted as $y(t) = h(\tilde{x}_t, \psi(t))$ if nature can also interfere with the observation.

If $\tilde{u}_t$ and $\tilde{y}_t$ are combined with the initial condition η_0, the *history I-state at time t* is obtained as

$$\eta_t = (\eta_0, \tilde{u}_t, \tilde{y}_t). \tag{11.53}$$

The *history I-space at time t* is the set of all possible η_t and is denoted as $\mathcal{I}_t$. Note that $\mathcal{I}_t$ is a space of functions because each $\eta_t \in \mathcal{I}_t$ is a function of time. Recall that in the discrete-stage case, every $\mathcal{I}_k$ was combined into a single history I-space, $\mathcal{I}_{hist}$, using (11.18) or (11.19). The continuous-time analog is obtained as

$$\mathcal{I}_{hist} = \bigcup_{t \in T} \mathcal{I}_t, \tag{11.54}$$

which is an irregular collection of functions because they have different domains; this irregularity also occurred in the discrete-stage case, in which $\mathcal{I}_{hist}$ was composed of sequences of varying lengths.

A continuous-time version of the cost functional in Formulation 11.1 can be given to evaluate the execution of a plan. Let L denote a cost functional that may be applied to any state-action history $(\tilde{x}_t, \tilde{u}_t)$ to yield

$$L(\tilde{x}_t, \tilde{u}_t) = \int_0^t l(x(t'), u(t'))dt' + l_F(x(t)), \tag{11.55}$$

in which $l(x(t'), u(t'))$ is the instantaneous cost and $l_F(x(t))$ is a final cost.

11.4.3 Derived Information Spaces

For continuous state spaces, the motivation to construct derived I-spaces is even stronger than in the discrete case because the I-space quickly becomes unwieldy.

Nondeterministic and probabilistic I-spaces for discrete stages

The concepts of I-maps and derived I-spaces from Section 11.2 extend directly to continuous spaces. In the nondeterministic case, κ_{ndet} once again transforms the initial condition and history into a subset of X. In the probabilistic case, κ_{prob} yields a probability density function over X. First, consider the discrete-stage case.

The nondeterministic I-states are obtained exactly as defined in Section 11.2.2, except that the discrete sets are replaced by their continuous counterparts. For example, $F(x,u)$ as defined in (11.28) is now a continuous set, as are X and $\Theta(x,u)$. Since probabilistic I-states are probability density functions, the derivation in Section 11.2.3 needs to be modified slightly. There are, however, no important conceptual differences. Follow the derivation of Section 11.2.3 and consider which parts need to be replaced.

The replacement for (11.35) is

$$p(x_k|y_k) = \frac{p(y_k|x_k)p(x_k)}{\displaystyle\int_X p(y_k|x_k)p(x_k)dx_k}, \tag{11.56}$$

which is based in part on deriving $p(y_k|x_k)$ from $p(\psi_k|x_k)$. The base of the induction, which replaces (11.36), is obtained by letting $k = 1$ in (11.56). By following the explanation given from (11.37) to (11.40), but using instead probability density functions, the following update equations are obtained:

$$\begin{aligned} p(x_{k+1}|\eta_k, u_k) &= \int_X p(x_{k+1}|x_k, u_k, \eta_k)p(x_k|\eta_k)dx_k \\ &= \int_X p(x_{k+1}|x_k, u_k)p(x_k|\eta_k)dx_k, \end{aligned} \tag{11.57}$$

and

$$p(x_{k+1}|y_{k+1}, \eta_k, u_k) = \frac{p(y_{k+1}|x_{k+1})p(x_{k+1}|\eta_k, u_k)}{\displaystyle\int_X p(y_{k+1}|x_{k+1})p(x_{k+1}|\eta_k, u_k)dx_{k+1}}. \tag{11.58}$$

Approximating nondeterministic and probabilistic I-spaces

Many other derived I-spaces extend directly to continuous spaces, such as the limited-memory models of Section 11.2.4 and Examples 11.11 and 11.12. In the present context, it is extremely useful to try to collapse the I-space as much as possible because it tends to be unmanageable in most practical applications. Recall that an I-map, $\kappa : \mathcal{I}_{hist} \rightarrow \mathcal{I}_{der}$, partitions $\mathcal{I}_{hist}$ into sets over which a constant action must be applied. The main concern is that restricting plans to $\mathcal{I}_{der}$ does not inhibit solutions.

Consider making derived I-spaces that approximate nondeterministic or probabilistic I-states. Approximations make sense because X is usually a metric space in the continuous setting. The aim is to dramatically simplify the I-space while trying to avoid the loss of critical information. A trade-off occurs in which the quality of the approximation is traded against the size of the resulting derived I-space. For the case of nondeterministic I-states, *conservative approximations* are formulated, which are sets that are guaranteed to contain the nondeterministic I-state. For the probabilistic case, *moment-based approximations* are presented,

which are based on general techniques from probability and statistics to approximate probability densities. To avoid unnecessary complications, the presentation will be confined to the discrete-stage model.

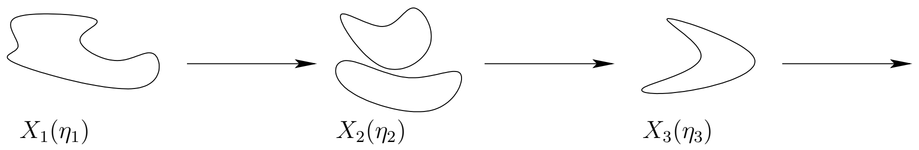


Figure 11.9: The nondeterministic I-states may be complicated regions that are difficult or impossible to compute.

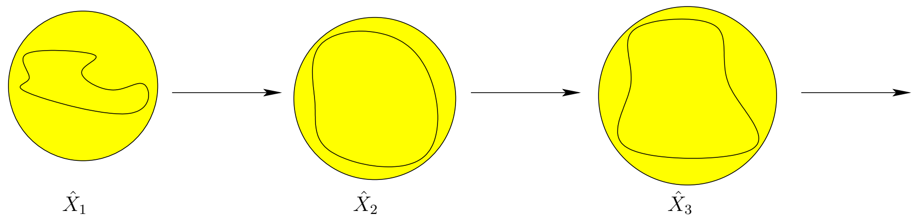


Figure 11.10: The nondeterministic I-states can be approximated by bounding spheres.

Conservative approximations Suppose that nondeterministic uncertainty is used and an approximation is made to the nondeterministic I-states. An I-map, $\kappa_{app} : \mathcal{I}_{ndet} \rightarrow \mathcal{I}_{app}$, will be defined in which $\mathcal{I}_{app}$ is a particular family of subsets of X. For example, $\mathcal{I}_{app}$ could represent the set of all ball subsets of X. If $X = \mathbb{R}^2$, then the balls become discs, and only three parameters (x, y, r) are needed to parameterize $\mathcal{I}_{app}$ (x, y for the center and r for the radius). This implies that $\mathcal{I}_{app} \subset \mathbb{R}^3$; this appears to be much simpler than $\mathcal{I}_{ndet}$, which could be a complicated collection of regions in $\mathbb{R}^2$. To make $\mathcal{I}_{app}$ even smaller, it could be required that x, y, and r are integers (or are sampled with some specified dispersion, as defined in Section 5.2.3). If $\mathcal{I}_{app}$ is bounded, then the number of derived I-states would become finite. Of course, this comes an at expense because $\mathcal{I}_{ndet}$ may be poorly approximated.

For a fixed sequence of actions $(u_1, u_2, \ldots)$ consider the sequence of nondeterministic I-states:

$$X_1(\eta_1) \stackrel{u_1,y_2}{\longrightarrow} X_2(\eta_2) \stackrel{u_2,y_3}{\longrightarrow} X_3(\eta_3) \stackrel{u_3,y_4}{\longrightarrow} \cdots, \tag{11.59}$$

which is also depicted in Figure 11.9. The I-map $\mathcal{I}_{app}$ must select a bounding region for every nondeterministic I-state. Starting with a history I-state, η, the nondeterministic I-state $X_k(\eta_k)$ can first be computed, followed by applying $\mathcal{I}_{app}$ to yield a bounding region. If there is a way to efficiently compute $X_k(\eta_k)$ for any η_k, then a plan on $\mathcal{I}_{app}$ could be much simpler than those on $\mathcal{I}_{ndet}$ or $\mathcal{I}_{hist}$.

If it is difficult to compute $X_k(\eta_k)$, one possibility is to try to define a derived information transition equation, as discussed in Section 11.2.1. The trouble, however, is that $\mathcal{I}_{app}$ is usually not a sufficient I-map. Imagine wanting to compute $\kappa_{app}(X_{k+1}(\eta_{k+1}))$, which is a bounding approximation to $X_{k+1}(\eta_{k+1})$. This can be accomplished by starting with $X_k(\eta_k)$, applying the update rules (11.30) and (11.31), and then applying κ_{app} to $X_{k+1}(\eta_{k+1})$. In general, this does not produce the same result as starting with the bounding volume $\mathcal{I}_{app}(X_k(\eta_k))$, applying (11.30) and (11.31), and then applying κ_{app}.

Thus, it is not possible to express the transitions entirely in $\mathcal{I}_{app}$ without some further loss of information. However, if this loss of information is tolerable, then an information-destroying approximation may nevertheless be useful. The general idea is to make a bounding region for the nondeterministic I-state in each iteration. Let $\hat{X}_k$ denote this bounding region at stage k. Be careful in using such approximations. As depicted in Figures 11.9 and 11.10, the sequences of derived I-states diverge. The sequence in Figure 11.10 is *not* obtained by simply bounding each calculated I-state by an element of $\mathcal{I}_{app}$; the entire sequence is different.

Initially, $\hat{X}_1$ is chosen so that $X_1(\eta_1) \subseteq \hat{X}_1$. In each inductive step, $\hat{X}_k$ is treated as if it were the true nondeterministic I-state (not an approximation). Using (11.30) and (11.31), the update for considering u_k and y_{k+1} is

$$\hat{X}'_{k+1} = \left(\bigcup_{x_k \in \hat{X}_k} F(x_k, u_k) \right) \cap H(y_{k+1}). \tag{11.60}$$

In general, $\hat{X}'_{k+1}(\eta_{k+1})$ might not lie in $\mathcal{I}_{app}$. Therefore, a bounding region, $\hat{X}_{k+1} \in \mathcal{I}_{app}$, must be selected to approximate $\hat{X}'$ under the constraint that $\hat{X}'_{k+1} \subseteq \hat{X}_{k+1}$. This completes the inductive step, which together with the base case yields a sequence

$$\hat{X}_1 \xrightarrow{u_1,y_2} \hat{X}_2 \xrightarrow{u_2,y_3} \hat{X}_3 \xrightarrow{u_3,y_4} \cdots, \tag{11.61}$$

which is depicted in Figure 11.10.

Both a plan, $\pi : \mathcal{I}_{app} \to U$, and information transitions can now be defined over $\mathcal{I}_{app}$. To ensure that a plan is sound, the approximation must be conservative. If in some iteration, $\hat{X}_{k+1}(\eta_{k+1}) \subset \hat{X}'_{k+1}(\eta_{k+1})$, then the true state may not necessarily be included in the approximate derived I-state. This could, for example, mean that a robot is in a collision state, even though the derived I-state indicates that this is impossible. This bad behavior is generally avoided by keeping conservative approximations. At one extreme, the approximations can be made very conservative by always assigning $\hat{X}_{k+1}(\eta_{k+1}) = X$. This, however, is useless because the only possible plans must apply a single, fixed action for every stage. Even if the approximations are better, it might still be impossible to cause transitions in the approximated I-state. To ensure that solutions exist to the planning problem, it is therefore important to make the bounding volumes as close as possible to the derived I-states.

This trade-off between the simplicity of bounding volumes and the computational expense of working with them was also observed in Section 5.3.2 for collision detection. Dramatic improvement in performance can be obtained by working with simpler shapes; however, in the present context this could come at the expense of failing to solve the problem. Using balls as described so far might not seem to provide very tight bounds. Imagine instead using solid ellipsoids. This would provide tighter approximations, but the dimension of $\mathcal{I}_{app}$ grows quadratically with the dimension of X. A sphere equation generally requires $n+1$ parameters, whereas the ellipsoid equation requires $\binom{n}{2} + 2n$ parameters. Thus, if the dimension of X is high, it may be difficult or even impossible to use ellipsoid approximations. Nonconvex bounding shapes could provide even better approximations, but the required number of parameters could easily become unmanageable, even if $X = \mathbb{R}^2$. For very particular problems, however, it may be possible to design a family of shapes that is both manageable and tightly approximates the nondeterministic I-states. This leads to many interesting research issues.

Moment-based approximations Since the probabilistic I-states are functions, it seems natural to use function approximation methods to approximate $\mathcal{I}_{prob}$. One possibility might be to use the first m coefficients of a Taylor series expansion. The derived I-space then becomes the space of possible Taylor coefficients. The quality of the approximation is improved as m is increased, but also the dimension of the derived I-space rises.

Since we are working with probability density functions, it is generally preferable to use moments as approximations instead of Taylor series coefficients or other generic function approximation methods. The first and second moments are the familiar *mean* and *covariance*, respectively. These are preferable over other approximations because the mean and covariance exactly represent the Gaussian density, which is the most basic and fundamental density in probability theory. Thus, approximating the probabilistic I-space with first and second moments is equivalent to assuming that the resulting probability densities are always Gaussian. Such approximations are frequently made in practice because of the convenience of working with Gaussians. In general, higher order moments can be used to obtain higher quality approximations at the expense of more coefficients. Let $\kappa_{mom} : \mathcal{I}_{prob} \to \mathcal{I}_{mom}$ denote a moment-based I-map.

The same issues arise for κ_{mom} as for κ_{app}. In most cases, κ_{mom} is not a sufficient I-map. The moments are computed in the same way as the conservative approximations. The update equations (11.57) and (11.58) are applied for probabilistic I-states; however, after each step, κ_{mom} is applied to the resulting probability density function. This traps the derived I-states in $\mathcal{I}_{mom}$. The moments could be computed after each of (11.57) and (11.58) or after both of them have been applied (different results may be obtained). The later case may be more difficult to compute, depending on the application.

First consider using the mean (first moment) to represent some probabilistic I-state, $p(x|\eta)$. Let x_i denote the ith coordinate of x. The mean, $\bar{x}_i$, with respect to x_i is generally defined as

$$\bar{x}_i = \int_X x_i \, p(x|\eta) dx. \tag{11.62}$$

This leads to the vector mean $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$. Suppose that we would like to construct $\mathcal{I}_{mom}$ using only the mean. Since there is no information about the covariance of the density, working with $\bar{x}$ is very similar to estimating the state. The mean value serves as the estimate, and $\mathcal{I}_{mom} = X$. This certainly helps to simplify the I-space, but there is no way to infer the amount of uncertainty associated with the I-state. Was the probability mass concentrated greatly around $\bar{x}$, or was the density function very diffuse over X?

Using second moments helps to alleviate this problem. The covariance with respect to two variables, x_i and x_i, is

$$\sigma_{i,j} = \int_X x_i x_j \, p(x|\eta) dx. \tag{11.63}$$

Since $\sigma_{ij} = \sigma_{ji}$, the second moments can be organized into a symmetric *covariance matrix*,

$$\Sigma = \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \cdots & \sigma_{1,n} \\ \sigma_{2,1} & \sigma_{2,2} & \cdots & \sigma_{2,n} \\ \vdots & \vdots & & \vdots \\ \sigma_{n,1} & \sigma_{n,2} & \cdots & \sigma_{n,n} \end{pmatrix} \tag{11.64}$$

for which there are $\binom{n}{2} + n$ unique elements, corresponding to every $x_{i,i}$ and every way to pair x_i with x_j for each distinct i and j such that $1 \leq i, j \leq n$. This implies that if first and second moments are used, then the dimension of $\mathcal{I}_{mom}$ is $\binom{n}{2} + 2n$. For some problems, it may turn out that all probabilistic I-states are indeed Gaussians. In this case, the mean and covariance exactly capture the probabilistic I-space. The I-map in this case is sufficient. This leads to a powerful tool called the Kalman filter, which is the subject of Section 11.6.1.

Higher quality approximations can be made by taking higher order moments. The *rth moment* is defined as

$$\int_X x_{i_1} x_{i_2} \cdots x_{i_r} \, p(x|\eta) dx, \tag{11.65}$$

in which i_1, i_2, ..., i_r are r integers chosen with replacement from $\{1, \ldots, n\}$.

The moment-based approximation is very similar to the conservative approximations for nondeterministic uncertainty. The use of mean and covariance appears very similar to using ellipsoids for the nondeterministic case. The level sets of a Gaussian density are ellipsoids. These level sets generalize the notion of confidence intervals to confidence ellipsoids, which provides a close connection between the nondeterministic and probabilistic cases. The domain of a Gaussian density is $\mathbb{R}^n$, which is not bounded, contrary to the nondeterministic case. However, for a given confidence level, it can be approximated as a bounded set. For example, an elliptical region can be computed in which 99.9% of the probability mass falls. In general, it may be possible to combine the idea of moments and bounding volumes to construct a derived I-space for the probabilistic case. This could yield the guaranteed correctness of plans while also taking probabilities into account. Unfortunately, this would once again increase the dimension of the derived I-space.

Derived I-spaces for continuous time

The continuous-time case is substantially more difficult, both to express and to compute in general forms. In many special cases, however, there are elegant ways to compute it. Some of these will be covered in Section 11.5 and Chapter 12. To help complete the I-space framework, some general expressions are given here. In general, I-maps and derived I-spaces can be constructed following the ideas of Section 11.2.1.

Since there are no discrete transition rules, the derived I-states cannot be expressed in terms of simple update rules. However, they can at least be expressed as a function that indicates the state $x(t)$ that will be obtained after $\tilde{u}_t$ and $\tilde{\theta}_t$ are applied from an initial state $x(0)$. Often, this is obtained via some form of integration (see Section 14.1), although this may not be explicitly given. In general, let $X_t(\eta_t) \subset X$ denote a nondeterministic I-state at time t; this is the replacement for X_k from the discrete-stage case. The initial condition is denoted as X_0, as opposed to X_1, which was used in the discrete-stage case.

More definitions are needed to precisely characterize $X_t(\eta_t)$. Let $\tilde{\theta}_t : [0, t) \rightarrow \Theta$ denote the history of nature actions up to time t. Similarly, let $\tilde{\psi}_t : [0, t] \rightarrow \Psi$ denote the history of nature sensing actions. Suppose that the initial condition is $X_0 \subset X$. The nondeterministic I-state is defined as

$$\begin{aligned} X_t(\eta_t) = \{x \in X \mid \exists x' \in X_0, \;\; \exists \tilde{\theta}_t, \text{ and } \exists \tilde{\psi}_t \text{ such that} \\ x = \Phi(x', \tilde{u}_t, \tilde{\theta}_t) \text{ and } \forall t' \in [0, t], \; y(t') = h(x(t'), \psi(t'))\}. \end{aligned} \tag{11.66}$$

In words, this means that a state $x(t)$ lies in $X_t(\eta_t)$ if and only if there exists an initial state $x' \in X_0$, a nature history $\tilde{\theta}_t$, and a nature sensing action history, $\tilde{\psi}_t$ such that the transition equation causes arrival at $x(t)$ and the observation history $\tilde{y}_t$ agrees with the sensor mapping over all time from 0 to t.

It is also possible to derive a probabilistic I-state, but this requires technical details from continuous-time stochastic processes and stochastic differential equations. In some cases, the resulting expressions work out very nicely; however, it is difficult to formulate a general expression for the derived I-state because it

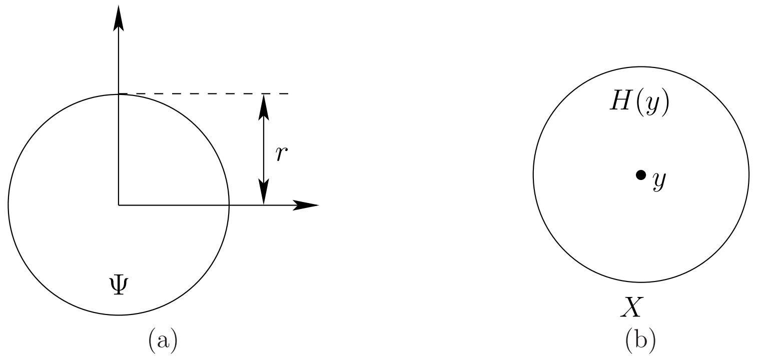


Figure 11.11: A simple sensing model in which the observation error is no more than r: (a) the nature sensing action space; (b) the preimage in X based on observation y.

depends on many technical assumptions regarding the behavior of the stochastic processes. For details on such systems, see [153].

11.5 Examples for Continuous State Spaces

11.5.1 Sensor Models

A remarkable variety of sensing models arises in applications that involve continuous state spaces. This section presents a catalog of different kinds of sensor models that is inspired mainly by robotics problems. The models are gathered together in one place to provide convenient reference. Some of them will be used in upcoming sections, and others are included to help in the understanding of I-spaces. For each sensor, there are usually two different versions, based on whether nature sensing actions are included.

Linear sensing models Developed mainly in control theory literature, *linear sensing models* are some of the most common and important. For all of the sensors in this family, assume that $X = Y = \mathbb{R}^n$ (nonsingular linear transformations allow the sensor space to effectively have lower dimension, if desired). The simplest case in this family is the *identity sensor*, in which $y = x$. In this case, the state is immediately known. If this sensor is available at every stage, then the I-space collapses to X by the I-map $\kappa_{sf} : \mathcal{I}_{hist} \rightarrow X$.

Now nature sensing actions can be used to corrupt this perfect state observation to obtain $y = h(x, \psi) = x + \psi$. Suppose that y is an estimate of x, the current state, with error bounded by a constant $r \in (0, \infty)$. This can be modeled by assigning for every $x \in X$, $\Psi(x)$ as a closed ball of radius r, centered at the

origin:

$$\Psi = \{\psi \in \mathbb{R}^n \mid \|\psi\| \leq r\}. \tag{11.67}$$

Figure 11.11 illustrates the resulting nondeterministic sensing model. If the observation y is received, then it is known that the true state lies within a ball in X of radius r, centered at y. This ball is the preimage, $H(y)$, as defined in (11.11). To make the model probabilistic, a probability density function can be defined over Ψ. For example, it could be assumed that $p(\psi)$ is a uniform density (although this model is not very realistic in many applications because there is a boundary at which the probability mass discontinuously jumps to zero).

A more typical probabilistic sensing model can be made by letting $\Psi(x) = \mathbb{R}^n$ and defining a probability density function over all of $\mathbb{R}^n$. (Note that the nondeterministic version of this sensor is completely useless.) One of the easiest choices to work with is the multivariate Gaussian probability density function,

$$p(\psi) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} e^{-\frac{1}{2}\psi^T \Sigma \psi}, \tag{11.68}$$

in which Σ is the covariance matrix (11.64), $|\Sigma|$ is its determinant, and $\psi^T \Sigma \psi$ is a quadratic form, which multiplies out to yield

$$\psi^T \Sigma \psi = \sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{i,j} \psi_i \psi_j. \tag{11.69}$$

If $p(x)$ is a Gaussian and y is received, then $p(y|x)$ must also be Gaussian under this model. This will become very important in Section 11.6.1.

The sensing models presented so far can be generalized by applying linear transformations. For example, let C denote a nonsingular $n \times n$ matrix with real-valued entries. If the sensor mapping is $y = h(x) = Cx$, then the state can still be determined immediately because the mapping $y = Cx$ is bijective; each $H(y)$ contains a unique point of X. A linear transformation can also be formed on the nature sensing action. Let W denote an $n \times n$ matrix. The sensor mapping is

$$y = h(x) = Cx + W\psi. \tag{11.70}$$

In general, C and W may even be singular, and a linear sensing model is still obtained. Suppose that $W = 0$. If C is singular, however, it is impossible to infer the state directly from a single sensor observation. This generally corresponds to a projection from an n-dimensional state space to a subset of Y whose dimension is the rank of C. For example, if

$$C = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \tag{11.71}$$

then $y = Cx$ yields $y_1 = x_2$ and $y_2 = 0$. Only x_2 of each $(x_1, x_2) \in X$ can be observed because C has rank 1. Thus, for some special cases, singular matrices can measure some state variables while leaving others invisible. For a general singular matrix C, the interpretation is that X is projected into some k-dimensional subspace by the sensor, in which k is the rank of C. If W is singular, this means that the effect of nature is limited. The degrees of freedom with which nature can distort the sensor observations is the rank of W. These concepts motivate the next set of sensor models.

Simple projection sensors Several common sensor models can be defined by observing particular coordinates of X while leaving others invisible. This is the continuous version of the selective sensor from Example 11.4. Imagine, for example, a mobile robot that rolls in a 2D world, $\mathcal{W} = \mathbb{R}^2$, and is capable of rotation. The state space (or configuration space) is $X = \mathbb{R}^2 \times \mathbb{S}^1$. For visualization purposes, it may be helpful to imagine that the robot is very tiny, so that it can be interpreted as a point, to avoid the complicated configuration space constructions of Section 4.3.[7] Let $p = (p_1, p_2)$ denote the coordinates of the point, and let $s \in \mathbb{S}^1$ denote its orientation. Thus, a state in $\mathbb{R}^2 \times \mathbb{S}^1$ is specified as (p_1, p_2, s) (rather than (x, y, θ), which may cause confusion with important spaces such as X, Y, and Θ).

Suppose that the robot can estimate its position but does not know its orientation. This leads to a *position sensor* defined as $Y = \mathbb{R}^2$, with $y_1 = p_1$ and $y_2 = p_2$ (also denoted as $y = h(x) = p$). The third state variable, s, of the state remains unknown. Of course, any of the previously considered nature sensing action models can be added. For example, nature might cause errors that are modeled with Gaussian probability densities.

A *compass* or *orientation sensor* can likewise be made by observing only the final state variable, s. In this case, $Y = \mathbb{S}^1$ and $y = s$. Nature sensing actions can be included. For example, the sensed orientation may be y, but it is only known that $|s - y| \leq \epsilon$ for some constant ϵ, which is the maximum sensor error. A Gaussian model cannot exactly be applied because its domain is unbounded and $\mathbb{S}^1$ is bounded. This can be fixed by truncating the Gaussian or by using a more appropriate distribution.

The position and orientation sensors generalize nicely to a 3D world, $\mathcal{W} = \mathbb{R}^3$. Recall from Section 4.2 that in this case the state space is $X = SE(3)$, which can be represented as $\mathbb{R}^3 \times \mathbb{RP}^3$. A position sensor measures the first three coordinates, whereas an orientation sensor measures the last three coordinates. A physical sensor that measures orientation in $\mathbb{R}^3$ is often called a *gyroscope*. These are usually based on the principle of precession, which means that they contain a spinning disc that is reluctant to change its orientation due to angular momentum. For the case of a linkage of bodies that are connected by revolute joints, a point in the state space gives the angles between each pair of attached links. A *joint encoder* is a sensor that yields one of these angles.

[7]This can also be handled, but it just adds unnecessary complication to the current discussion.

Dynamics of mechanical systems will not be considered until Part IV; however, it is worth pointing out several sensors. In these problems, the state space will be expanded to include velocity parameters and possibly even acceleration parameters. In this case, a *speedometer* can sense a velocity vector or a scalar speed. Sensors even exist to measure *angular velocity*, which indicates the speed with which rotation occurs. Finally, an *accelerometer* can be used to sense acceleration parameters. With any of these models, nature sensing actions can be used to account for measurement errors.

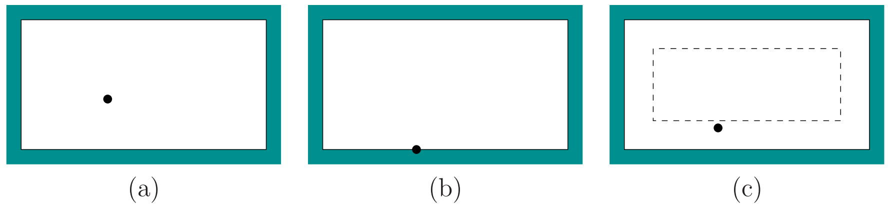


Figure 11.12: Boundary sensors indicate whether contact with the boundary has occurred. In the latter case, a proximity sensor may indicate whether the boundary is within a specified distance.

Boundary sensors If the state space has an interesting boundary, as in the case of $\mathcal{C}_{free}$ for motion planning problems, then many important *boundary sensors* can be formulated based on the detection of the boundary. Figure 11.12 shows several interesting cases on which sensors are based.

Suppose that the state space is a closed set with some well-defined boundary. To provide a connection to motion planning, assume that $X = \mathrm{cl}(\mathcal{C}_{free})$, the closure of $\mathcal{C}_{free}$. A *contact sensor* determines whether the boundary is being contacted. In this case, $Y = \{0, 1\}$ and h is defined as $h(x) = 1$ if $x \in \partial X$, and $h(x) = 0$ otherwise. These two cases are shown in Figures 11.12a and 11.12b, respectively. Using this sensor, there is no information regarding where along the boundary the contact may be occurring. In mobile robotics, it may be disastrous if the robot is in contact with obstacles. Instead, a *proximity sensor* is often used, which yields $h(x) = 1$ if the state or position is within some specified constant, r, of ∂X, and $h(x) = 0$ otherwise. This is shown in Figure 11.12.

In robot manipulation, haptic interfaces, and other applications in which physical interaction occurs between machines and the environment, a *force sensor* may be used. In addition to simply indicating contact, a force sensor can indicate the magnitude and direction of the force. The robot model must be formulated so that it is possible to derive the anticipated force value from a given state.

Landmark sensors Many important sensing models can be defined in terms of *landmarks*. A landmark is a special point or region in the state space that can be detected in some way by the sensor. The measurements of the landmark can be used to make inferences about the current state. An ancient example is using stars to navigate on the ocean. Based on the location of the stars relative to a ship, its orientation can be inferred. You may have found landmarks useful for trying to find your way through an unfamiliar city. For example, mountains around the perimeter of Mexico City or the Eiffel Tower in Paris might be used to infer your heading. Even though the streets of Paris are very complicated, it might be possible to walk to the Eiffel Tower by walking toward it whenever it is visible. Such models are common in the *competitive ratio* framework for analyzing on-line algorithms [185].

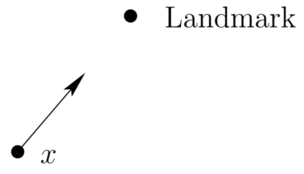


Figure 11.13: The most basic landmark sensor indicates only its direction.

In general, a set of states may serve as landmarks. A common model is to make x_G a single landmark. In robotics applications, these landmarks may be instead considered as points in the world, $\mathcal{W}$. Generalizations from points to landmark regions are also possible. The ideas, here, however, will be kept simple to illustrate the concept. Following this presentation, you can imagine a wide variety of generalizations. Assume for all examples of landmarks that $X = \mathbb{R}^2$, and let a state be denoted by $x = (x_1, x_2)$.

For the first examples, suppose there is only one landmark, $l \in X$, with coordinates (l_1, l_2). A *homing sensor* is depicted in Figure 11.13 and yields values in $Y = \mathbb{S}^1$. The sensor mapping is $h(x) = \text{atan2}(l_1 - x_1, l_2 - x_2)$, in which atan2 gives the angle in the proper quadrant.

Another possibility is a *Geiger counter sensor* (radiation level), in which $Y = [0, \infty)$ and $h(x) = \|x - l\|$. In this case, only the distance to the landmark is reported, but there is no directional information.

A contact sensor could also be combined with the landmark idea to yield a sensor called a *pebble*. This sensor reports 1 if the pebble is "touched"; otherwise, it reports 0. This idea can be generalized nicely to regions. Imagine that there is a *landmark region*, $X_l \subset X$. If $x \in X_l$, then the *landmark region detector* reports 1; otherwise, it reports 0.

Many useful and interesting sensing models can be formulated by using the ideas explained so far with multiple landmarks. For example, using three homing sensors that are not collinear, it is possible to reconstruct the exact state. Many interesting problems can be made by populating the state space with landmark regions and their associated detectors. In mobile robotics applications, this can be implemented by placing stationary cameras or other sensors in an environment.

The sensors can indicate which cameras can currently view the robot. They might also provide the distance from each camera.

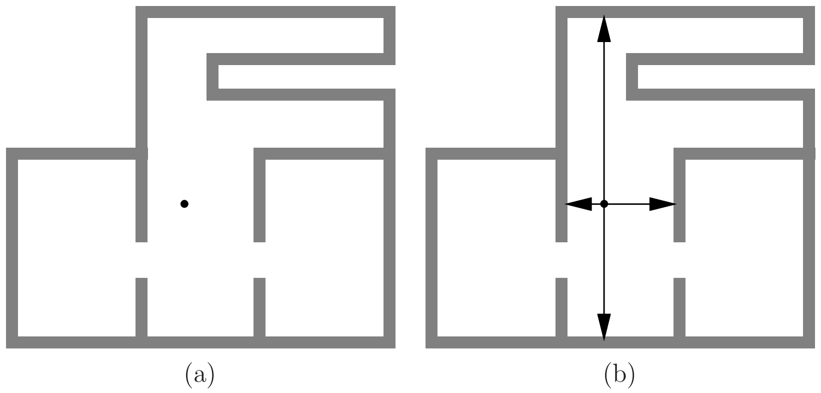


Figure 11.14: (a) A mobile robot is dropped into an unknown environment. (b) Four sonars are used to measure distances to the walls.

Depth-mapping sensors In many robotics applications, the robot may not have a map of the obstacles in the world. In this case, sensing is used to both learn the environment and to determine its position and orientation within the environment. Suppose that a robot is dropped into an environment as shown in Figure 11.14a. For problems such as this, the state represents both the position of the robot and the obstacles themselves. This situation is explained in further detail in Section 12.3. Here, some sensor models for problems of this type are given. These are related to the boundary and proximity sensors of Figure 11.12, but they yield more information when the robot is not at the boundary.

One of the oldest sensors used in mobile robotics is an acoustic *sonar*, which emits a high-frequency sound wave in a specific direction and measures the time that it takes for the wave to reflect off a wall and return to the sonar (often the sonar serves as both a speaker and a microphone). Based on the speed of sound and the time of flight, the distance to the wall can be estimated. Sometimes, the wave never returns; this can be modeled with nature. Also, errors in the distance estimate can be modeled with nature. In general, the observation space Y for a single sonar is $[0, \infty]$, in which ∞ indicates that the wave did not return. The interpretation of Y could be the time of flight, or it could already be transformed into estimated distance. If there are k sonars, each pointing in a different direction, then $Y = [0, \infty]^k$, which indicates that one reading can be obtained for each sonar. For example, Figure 11.14b shows four sonars and the distances that they can measure. Each observation therefore yields a point in $\mathbb{R}^4$.

Modern laser scanning technology enables very accurate distance measurements with very high angular density. For example, the SICK LMS-200 can

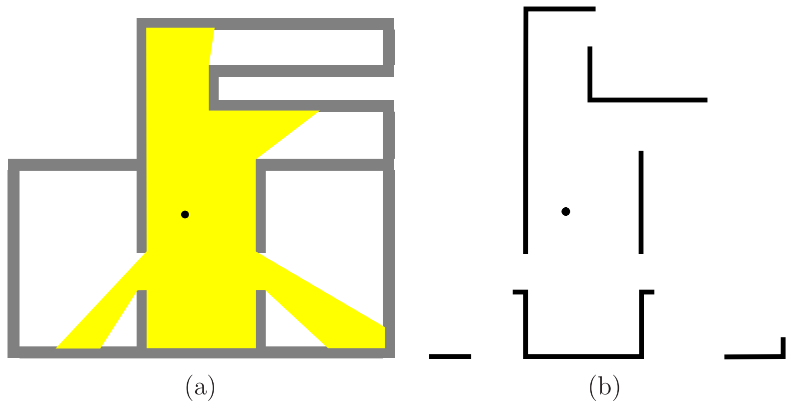


Figure 11.15: A range scanner or visibility sensor is like having a continuum of sonars, even with much higher accuracy. A distance value is provided for each $s \in \mathbb{S}^1$.

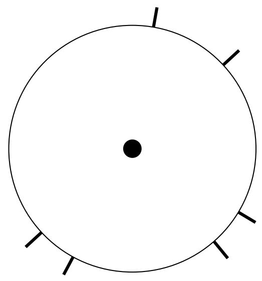


Figure 11.16: A gap sensor indicates only the directions at which discontinuities in depth occur, instead of providing distance information.

obtain a distance measurement for at least every 1/2 degree and sweep the full 360 degrees at least 30 times a second. The measurement accuracy in an indoor environment is often on the order of a few millimeters. Imagine the limiting case, which is like having a continuum of sonars, one for every angle in $\mathbb{S}^1$. This results in a sensor called a *range scanner* or *visibility sensor*, which provides a distance measurement for each $s \in \mathbb{S}^1$, as shown in Figure 11.15.

A weaker sensor can be made by only indicating points in $\mathbb{S}^1$ at which discontinuities (or gaps) occur in the depth scan. Refer to this as a *gap sensor*; an example is shown in Figure 11.16. It might even be the case that only the circular ordering of these gaps is given around $\mathbb{S}^1$, without knowing the relative angles between them, or the distance to each gap. A planner based on this sensing model is presented in Section 12.3.4.

Odometry sensors A final category will be given, which provides interesting examples of history-based sensor mappings, as defined for discrete state spaces in Section 11.1.1. Mobile robots often have *odometry sensors*, which indicate how far the robot has traveled, based on the amount that the wheels have turned. Such measurements are often inaccurate because of wheel slippage, surface imperfections, and small modeling errors. For a given state history, $\tilde{x}_t$, a sensor can estimate the total distance traveled. For this model, $Y = [0, \infty)$ and $y = h(\tilde{x}_t)$, in which the argument, $\tilde{x}_t$, to h is the entire state history up to time t. Another way to model odometry is to have a sensor indicate the estimated distance traveled since the last stage. This avoids the dependency on the entire history, but it may be harder to model the resulting errors in distance estimation.

In some literature (e.g., [105]) the action history, $\tilde{u}_k$, is referred to as odometry. This interpretation is appropriate in some applications. For example, each action might correspond to turning the pedals one full revolution on a bicycle. The number of times the pedals have been turned could serve as an odometry reading. Since this information already appears in η_k, it is not modeled in this book as part of the sensing process. For the bicycle example, there might be an odometry sensor that bases its measurements on factors other than the pedal motions. It would be appropriate to model this as a history-based sensor.

Another kind of history-based sensor is to observe a *wall clock* that indicates how much time has passed since the initial stage. This, in combination with other information, such as the speed of the robot, could enable strong inferences to be made about the state.

11.5.2 Simple Projection Examples

This section gives examples of I-spaces for which the sensor mapping is $y = h(x)$ and h is a projection that reveals some of the state variables, while concealing others. The examples all involve continuous time, and the focus is mainly on the nondeterministic I-space $\mathcal{I}_{ndet}$. It is assumed that there are no actions, which means that $U = \emptyset$. Nature actions, $\Theta(x)$, however, will be allowed. Since there are no robot actions and no nature sensing actions, all of the uncertainty arises from the fact that h is a projection and the nature actions that affect the state transition equation are not known. This is a very important and interesting class of problems in itself. The examples can be further complicated by allowing some control from the action set, U; however, the purpose here is to illustrate I-space concepts. Therefore, it will not be necessary.

Example 11.20 (Moving on a Sine Curve) Suppose that the state space is the set of points that lie on the sine curve in the plane:

$$X = \{(x_1, x_2) \in \mathbb{R}^2 \mid x_2 = \sin x_1\}. \tag{11.72}$$

Let $U = \emptyset$, which results in no action history. The observation space is $Y = [-1, 1]$

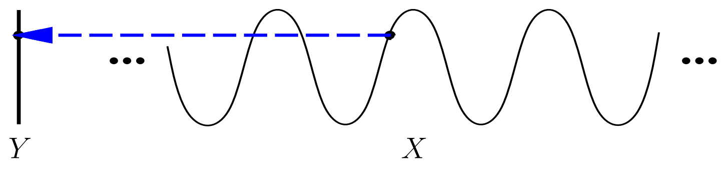


Figure 11.17: The state space is the set of points traced out by a sine curve in $\mathbb{R}^2$.

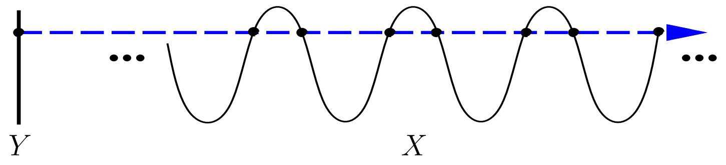


Figure 11.18: The preimage, $H(y)$, of an observation y is a countably infinite set of points along X.

and the sensor mapping yields $y = h(x) = x_2$, the height of the point on the sine curve, as shown in Figure 11.17.

The nature action space is $\Theta = \{-1, 1\}$, in which -1 means to move at unit speed in the $-x_1$ direction along the sine curve, and 1 means to move at unit speed in the x_1 direction along the curve. Thus, for some nature action history $\tilde{\theta}_t$, a state trajectory $\tilde{x}_t$ that moves the point along the curve can be determined by integration.

A history I-state takes the form $\eta_t = (X_0, \tilde{y}_t)$, which includes the initial condition $X_0 \subseteq X$ and the observation history $\tilde{y}_t$ up to time t. The nondeterministic I-states are very interesting for this problem. For each observation y, the preimage $H(y)$ is a countably infinite set of points that corresponds to the intersection of X with a horizontal line at height y, as shown in Figure 11.18.

The uncertainty for this problem is always characterized by the number of intersection points that might contain the true state. Suppose that $X_0 = X$. In this case, there is no state trajectory that can reduce the amount of uncertainty. As the point moves along X, the height is always known because of the sensor, but the x_1 coordinate can only be narrowed down to being any of the intersection points.

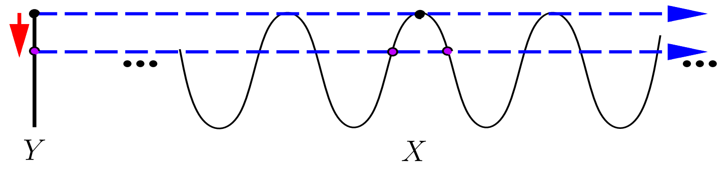


Figure 11.19: A bifurcation occurs when $y = 1$ or $y = -1$ is received. This irreversibly increases the amount of uncertainty in the state.

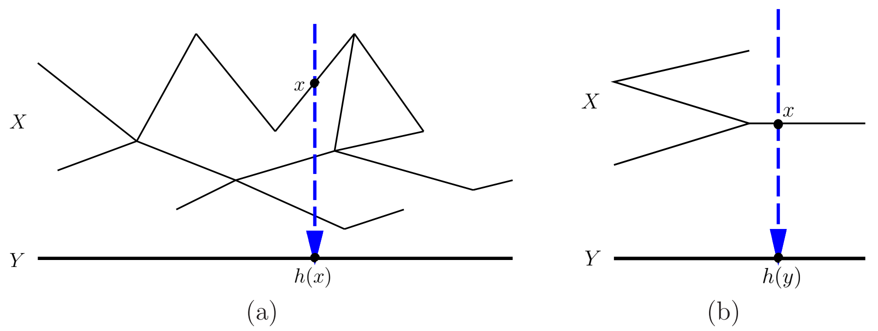


Figure 11.20: (a) Imagine trying to infer the location of a point on a planar graph while observing only a single coordinate. (b) This simple example involves a point moving along a graph that has four edges. When the point is on the rightmost edge, there is no uncertainty; however, uncertainty exists when the point travels along the other edges.

Suppose instead that $X_0 = \{x_0\}$, in which x_0 is some particular point along X. If y remains within $(0, 1)$ over some any period of time starting at $t = 0$, then $x(t)$ is known because the exact segment of the sine curve that contains the state is known. However, if the point reaches an extremum, which results in $y = 0$ or $y = 1$, then it is not known which way the point will travel. From this point, the sensor cannot disambiguate moving in the $-x_1$ direction from the x_1 direction. Therefore, the uncertainty grows, as shown in Figure 11.19. After the observation $y = 1$ is obtained, there are two possibilities for the current state, depending on which action was taken by nature when $y = 1$; hence, the nondeterministic I-state contains two states. If the motion continues until $y = -1$, then there will be four states in the nondeterministic I-state. Unfortunately, the uncertainty can only grow in this example. There is no way to use the sensor to reduce the size of the nondeterministic I-states. ■

The previous example can be generalized to observing a single coordinate of a point that moves around in a planar topological graph, as shown in Figure 11.20a. Most of the model remains the same as for Example 11.20, except that the state space is now a graph. The set of nature actions, $\Theta(x)$, needs to be extended so that if x is a vertex of the graph, then there is one input for each incident edge. These are the possible directions along which the point could move.

Example 11.21 (Observing a Point on a Graph) Consider the graph shown in Figure 11.20b, in which there are four edges.[8] When the point moves on the interior of the rightmost edge of the graph, then the state can be inferred from

[8]This example was significantly refined after a helpful discussion with Rob Ghrist.

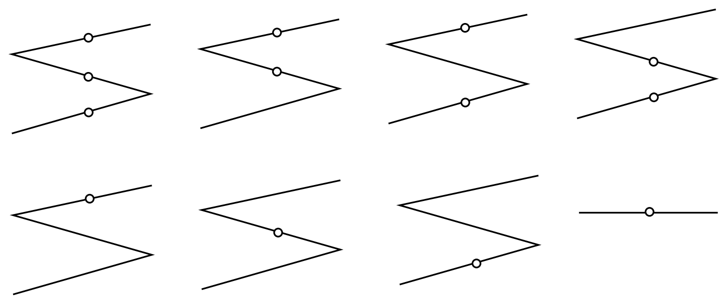


Figure 11.21: Pieces of the nondeterministic I-space $\mathcal{I}_{ndet}$ are obtained by the different possible sets of edges on which the point may lie.

the sensor. The set $H(y)$ contains a single point on the rightmost edge. If the point moves in the interior of one of the other edges, then $H(y)$ contains three points, one for each edge above y. This leads to seven possible cases for the nondeterministic I-state, as shown in Figure 11.21. Any subset of these edges may be possible for the nondeterministic I-state, except for the empty set.

The eight pieces of $\mathcal{I}_{ndet}$ depicted in Figure 11.21 are connected together in an interesting way. Suppose that the point is on the rightmost edge and moves left. After crossing the vertex, the I-state must be the case shown in the upper right of Figure 11.21, which indicates that the point could be on one of two edges. If the point travels right from one of the I-states of the left edges, then the I-state shown in the bottom right of Figure 11.20 is always reached; however, it is not necessarily possible to return to the same I-state on the left. Thus, in general, there are directional constraints on $\mathcal{I}_{ndet}$. Also, note that from the I-state on the lower left of Figure 11.20, it is impossible to reach the I-state on the lower right by moving straight right. This is because it is known from the structure of the graph that this is impossible. ■

The graph example can be generalized substantially to reflect a wide variety of problems that occur in robotics and other areas. For example, Figure 11.22 shows a polygon in which a point can move. Only one coordinate is observed, and the resulting nondeterministic I-space has layers similar to those obtained for Example 11.21. These ideas can be generalized to any dimension. Interesting models can be constructed using the simple projection sensors, such as a position sensor or compass, from Section 11.5.1. In Section 12.4, such layers will appear in a pursuit-evasion game that uses visibility sensors to find moving targets.

11.5.3 Examples with Nature Sensing Actions

This section illustrates the effect of nature sensing actions, but only for the nondeterministic case. General methods for computing probabilistic I-states are covered

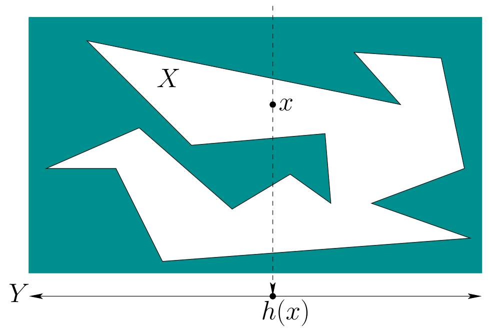


Figure 11.22: The graph can be generalized to a planar region, and layers in the nondeterministic I-space will once again be obtained.

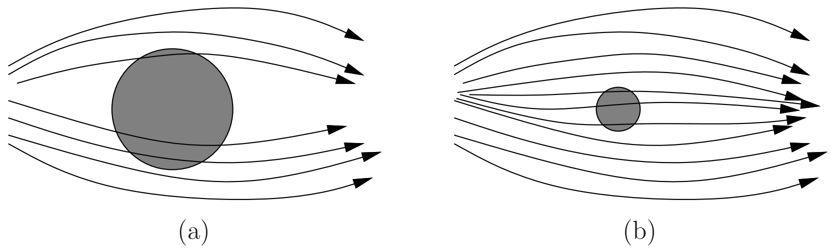


Figure 11.23: (a) It is always possible to determine whether the state trajectory went above or below the designated region. (b) Now the ability to determine whether the trajectory went above or below the hole depends on the particular observations. In some cases, it may not be possible.

in Section 11.6.

Example 11.22 (Above or Below Disc?) This example involves continuous time. Suppose that the task is to gather information and determine whether the state trajectory travels above or below some designated region of the state space, as shown in Figure 11.23.

Let $X = \mathbb{R}^2$. Motions are generated by integrating the velocity $(\dot{x}, \dot{y})$, which is expressed as $\dot{x} = \cos(u(t) + \theta(t))$ and $\dot{y} = \sin(u(t) + \theta(t))$. For simplicity, assume $u(t) = 0$ is applied for all time, which is a command to move right. The nature action $\theta(t) \in \Theta = [-\pi/4, \pi/4]$ interferes with the outcome. The robot tries to make progress by moving in the positive x_1 direction; however, the interference of nature makes it difficult to predict the x_2 direction. Without nature, there should be no change in the x_2 coordinate; however, with nature, the error in the x_2 direction could be as much as t, after t seconds have passed. Figure 11.24 illustrates the possible resulting motions.

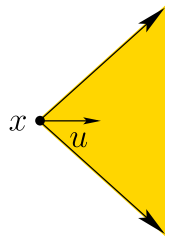


Figure 11.24: Nature interferes with the commanded direction, so that the true state could be anywhere within a circular section.

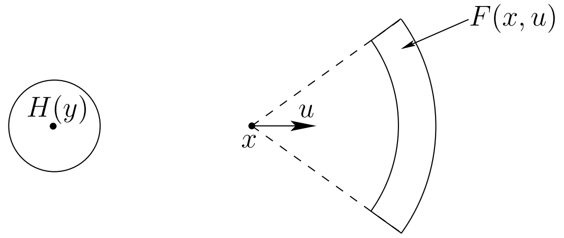


Figure 11.25: A simple mobile robot motion model in which the sensing model is as given in Figure 11.11 and then nature interferes with commanded motions to yield an uncertainty region that is a circular ring.

Sensor observations will be made that alleviate the growing cone of uncertainty; use the sensing model from Figure 11.11, and suppose that the measurement error r is 1. Suppose there is a disc in $\mathbb{R}^2$ of radius larger than 1, as shown in Figure 11.23a. Since the true state is never further than 1 from the measured state, it is always possible to determine whether the state passed above or below the disc. Multiple possible observation histories are shown in Figure 11.23a. The observation history need not even be continuous, but it is drawn that way for convenience. For a disc with radius less than 1, there may exist some observation histories for which it is impossible to determine whether the true state traveled above or below the disc; see Figure 11.23b. For other observation histories, it may still be possible to make the determination; for example, from the uppermost trajectory shown in Figure 11.23b it is known for certain that the true state traveled above the disc. ■

Example 11.23 (A Simple Mobile Robot Model) In this example, suppose that a robot is modeled as a point that moves in $X = \mathbb{R}^2$. The sensing model is the same as in Example 11.22, except that discrete stages are used instead of continuous time. It can be imagined that each stage represents a constant interval of time (e.g., 1 second).

To control the robot, a motion command is given in the form of an action $u_k \in U = \mathbb{S}^1$. Nature interferes with the motions in two ways: 1) The robot tries to travel some distance d, but there is some error $\epsilon_d > 0$, for which the true distance traveled, d', is known satisfy $|d' - d| < \epsilon_d$; and 2) the robot tries to move in a direction u, but there is some error, $\epsilon_u > 0$, for which the true direction u' is known to satisfy $|u - u'| < \epsilon_u$. These two independent errors can be modeled by defining a 2D nature action set, $\Theta(x)$. The transition equation is then defined so that the forward projection $F(x,u)$ is as shown in Figure 11.25.

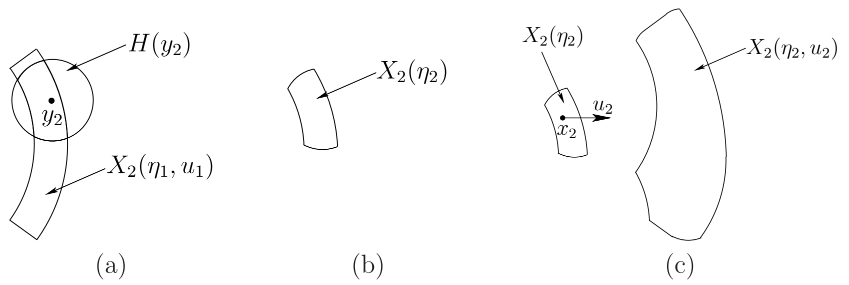


Figure 11.26: (a) Combining information from $X_2(\eta_1,u_1)$ and the observation y_2; (b) the intersection must be taken between $X_2(\eta_1,u_1)$ and $H(y_2)$. (c) The action u_2 leads to a complicated nondeterministic I-state that is the union of $F(x_2,u_2)$ over all $x_2 \in X_2(\eta_2)$.

Some nondeterministic I-states will now be constructed. Suppose that the initial state x_1 is known, and history I-states take the form

$$\eta_k = (x_1, u_1, \ldots, u_{k-1}, y_1, \ldots, y_k). \tag{11.73}$$

The first sensor observation, y_1, is useless because the initial state is known. Equation (11.29) is applied to yield $H(y_1) \cap \{x_1\} = \{x_1\}$. Suppose that the action $u_1 = 0$ is applied, indicating that the robot should move horizontally to the right. Equation (11.30) is applied to yield $X_2(\eta_1,u_1)$, which looks identical to the $F(x,u)$ shown in Figure 11.25. Suppose that an observation y_2 is received as shown in Figure 11.26a. Using this, $X_2(\eta_2)$ is computed by taking the intersection of $H(y_2)$ and $X_2(\eta_1,u_1)$, as shown in Figure 11.26b.

The next step is considerably more complicated. Suppose that $u_2 = 0$ and that (11.30) is applied to compute $X_3(\eta_2,u_2)$ from $X_2(\eta_2)$. The shape shown in Figure 11.26c is obtained by taking the union of $F(x_2,u_2)$ for all possible $x_2 \in X_2(\eta_2)$. The resulting shape is composed of circular arcs and straight line segments (see Exercise 13). Once y_3 is obtained, an intersection is taken once again to yield $X_3(\eta_3) = X_3(\eta_2,u_2) \cap H(y_3)$, as shown in Figure 11.27. The process repeats in the same way for the desired number of stages. The complexity of the region in Figure 11.26c provides motivation for the approximation methods of Section 11.4.3. For example, the nondeterministic I-states could be nicely approximated

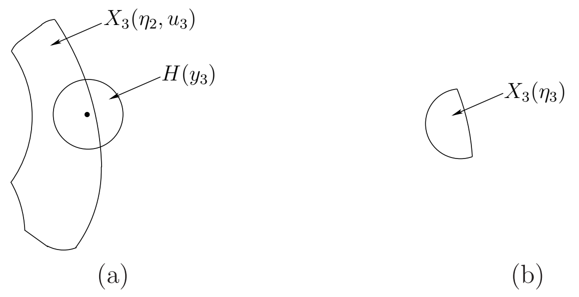


Figure 11.27: After the sensor observation, y_3, the intersection must be taken between $X_3(\eta_2, u_2)$ and $H(y_3)$.

by ellipsoidal regions. ■

11.5.4 Gaining Information Without Sensors

For some problems, it is remarkable that uncertainty may be reduced without even using sensors. Recall Example 11.17. This is counterintuitive because it seems that information regarding the state can only be gained from sensing. It is possible, however, to also gain information from the knowledge that some actions have been executed and the effect that should have in terms of the state transitions. The example presented in this section is inspired by work on *sensorless manipulation planning* [97, 114], which is covered in more detail in Section 12.5.2. This topic underscores the advantages of reasoning in terms of an I-space, as opposed to requiring that accurate state estimates can be made.

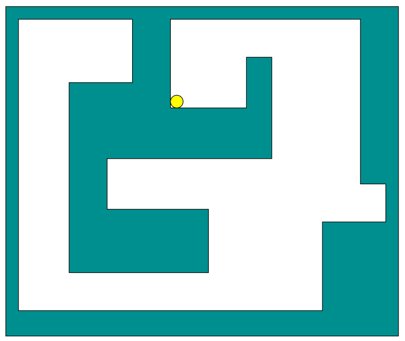


Figure 11.28: A top view of a tray that must be tilted to roll the ball into the desired corner.

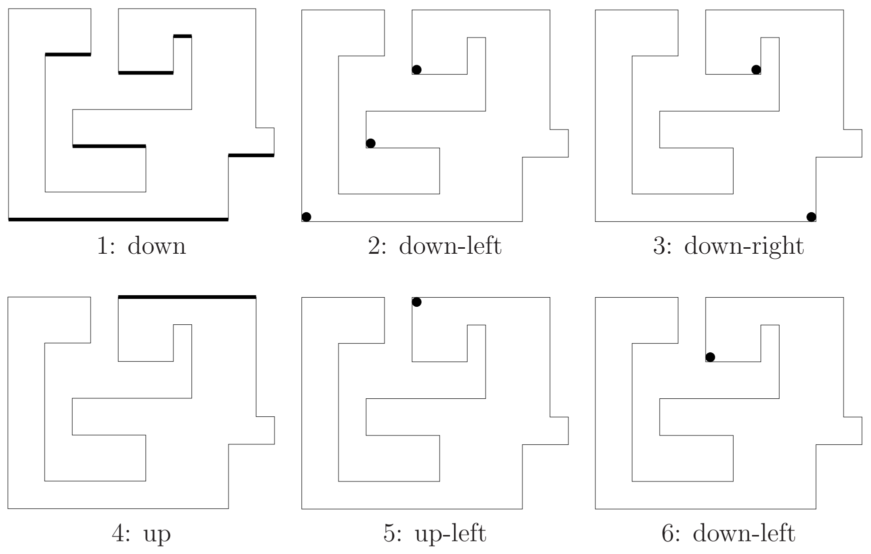


Figure 11.29: A plan is shown that places the ball in the desired location using a sequence of six tilts, regardless of its initial position and in spite of the fact that there are no sensors. The thickened black lines and black dots indicate the possible locations for the ball: the nondeterministic I-states. Under each picture, the direction that the ball rolls due to the action is written.

Example 11.24 (Tray Tilting) The state space, $X \subset \mathbb{R}^2$, indicates the position of a ball that rolls on a flat surface, as shown Figure 11.28. The ball is confined to roll within the polygonal region shown in the figure. It can be imagined that the ball rolls in a tray on which several barriers have been glued to confine its motion (try this experiment at home!). If the tray is tilted, it is assumed that the ball rolls in a direction induced by gravity (in the same way that a ball rolls to the bottom of a pinball machine).

The tilt of the tray is considered as an action that can be chosen by the robot. It is assumed that the initial position of the ball (initial state) is unknown and there are no sensors that can be used to estimate the state. The task is to find some tilting motions that are guaranteed to place the ball in the position shown in Figure 11.28, regardless of its initial position.

The problem could be modeled with continuous time, but this complicates the design. If the tray is tilted in a particular orientation, it is assumed that the ball rolls in a direction, possibly following the boundary, until it comes to rest. This can be considered as a discrete-stage transition: The ball is in some rest state, a tilt action is applied, and a then it enters another rest state. Thus, a discrete-stage state transition equation, $x_{k+1} = f(x_k, u_k)$, is used.

To describe the tilting actions, we can formally pick directions for the upward normal vector to the tray from the upper half of $\mathbb{S}^2$; however, this can be reduced to a one-dimensional set because the steepness of the tilt is not important, as long as the ball rolls to its new equilibrium state. Therefore, the set of actions can be considered as $U = \mathbb{S}^1$, in which a direction $u \in \mathbb{S}^1$ indicates the direction that the ball rolls due to gravity. Before any action is applied, it is assumed that the tray is initially level (its normal is parallel to the direction of gravity). In practice, one should be more careful and model the motion of the tray between a pair of actions; this is neglected here because the example is only for illustrative purposes. This extra level of detail could be achieved by introducing new state variables that indicate the orientation of the tray or by using continuous-time actions. In the latter case, the action is essentially providing the needed state information, which means that the action function would have to be continuous. Here it is simply assumed that a sequence of actions from $\mathbb{S}^1$ is applied.

The initial condition is $X_1 = X$ and the history I-state is

$$\eta_k = (X_1, u_1, u_2, \ldots, u_{k-1}). \tag{11.74}$$

Since there are no observations, the path through the I-space is predictable. Therefore, a plan, π, is simply an action sequence, $\pi = (u_1, u_2, \ldots, u_K)$, for any desired K.

It is surprisingly simple to solve this task by reasoning in terms of nondeterministic I-states, each of which corresponds to a set of possible locations for the ball. A sequence of six actions, as shown in Figure 11.29, is sufficient to guarantee that the ball will come to rest at the goal position, regardless of its initial position. ■

11.6 Computing Probabilistic Information States

The probabilistic I-states can be quite complicated in practice because each element of $\mathcal{I}_{prob}$ is a probability distribution or density function. Therefore, substantial effort has been invested in developing efficient techniques for computing probabilistic I-states efficiently. This section can be considered as a continuation of the presentations in Sections 11.2.3 (and part of Section 11.4, for the case of continuous state spaces). Section 11.6.1 covers Kalman filtering, which provides elegant computations of probabilistic I-states. It is designed for problems in which the state transitions and sensor mapping are linear, and all acts of nature are modeled by multivariate Gaussian densities. Section 11.6.2 covers a general sampling-based planning approach, which is approximate but applies to a broader class of problems. One of these methods, called *particle filtering*, has become very popular in recent years for mobile robot localization.

11.6.1 Kalman Filtering

This section covers the most successful and widely used example of a derived I-space that dramatically collapses the history I-space. In the special case in which both f and h are linear functions, and $p(\theta)$, $p(\psi)$, and $p(x_1)$ are Gaussian, all probabilistic I-states become Gaussian. This means that the probabilistic I-space, $\mathcal{I}_{prob}$, does not need to represent every conceivable probability density function. The probabilistic I-state is always trapped in the subspace of $\mathcal{I}_{prob}$ that corresponds only to Gaussians. The subspace is denoted as $\mathcal{I}_{gauss}$. This implies that an I-map, $\kappa_{mom} : \mathcal{I}_{prob} \rightarrow \mathcal{I}_{gauss}$, can be applied without any loss of information.

The model is called *linear-Gaussian* (or *LG*). Each Gaussian density on $\mathbb{R}^n$ is fully specified by its n-dimensional mean vector μ and an $n \times n$ symmetric covariance matrix, Σ. Therefore, $\mathcal{I}_{gauss}$ can be considered as a subset of $\mathbb{R}^m$ in which $m = 2n + \binom{n}{2}$. For example, if $X = \mathbb{R}^2$, then $\mathcal{I}_{gauss} \subset \mathbb{R}^5$, because two independent parameters specify the mean and three independent parameters specify the covariance matrix (not four, because of symmetry). It was mentioned in Section 11.4.3 that moment-based approximations can be used in general; however, for an LG model it is important to remember that $\mathcal{I}_{gauss}$ is an *exact* representation of $\mathcal{I}_{prob}$.

In addition to the fact that the $\mathcal{I}_{prob}$ collapses nicely, κ_{mom} is a sufficient I-map, and convenient expressions exist for incrementally updating the derived I-states entirely in terms of the computed means and covariance. This implies that we can work directly with $\mathcal{I}_{gauss}$, without any regard for the original histories or even the general formulas for the probabilistic I-states from Section 11.4.1. The update expressions are given here without the full explanation, which is lengthy but not difficult and can be found in virtually any textbook on stochastic control (e.g., [25, 151]).

For Kalman filtering, all of the required spaces are Euclidean, but they may have different dimensions. Therefore, let $X = \mathbb{R}^n$, $U = \Theta = \mathbb{R}^m$, and $Y = \Psi = \mathbb{R}^r$. Since Kalman filtering relies on linear models, everything can be expressed in terms of matrix transformations. Let A_k, B_k, C_k, G_k, and H_k each denote a matrix with constant real-valued entries and which may or may not be singular. The dimensions of the matrices will be inferred from the equations in which they will appear (the dimensions have to be defined correctly to make the multiplications work out right). The k subscript is used to indicate that a different matrix may be used in each stage. In many applications, the matrices will be the same in each stage, in which case they can be denoted by A, B, C, G, and H. Since Kalman filtering can handle the more general case, the subscripts are included (even though they slightly complicate the expressions).

In general, the state transition equation, $x_{k+1} = f_k(x_k, u_k, \theta_k)$, is defined as

$$x_{k+1} = A_k x_k + B_k u_k + G_k \theta_k, \tag{11.75}$$

in which the matrices A_k, B_k, and G_k are of appropriate dimensions. The notation

f_k is used instead of f, because the Kalman filter works even if f is different in every stage.

Example 11.25 (Linear-Gaussian Example) For a simple example of (11.75), suppose $X = \mathbb{R}^3$ and $U = \Theta = \mathbb{R}^2$. A particular instance is

$$x_{k+1} = \begin{pmatrix} 0 & \sqrt{2} & 1 \\ 1 & -1 & 4 \\ 2 & 0 & 1 \end{pmatrix} x_k + \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} u_k + \begin{pmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \theta_k. \tag{11.76}$$

■

The general form of the sensor mapping $y_k = h_k(x_k, \psi_k)$ is

$$y_k = C_k x_k + H_k \psi_k, \tag{11.77}$$

in which the matrices C_k and H_k are of appropriate dimension. Once again, h_k is used instead of h because a different sensor mapping can be used in every stage.

So far the linear part of the model has been given. The next step is to specify the Gaussian part. In each stage, both nature actions θ_k and ψ_k are modeled with zero-mean Gaussians. Thus, each has an associated covariance matrix, denoted by Σ_θ and Σ_ψ, respectively. Using the model given so far and starting with an initial Gaussian density over X, all resulting probabilistic I-states will be Gaussian [151].

Every derived I-state in $\mathcal{I}_{gauss}$ can be represented by a mean and covariance. Let μ_k and Σ_k denote the mean and covariance of $P(x_k|\eta_k)$. The expressions given in the remainder of this section define a derived information transition equation that computes μ_{k+1} and Σ_{k+1}, given μ_k, Σ_k, u_k, and y_{k+1}. The process starts by computing μ_1 and Σ_1 from the initial conditions.

Assume that an initial condition is given that represents a Gaussian density over $\mathbb{R}^n$. Let this be denoted by μ_0, and Σ_0. The first I-state, which incorporates the first observation y_1, is computed as $\mu_1 = \mu_0 + L_1(y_1 - C_1\mu_0)$ and

$$\Sigma_1 = (I - L_1 C_1)\Sigma_0, \tag{11.78}$$

in which I is the identity matrix and

$$L_1 = \Sigma_0 C_1^T \big(C_1 \Sigma_0 C_1^T + H_1 \Sigma_\psi H_1\big)^{-1}. \tag{11.79}$$

Although the expression for L_1 is complicated, note that all matrices have been specified as part of the model. The only unfortunate part is that a matrix inversion is required, which sometimes leads to numerical instability in practice; see [151] or other sources for an alternative formulation that alleviates this problem.

Now that μ_1 and Σ_1 have been expressed, the base case is completed. The next part is to give the iterative updates from stage k to stage $k+1$. Using μ_k, the mean at the next stage is computed as

$$\mu_{k+1} = A_k\mu_k + B_k u_k + L_{k+1}(y_{k+1} - C_{k+1}(A_k\mu_k + B_k u_k)), \tag{11.80}$$

in which L_{k+1} will be defined shortly. The covariance is computed in two steps; one is based on applying u_k, and the other arises from considering y_{k+1}. Thus, after u_k is applied, the covariance becomes

$$\Sigma'_{k+1} = A_k \Sigma_k A_k^T + G_k \Sigma_\theta G_k^T. \tag{11.81}$$

After y_{k+1} is received, the covariance Σ_{k+1} is computed from Σ'_{k+1} as

$$\Sigma_{k+1} = (I - L_{k+1} C_{k+1})\Sigma'_{k+1}. \tag{11.82}$$

The expression for L_k is

$$L_k = \Sigma'_k C_k^T \big(C_k \Sigma'_k C_k^T + H_k \Sigma_\psi H_k\big)^{-1}. \tag{11.83}$$

To obtain L_{k+1}, substitute $k+1$ for k in (11.83). Note that to compute μ_{k+1} using (11.80), Σ'_{k+1} must first be computed because (11.80) depends on L_{k+1}, which in turn depends on Σ'_{k+1}.

The most common use of the Kalman filter is to provide reliable estimates of the state x_k by using μ_k. It turns out that the optimal expected-cost feedback plan for a cost functional that is a quadratic form can be obtained for LG systems in a closed-from expression; see Section 15.2.2. This model is often called LQG, to reflect the fact that it is linear, quadratic-cost, and Gaussian. The optimal feedback plan can even be expressed directly in terms of μ_k, without requiring Σ_k. This indicates that the I-space may be collapsed down to X; however, the corresponding I-map is not sufficient. The covariances are still needed to compute the means, as is evident from (11.80) and (11.83). Thus, an optimal plan can be specified as $\pi : X \to U$, but the derived I-states in $\mathcal{I}_{gauss}$ need to be represented for the I-map to be sufficient.

The Kalman filter provides a beautiful solution to the class of linear Gaussian models. It is even successfully applied quite often in practice for problems that do not even satisfy these conditions. This is called the *extended Kalman filter*. The success may be explained by recalling that the probabilistic I-space may be approximated by mean and covariance in a second-order moment-based approximation. In general, such an approximation may be inappropriate, but it is nevertheless widely used in practice.

11.6.2 Sampling-Based Approaches

Since probabilistic I-space computations over continuous spaces involve the evaluation of complicated, possibly high-dimensional integrals, there is strong motivation for using sampling-based approaches. If a problem is nonlinear and/or non-Gaussian, such approaches may provide the only practical way to compute probabilistic I-states. Two approaches are considered here: grid-based sampling and particle filtering. One of the most common applications of the techniques described here is mobile robot localization, which is covered in Section 12.2.

A grid-based approach Perhaps the most straightforward way to numerically compute probabilistic I-states is to approximate probability density functions over a grid and use numerical integration to evaluate the integrals in (11.57) and (11.58).

A grid can be used to compute a discrete probability distribution that approximates the continuous probability density function. Consider, for example, using the Sukharev grid shown in Figure 5.5a, or a similar grid adapted to the state space. Consider approximating some probability density function $p(x)$ using a finite set, $S \subset X$. The Voronoi region surrounding each point can be considered as a "bucket" that holds probability mass. A probability is associated with each sample and is defined as the integral of $p(x)$ over the Voronoi region associated with the point. In this way, the samples S and their discrete probability distribution, $P(s)$ for all $s \in S$ approximate $p(x)$ over X. Let $P(s_k)$ denote the probability distribution over S_k, the set of grid samples at stage k.

In the initial step, $P(s)$ is computed from $p(x)$ by numerically evaluating the integrals of $p(x_1)$ over the Voronoi region of each sample. This can alternatively be estimated by drawing random samples from the density $p(x_1)$ and then recording the number of samples that fall into each bucket (Voronoi region). Normalizing the counts for the buckets yields a probability distribution, $P(s_1)$. Buckets that have little or no points can be eliminated from future computations, depending on the desired accuracy. Let S_1 denote the samples for which nonzero probabilities are associated.

Now suppose that $P(s_k|\eta_k)$ has been computed over S_k and the task is to compute $P(s_{k+1}|\eta_{k+1})$ given u_k and y_{k+1}. A discrete approximation, $P(s_{k+1}|s_k, u_k)$, to $p(x_{k+1}|x_k, u_k)$ can be computed using a grid and buckets in the manner described above. At this point the densities needed for (11.57) have been approximated by discrete distributions. In this case, (11.38) can be applied over S_k to obtain a grid-based distribution over S_{k+1} (again, any buckets that do not contain enough probability mass can be discarded). The resulting distribution is $P(s_{k+1}|\eta_k, u_k)$, and the next step is to consider y_{k+1}. Once again, a discrete distribution can be computed; in this case, $p(x_{k+1}|y_{k+1})$ is approximated by $P(s_{k+1}|y_{k+1})$ by using the grid samples. This enables (11.58) to be replaced by the discrete counterpart (11.39), which is applied to the samples. The resulting distribution, $P(s_{k+1}|\eta_{k+1})$, represents the approximate probabilistic I-state.

Particle filtering As mentioned so far, the discrete distributions can be estimated by using samples. In fact, it turns out that the Voronoi regions over the samples do not even need to be carefully considered. One can work directly with a collection of samples drawn randomly from the initial probability density, $p(x_1)$. The general method is referred to as *particle filtering* and has yielded good performance in applications to experimental mobile robotics. Recall Figure 1.7 and see Section 12.2.3.

Let $S \subset X$ denote a finite collection of samples. A probability distribution is

defined over S. The collection of samples, together with its probability distribution, is considered as an approximation of a probability density over X. Since S is used to represent probabilistic I-states, let P_k denote the probability distribution over S_k, which is computed at stage k using the history I-state η_k. Thus, at every stage, there is a new sample set, S_k, and probability distribution, P_k.

The general method to compute the probabilistic I-state update proceeds as follows. For some large number, m, of iterations, perform the following:

1. Select a state $x_k \in S_k$ according to the distribution P_k.

2. Generate a new sample, x_{k+1}, for S_{k+1} by generating a single sample according to the density $p(x_{k+1}|x_k, u_k)$.

3. Assign the weight, $w(x_{k+1}) = p(y_{k+1}|x_{k+1})$.

After the m iterations have completed, the weights over S_{k+1} are normalized to obtain a valid probability distribution, P_{k+1}. It turns out that this method provides an approximation that converges to the true probabilistic I-states as m tends to infinity. Other methods exist, which provide faster convergence [143]. One of the main difficulties with using particle filtering is that for some problems it is difficult to ensure that a sufficient concentration of samples exists in the places where they are needed the most. This is a general issue that plagues many sampling-based algorithms, including the motion planning algorithms of Chapter 5.

11.7 Information Spaces in Game Theory

This section unifies the sequential game theory concepts from Section 10.5 with the I-space concepts from this chapter. Considerable attention is devoted to the modeling of information in game theory. The problem is complicated by the fact that each player has its own frame of reference, and hence its own I-space. Game solution concepts, such as saddle points or Nash equilibria, depend critically on the information available to each player as it makes it decisions. Paralleling Section 10.5, the current section first covers I-states in game trees, followed by I-states for games on state spaces. The presentation in this section will be confined to the case in which the state space and stages are finite. The formulation of I-spaces extends naturally to countably infinite or continuous state spaces, action spaces, and stages [9].

11.7.1 Information States in Game Trees

Recall from Section 10.5.1 that an important part of formulating a sequential game in a game tree is specifying the information model. This was described in Step 4 of Formulation 10.3. Three information models were considered in Section

10.5.1: alternating play, stage-by-stage, and open loop. These and many other information models can be described using I-spaces.

From Section 11.1, it should be clear that an I-space is always defined with respect to a state space. Even though Section 10.5.1 did not formally introduce a state space, it is not difficult to define one. Let the state space X be N, the set of all vertices in the game tree. Assume that two players are engaged in a sequential zero-sum game. Using notation from Section 10.5.1, N_1 and N_2 are the decision vertices of P_1 and P_2, respectively. Consider the nondeterministic I-space $\mathcal{I}_{ndet}$ over N. Let η denote a nondeterministic I-state; thus, each $\eta \in \mathcal{I}_{ndet}$ is a subset of N.

There are now many possible ways in which the players can be confused while making their decisions. For example, if some η contains vertices from both N_1 and N_2, the player does not know whether it is even its turn to make a decision. If η additionally contains some leaf vertices, the game may be finished without a player even being aware of it. Most game tree formulations avoid these strange situations. It is usually assumed that the players at least know when it is their turn to make a decision. It is also usually assumed that they know the stage of the game. This eliminates many sets from $\mathcal{I}_{ndet}$.

While playing the game, each player has its own nondeterministic I-state because the players may hide their decisions from each other. Let η_1 and η_2 denote the nondeterministic I-states for P_1 and P_2, respectively. For each player, many sets in $\mathcal{I}_{ndet}$ are eliminated. Some are removed to avoid the confusions mentioned above. We also impose the constraint that $\eta_i \subseteq N_i$ for $i = 1$ and $i = 2$. We only care about the I-state of a player when it is that player's turn to make a decision. Thus, the nondeterministic I-state should tell us which decision vertices in N_i are possible as P_i faces a decision. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ represent the nondeterministic I-spaces for P_1 and P_2, respectively, with all impossible I-states eliminated.

The I-spaces $\mathcal{I}_1$ and $\mathcal{I}_2$ are usually defined directly on the game tree by circling vertices that belong to the same I-state. They form a partition of the vertices in each level of the tree (except the leaves). In fact, $\mathcal{I}_i$ even forms a partition of N_i for each player. Figure 11.30 shows four information models specified in this way for the example in Figure 10.13. The first three correspond directly to the models allowed in Section 10.5.1. In the alternating-play model, each player always knows the decision vertex. This corresponds to a case of perfect state information. In the stage-by-stage model, P_1 always knows the decision vertex; P_2 knows the decision vertex from which P_1 made its last decision, but it does not know which branch was chosen. The open-loop model represents the case that has the poorest information. Only P_1 knows its decision vertex at the beginning of the game. After that, there is no information about the actions chosen. In fact, the players cannot even remember their own previous actions. Figure 11.30d shows an information model that does not fit into any of the three previous ones. In this model, very strange behavior results. If P_1 and P_2 initially choose right branches, then the resulting decision vertex is known; however, if P_2 instead chooses the left branch,

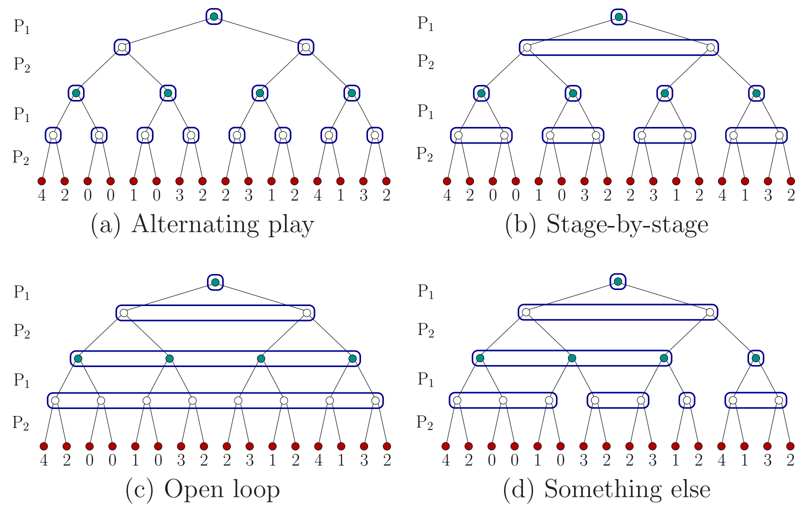


Figure 11.30: Several different information models are illustrated for the game in Figure 10.13.

then P_1 will forget which action it applied (as if the action of P_2 caused P_1 to have amnesia!). Here is a single-stage example:

Example 11.26 (An Unusual Information Model) Figure 11.31 shows a game that does not fit any of the information models in Section 10.5.1. It is actually a variant of the game considered before in Figure 10.12. The game is a kind of hybrid that partly looks like the alternating-play model and partly like the stage-by-stage model. This particular problem can be solved in the usual way, from the bottom up. A value is computed for each of the nondeterministic I-states, for the level in which P_2 makes a decision. The left I-state has value 5, which corresponds to P_1 choosing 1 and P_2 responding with 3. The right I-state has value 4, which results from the deterministic saddle point in a 2×3 matrix game played between P_1 and P_2. The overall game has a deterministic saddle point in which P_1 chooses 3 and P_2 chooses 3. This results in a value of 4 for the game. ■

Plans are now defined directly as functions on the I-spaces. A *(deterministic) plan for* P_1 is defined as a function π_1 on $\mathcal{I}_1$ that yields an action $u \in U(\eta_1)$ for each $\eta_1 \in \mathcal{I}_1$, and $U(\eta_1)$ is the set of actions that can be inferred from the I-state η_1; assume that this set is the same for all decision vertices in η_1. Similarly, a *(deterministic) plan for* P_2 is defined as a function π_2 on $\mathcal{I}_2$ that yields an action $v \in V(\eta_2)$ for each $\eta_2 \in \mathcal{I}_2$.

There are generally two alternative ways to define a randomized plan in terms of I-spaces. The first choice is to define a *globally randomized plan*, which is a

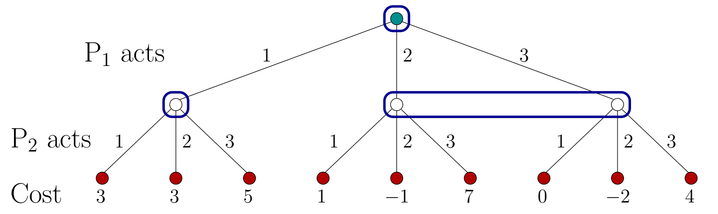


Figure 11.31: A single-stage game that has an information model unlike those in Section 10.5.1.

probability distribution over the set of all deterministic plans. During execution, this means that an entire deterministic plan will be sampled in advance according to the probability distribution. An alternative is to sample actions as they are needed at each I-state. This is defined as follows. For the randomized case, let $W(\eta_1)$ and $Z(\eta_2)$ denote the sets of all probability distributions over $U(\eta_1)$ and $V(\eta_2)$, respectively. A *locally randomized plan for* P$_1$ is defined as a function that yields some $w \in W(\eta_1)$ for each $\eta_1 \in \mathcal{I}_1$. Likewise, a *locally randomized plan for* P$_2$ is a function that maps from $\mathcal{I}_2$ into $Z(\eta_2)$. Locally randomized plans expressed as functions of I-states are often called *behavioral strategies* in game theory literature.

A randomized saddle point on the space of locally randomized plans does not exist for all sequential games [9]. This is unfortunate because this form of randomization seems most natural for the way decisions are made during execution. At least for the stage-by-stage model, a randomized saddle point always exists on the space of locally randomized plans. For the open-loop model, randomized saddle points are only guaranteed to exist using a globally randomized plan (this was actually done in Section 10.5.1). To help understand the problem, suppose that the game tree is a balanced, binary tree with k stages (hence, $2k$ levels). For each player, there are 2^k possible deterministic plans. This means that $2^k - 1$ probability values may be assigned independently (the last one is constrained to force them to sum to 1) to define a globally randomized plan over the space of deterministic plans. Defining a locally randomized plan, there are k I-states for each player, one for each search stage. At each stage, a probability distribution is defined over the action set, which contains only two elements. Thus, each of these distributions has only one independent parameter. A randomized plan is specified in this way using $k - 1$ independent parameters. Since $k - 1$ is much less than $2^k - 1$, there are many globally randomized plans that cannot be expressed as a locally randomized plan. Unfortunately, in some games the locally randomized representation removes the randomized saddle point.

This strange result arises mainly because players can forget information over time. A player with *perfect recall* remembers its own actions and also never forgets any information that it previously knew. It was shown by Kuhn that the space of all globally randomized plans is equivalent to the space of all locally randomized plans if and only if the players have perfect memory [150]. Thus, by sticking to games in which all players have perfect recall, a randomized saddle point always exists in the space locally randomized plans. The result of Kuhn even holds for the more general case of the existence of randomized Nash equilibria on the space of locally randomized plans.

The nondeterministic I-states can be used in game trees that involve more players. Accordingly, deterministic, globally randomized, and locally randomized plans can be defined. The result of Kuhn applies to any number of players, which ensures the existence of a randomized Nash equilibrium on the space of locally randomized strategies if (and only if) the players have perfect recall. It is generally preferable to exploit this fact and decompose the game tree into smaller matrix games, as described in Section 10.5.1. It turns out that the precise condition that allows this is that it must be *ladder-nested* [9]. This means that there are decision vertices, other than the root, at which 1) the player that must make a decision knows it is at that vertex (the nondeterministic I-state is a singleton set), and 2) the nondeterministic I-state will not leave the subtree rooted at that vertex (vertices outside of the subtree cannot be circled when drawing the game tree). In this case, the game tree can be decomposed at these special decision vertices and replaced with the game value(s). Unfortunately, there is still the nuisance of multiple Nash equilibria.

It may seem odd that nondeterministic I-states were defined without being derived from a history I-space. Without much difficulty, it is possible to define a sensing model that leads to the nondeterministic I-states used in this section. In many cases, the I-state can be expressed using only a subset of the action histories. Let $\tilde{u}_k$ and $\tilde{v}_k$ denote the action histories of P$_1$ and P$_2$, respectively. The history I-state for the alternating-play model at stage k is $(\tilde{u}_{k-1}, \tilde{v}_{k-1})$ for P$_1$ and $(\tilde{u}_k, \tilde{v}_{k-1})$ for P$_2$. The history I-state for the stage-by-stage model is $(\tilde{u}_{k-1}, \tilde{v}_{k-1})$ for both players. The nondeterministic I-states used in this section can be derived from these histories. For other models, such as the one in Figure 11.31, a sensing model is additionally needed because only partial information regarding some actions appears. This leads into the formulation covered in the next section, which involves both sensing models and a state space.

11.7.2 Information Spaces for Games on State Spaces

I-space concepts can also be incorporated into sequential games that are played over state spaces. The resulting formulation naturally extends Formulation 11.1 of Section 11.1 to multiple players. Rather than starting with two players and generalizing later, the full generality of having n players is assumed up front. The focus in this section is primarily on *characterizing* I-spaces for such games, rather than solving them. Solution approaches depend heavily on the particular information models; therefore, they will not be covered here.

As in Section 11.7.1, each player has its own frame of reference and therefore its own I-space. The I-state for each player indicates its information regarding a common game state. This is the same state as introduced in Section 10.5; however, each player may have different observations and may not know the actions of others. Therefore, the I-state is different for each decision maker. In the case of perfect state sensing, these I-spaces all collapse to X.

Suppose that there are n players. As presented in Section 10.5, each player has its own action space, U^i; however, here it is not allowed to depend on x, because the state may generally be unknown. It can depend, however, on the I-state. If nature actions may interfere with the state transition equation, then (10.120) is used (if there are two players); otherwise, (10.121) is used, which leads to predictable future states if the actions of all of the players are given. A single nature action, $\theta \in \Theta(x, u^1, u^2, \ldots, u^n)$, is used to model the effect of nature across all players when uncertainty in prediction exists.

Any of the sensor models from Section 11.1.1 may be defined in the case of multiple players. Each has its own observation space Y^i and sensor mapping h^i. For each player, nature may interfere with observations through nature sensing actions, $\Psi^i(x)$. A state-action sensor mapping appears as $y^i = h^i(x, \psi^i)$; state sensor mappings and history-based sensor mappings may also be defined.

Consider how the game appears to a single player at stage k. What information might be available for making a decision? Each player produces the following in the most general case: 1) an initial condition, η_0^i; 2) an action history, $\tilde{u}_{k-1}^i$; and 3) and an observation history, $\tilde{y}_k^i$. It must be specified whether one player knows the previous actions that have been applied by other players. It might even be possible for one player to receive the observations of other players. If P_i receives all of this information, its history I-state at stage k is

$$\eta_k^i = (\eta_0^i, \tilde{u}_{k-1}^1, \tilde{u}_{k-1}^2, \ldots, \tilde{u}_{k-1}^n, \tilde{y}_k^1, \tilde{y}_k^2, ..., \tilde{y}_k^n). \tag{11.84}$$

In most situations, however, η_k^i only includes a subset of the histories from (11.84). A typical situation is

$$\eta_k^i = (\eta_0^i, \tilde{u}_{k-1}^i, \tilde{y}_k^i), \tag{11.85}$$

which means that P_i knows only its own actions and observations. Another possibility is that all players know all actions that have been applied, but they do not receive the observations of other players. This results in

$$\eta_k^i = (\eta_0^i, \tilde{u}_{k-1}^1, \tilde{u}_{k-1}^2, \ldots, \tilde{u}_{k-1}^n, \tilde{y}_k^i). \tag{11.86}$$

Of course, many special cases may be defined by generalizing many of the examples in this chapter. For example, an intriguing sensorless game may be defined in which the history I-state consists only of actions. This could yield

$$\eta_k^i = (\eta_0^i, \tilde{u}_{k-1}^1, \tilde{u}_{k-1}^2, \ldots, \tilde{u}_{k-1}^n), \tag{11.87}$$

or even a more secretive game in which the actions of other players are not known:

$$\eta_k^i = (\eta_0^i, \tilde{u}_{k-1}^i). \tag{11.88}$$

Once the I-state has been decided upon, a history I-space $\mathcal{I}_{hist}^i$ for each player is defined as the set of all history I-states. In general, I-maps and derived I-spaces can be defined to yield alternative simplifications of each history I-space.

Assuming all spaces are finite, the concepts given so far can be organized into a sequential game formulation that is the imperfect state information counterpart of Formulation 10.4:

Formulation 11.4 (Sequential Game with I-Spaces)

1. A set of n players, P_1, P_2, ..., P_n.

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

3. For each P_i, a finite *action space* U^i. We also allow a more general definition, in which the set of available choices depends on the history I-state; this can be written as $U^i(\eta^i)$.

4. A finite *nature action space* $\Theta(x, u^1, \ldots, u^n)$ for each $x \in X$, and $u^i \in U^i$ for each i such that $1 \leq i \leq m$.

5. A *state transition function* f that produces a state, $f(x, u^1, \ldots, u^n, \theta)$, for every $x \in X$, $\theta \in \Theta(x, u)$, and $u^i \in U^i$ for each i such that $1 \leq i \leq n$.

6. For each P_i, a finite *observation space* Y^i.

7. For each P_i, a finite *nature sensing action space* $\Psi^i(x)$ for each $x \in X$.

8. For each P_i, a *sensor mapping* h^i which produces an observation, $y = h^i(x, \psi^i)$, for each $x \in X$ and $\psi^i \in \Psi^i(x)$. This definition assumes a state-nature sensor mapping. A state sensor mapping or history-based sensor mapping, as defined in Section 11.1.1, may alternatively be used.

9. A set of K *stages*, each denoted by k, which begins at $k = 1$ and ends at $k = K$. Let $F = K + 1$.

10. For each P_i, an *initial condition* η_0^i, which is an element of an *initial condition space* $\mathcal{I}_0^i$.

11. For each P_i, a *history I-space* $\mathcal{I}_{hist}^i$ which is the set of all history I-states, formed from action and observation histories, and may include the histories of other players.

12. For each P_i, let L^i denote a stage-additive cost functional,

$$L^i(\tilde{x}_F, \tilde{u}_K^1, \ldots, \tilde{u}_K^2) = \sum_{k=1}^{K} l(x_k, u_k^1, \ldots, u_k^n) + l_F(x_F). \tag{11.89}$$

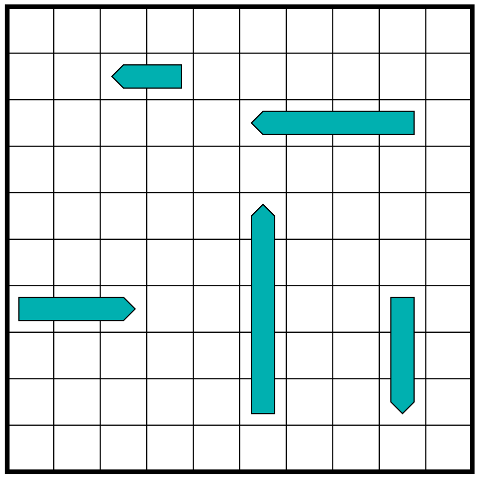


Figure 11.32: In the Battleship game, each player places several ships on a grid. The other player must guess the locations of ships by asking whether a particular tile is occupied.

Extensions exist for cases in which one or more of the spaces are continuous; see [9]. It is also not difficult to add goal sets and termination conditions and allow the stages to run indefinitely.

An interesting specialization of Formulation 11.4 is when all players have identical cost functions. This is not equivalent to having a single player because the players have different I-states. For example, a task may be for several robots to search for a treasure, but they have limited communication between them. This results in different I-states. They would all like to cooperate, but they are unable to do so without knowing the state. Such problems fall under the subject of *team theory* [57, 125, 142].

As for the games considered in Formulation 10.4, each player has its own plan. Since the players do not necessarily know the state, the decisions are based on the I-state. The definitions of a deterministic plan, a globally randomized plan, and a locally randomized plan are essentially the same as in Section 11.7.1. The only difference is that more general I-spaces are defined in the current setting. Various kinds of solution concepts, such as saddle points and Nash equilibria, can be defined for the general game in Formulation 11.4. The existence of locally randomized saddle points and Nash equilibria depends on general on the particular information model [9].

Example 11.27 (Battleship Game) Many interesting I-spaces arise from classical board games. A brief illustration is provided here from Battleship, which is a sequential game under the alternating-turn model. Two players, P_1 and P_2, each having a collection of battleships that it arranges secretly on a 10×10 grid;

see Figure 11.32.

A state is the specification of the exact location of all ships on each player's grid. The state space yields the set of all possible ship locations for both players. Each player always knows the location of its own ships. Once they are placed on the grid, they are never allowed to move.

The players take turns guessing a single grid tile, expressed as a row and column, that it suspects contains a ship. The possible observations are "hit" and "miss," depending on whether a ship was at that location. In each turn, a single guess is made, and the players continue taking turns until one player has observed a hit for every tile that was occupied by a ship.

This is an interesting game because once a "hit" is discovered, it is clear that a player should search for other hits in the vicinity because there are going to be several contiguous tiles covered by the same ship. The only problem is that the precise ship position and orientation are unknown. A good player essentially uses the nondeterministic I-state to improve the chances that a hit will occur next. ■

Example 11.28 (The Princess and the Monster) This is a classic example from game theory that involves no sensing. A princess and a monster move about in a 2D environment. A simple motion model is assumed; for example, they take single steps on a grid. The princess is trying not to be discovered by the monster, and the game is played in complete darkness. The game ends when the monster and the princess are on the same grid point. There is no form of feedback that can be used during the game; however, it is possible to construct nondeterministic I-states for the players. For most environments, it is impossible for the monster to be guaranteed to win; however, for some environments it is guaranteed to succeed. This example can be considered as a special kind of *pursuit-evasion game*. A continuous-time pursuit-evasion game that involves I-spaces is covered in Section 12.4. ■

Further Reading

The basic concept of an information space can be traced back to work of Kuhn [150] in the context of game trees. There, the nondeterministic I-state is referred to as an *information set*. After spreading throughout game theory, the concept was also borrowed into stochastic control theory (see [25, 151]). The term *information space* is used extensively in [9] in the context of sequential and differential game theory. For further reading on I-spaces in game theory, see [9, 210]. In artificial intelligence literature, I-states are referred to as *belief states* and are particularly important in the study of POMDPs; see the literature suggested at the end of Chapter 12. The *observability problem* in control theory also results in I-spaces [51, 89, 130, 266], in which *observers* are used to reconstruct the current state from the history I-state. In robotics literature, they have been called *hyperstates* [114] and *knowledge states* [94]. Concepts closely related to I-spaces also appear as *perceptual equivalence classes* in [80] and also appear in the *information invariants* framework of Donald [79]. I-spaces were proposed as a general way to represent planning under sensing uncertainty in [12, 165, 166]. For further reading on sensors in general, see [106].

The Kalman filter is covered in great detail in numerous other texts; see for example, [58, 151, 266]. The original reference is [137]. For more on particle filters, see [5, 81, 105, 143].

Exercises

1. Forward projections in $\mathcal{I}_{ndet}$:

   (a) Starting from a nondeterministic I-state, $X_k(\eta_k)$, and applying an action u_k, derive an expression for the nondeterministic one-stage forward projection by extending the presentation in Section 10.1.2.

   (b) Determine an expression for the two-stage forward projection starting from $X_k(\eta_k)$ and applying u_k and u_{k+1}.

2. Forward projections in $\mathcal{I}_{prob}$:

   (a) Starting from a probabilistic I-state, $P(x_k|\eta_k)$, and applying an action u_k, derive an expression for the probabilistic one-stage forward projection.

   (b) Determine an expression for the two-stage forward projection starting from $P(x_k|\eta_k)$ and applying u_k and u_{k+1}.

3. Determine the strong and weak backprojections on $\mathcal{I}_{hist}$ for a given history I-state, η_k. These should give sets of possible $\eta_{k-1} \in \mathcal{I}_{hist}$.

4. At the end of Section 11.3.2, it was mentioned that an equivalent DFA can be constructed from an NFA.

   (a) Give an explicit DFA that accepts the same set of strings as the NFA in Figure 11.8b.

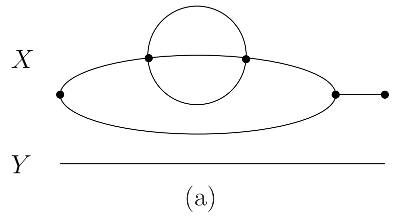
   (b) Express the problem of determining whether the NFA in Figure 11.8b accepts any strings as a planning problem using Formulation 2.1.

5. This problem involves computing probabilistic I-states for Example 11.14. Let the initial I-state be

$$P(x_1) = [1/3 \;\; 1/3 \;\; 1/3], \tag{11.90}$$

in which the ith entry in the vector indicates $P(x_1 = i + 1)$. Let $U = \{0, 1\}$. For each action, a state transition matrix can be specified, which gives the probabilities $P(x_{k+1}|x_k, u_k)$. For $u = 0$, let $P(x_{k+1}|x_k, u_k = 0)$ be

$$\begin{pmatrix} 4/5 & 1/5 & 0 \\ 1/10 & 4/5 & 1/10 \\ 0 & 1/5 & 4/5 \end{pmatrix}. \tag{11.91}$$



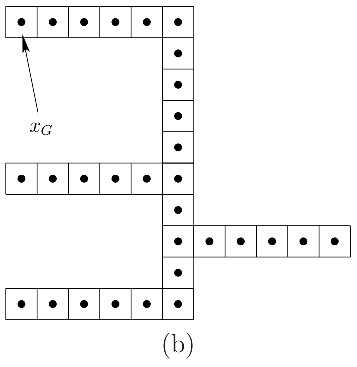


Figure 11.33: (a) A topological graph in which a point moves (note that two vertices are vertically aligned). (b) An exercise that is a variant of Example 11.17.

The jth entry of the ith row yields $P(x_{k+1} = i \mid x_k = j, u_k = 0)$. For $u = 1$, let $P(x_{k+1} \mid x_k, u_k = 1)$ be

$$\begin{pmatrix} 1/10 & 5/5 & 1/10 \\ 0 & 1/5 & 4/5 \\ 0 & 0 & 1 \end{pmatrix}. \tag{11.92}$$

The sensing model is specified by three vectors:

$$P(y_k|x_k = 0) = [4/5 \;\; 1/5], \tag{11.93}$$

$$P(y_k|x_k = 1) = [1/2 \;\; 1/2], \tag{11.94}$$

and

$$P(y_k|x_k = 2) = [1/5 \;\; 4/5], \tag{11.95}$$

in which the ith component yields $P(y_k = i \mid x_k)$. Suppose that $k = 3$ and the history I-state obtained so far is

$$(\eta_0, u_1, u_2, y_1, y_2, y_3) = (\eta_0, 1, 0, 1, 0, 0). \tag{11.96}$$

The task is to compute the probabilistic I-state. Starting from $P(x_1)$, compute the following distributions: $P(x_1|\eta_1)$, $P(x_2|\eta_1, u_1)$, $P(x_2|\eta_2)$, $P(x_3|\eta_2, u_2)$, $P(x_3|\eta_3)$.

6. Explain why it is not possible to reach every nondeterministic I-state from every other one for Example 11.7. Give an example of a nondeterministic I-state that cannot be reached from the initial I-state. Completely characterize the reachability of nondeterministic I-states from all possible initial conditions.

7. In the same spirit as Example 11.21, consider a point moving on the topological graph shown in Figure 11.33. Fully characterize the connectivity of $\mathcal{I}_{ndet}$ (you may exploit symmetries to simplify the answer).

8. Design an I-map for Example 11.17 that is not necessarily sufficient but leads to a solution plan defined over only three derived I-states.

9. Consider the discrete problem in Figure 11.33b, using the same sensing and motion model as in Example 11.17.

   (a) Develop a sufficient I-map and a solution plan that uses as few derived I-states as possible.

   (b) Develop an I-map that is not necessarily sufficient, and a solution plan that uses as few derived I-states as possible.

10. Suppose that there are two I-maps, $\kappa_1 : \mathcal{I}_1 \rightarrow \mathcal{I}_2$ and $\kappa_2 : \mathcal{I}_2 \rightarrow \mathcal{I}_3$, and it is given that κ_1 is sufficient with respect to $\mathcal{I}_1$, and κ_2 is sufficient with respect to $\mathcal{I}_2$. Determine whether the I-map $\kappa_2 \circ \kappa_1$ is sufficient with respect to $\mathcal{I}_1$, and prove your claim.

11. Propose a solution to Example 11.16 that uses fewer nondeterministic I-states.

12. Suppose that a point robot moves in $\mathbb{R}^2$ and receives observations from three homing beacons that are not collinear and originate from known locations. Assume that the robot can calibrate the three observations on $\mathbb{S}^1$.

    (a) Prove that the robot can always recover its position in $\mathbb{R}^2$.

    (b) What can the robot infer if there are only two beacons?

13. Nondeterministic I-state problems:

    (a) Prove that the nondeterministic I-states for Example 11.23 are always a single connected region whose boundary is composed only of circular arcs and line segments.

    (b) Design an algorithm for efficiently computing the nondeterministic I-states from stage to stage.

14. Design an algorithm that takes as input a simply connected rectilinear region (i.e., described by a polygon that has all right angles) and a goal state, and designs a sequence of tray tilts that guarantees the ball will come to rest at the goal. Example 11.24 provides an illustration.

15. Extend the game-theoretic formulation from Section 11.7.2 of history I-spaces to continuous time.

16. Consider the "where did I come from?" problem.

    (a) Derive an expression for $X_1(\eta_k)$.

    (b) Derive an expression for $P(x_1|\eta_k)$.

17. In the game of Example 11.27, could there exist a point in the game at which one player has not yet observed every possible "hit" yet it knows the state of the game (i.e., the exact location of all ships)? Explain.

18. When playing blackjack in casinos, many *card-counting* strategies involve remembering simple statistics of the cards, rather than the entire history of cards seen so far. Define a game of blackjack and card counting as an example of history I-states and an I-map that dramatically reduces the size of the I-space, and an information-feedback plan.

Implementations

19. Implement the Kalman filter for the case of a robot moving in the plane. Show the confidence ellipsoids obtained during execution. Be careful of numerical issues (see [151]).

20. Implement probabilistic I-state computations for a point robot moving in a 2D polygonal environment. Compare the efficiency and accuracy of grid-based approximations to particle filtering.

21. Design and implement an algorithm that uses nondeterministic I-states to play a good game of Battleship, as explained in Example 11.27.

Chapter 12

Planning Under Sensing Uncertainty

The main purpose of Chapter 11 was to introduce information space (I-space) concepts and to provide illustrative examples that aid in understanding. This chapter addresses planning under sensing uncertainty, which amounts to planning in an I-space. Section 12.1 covers general-purpose algorithms, for which it will quickly be discovered that only problems with very few states can be solved because of the explosive growth of the I-space. In Chapter 6, it was seen that general-purpose motion planning algorithms apply only to simple problems. Ways to avoid this were either to develop sampling-based techniques or to focus on a narrower class of problems. It is intriguing to apply sampling-based planning ideas to I-spaces, but as of yet this idea remains largely unexplored. Therefore, the majority of this chapter focuses on planning algorithms designed for narrower classes of problems. In each case, interesting algorithms have been developed that can solve problems that are much more complicated than what could be solved by the general-purpose algorithms. This is because they exploit some structure that is specific to the problem.

An important philosophy when dealing with an I-space is to develop an I-map that reduces its size and complexity as much as possible by obtaining a simpler derived I-space. Following this, it may be possible to design a special-purpose algorithm that efficiently solves the new problem by relying on the fact that the I-space does have the full generality. This idea will appear repeatedly throughout the chapter. The most common derived I-space is $\mathcal{I}_{ndet}$ from Section 11.2.2; $\mathcal{I}_{prob}$, from Section 11.2.3, will also arise.

After Section 12.1, the problems considered in the remainder of the chapter are inspired mainly by robotics applications. Section 12.2 addresses the localization problem, which means that a robot must use sensing information to determine its location. This is essentially a matter of maintaining derived I-states and computing plans that lead to the desired derived I-space. Section 12.3 generalizes localization to problems in which the robot does not even know its environment.

In this case, the state space and I-space take into account both the possible environments in which the robot might be and the possible locations of the robot within each environment. This section is fundamental to robotics because it is costly and difficult to build precise maps of a robot's environment. By careful consideration of the I-space, a complete representation may be safely avoided in many applications.

Section 12.4 covers a kind of pursuit-evasion game that can be considered as a formal version of the children's game of "hide and seek." The pursuer carries a lantern and must illuminate an unpredictable evader that moves with unbounded speed. The nondeterministic I-states for this problem characterize the set of possible evader locations. The problem is solved by performing a cell decomposition of $\mathcal{I}_{ndet}$ to obtain a finite, graph-search problem. The method is based on finding critical curves in the I-space, much like the critical-curve method in Section 6.3.4 for moving a line-segment robot.

Section 12.5 concludes the chapter with manipulation planning under imperfect state information. This differs from the manipulation planning considered in Section 7.3.2 because it was assumed there that the state is always known. Section 12.5.1 presents the preimage planning framework, which was introduced two decades ago to address manipulation planning problems that have bounded uncertainty models for the state transitions and the sensors. Many important I-space ideas and complexity results were obtained from this framework and the body of literature on which it was based; therefore, it will be covered here. Section 12.5.2 addresses problems in which the robots have very limited sensing information and rely on the information gained from the physical interaction of objects. In some cases, these methods surprisingly do not even require sensing.

12.1 General Methods

This section presents planning methods for the problems introduced in Section 11.1. They are based mainly on general-purpose dynamic programming, without exploiting any particular structure to the problem. Therefore, their application is limited to small state spaces; nevertheless, they are worth covering because of their extreme generality. The basic idea is to use either the nondeterministic or probabilistic I-map to express the problem entirely in terms of the derived I-space, $\mathcal{I}_{ndet}$ or $\mathcal{I}_{prob}$, respectively. Once the derived information transition equation (recall Section 11.2.1) is defined, it can be imagined that $\mathcal{I}_{ndet}$ or $\mathcal{I}_{prob}$ is a state space in which perfect state measurements are obtained during execution (because the I-state is always known).

12.1.1 The Information Space as a Big State Space

Recall that any problem specified using Formulation 11.1 can be converted using derived I-states into a problem under Formulation 10.1. By building on the

Item	Notation	Explanation
State	$\vec{x} = \eta_{der}$	Derived I-state
State space	$\vec{X} = \mathcal{I}_{der}$	Derived I-space
Action space	$\vec{U} = U$	Original action space
Nature action space	$\vec{\Theta} \subseteq Y$	Original observation space
State transition equation	$\vec{f}(\vec{x}, \vec{u}, \vec{\theta})$	Nature action is just y
Initial state	$\vec{x}_I = \eta_0$	Initial I-state, $\eta_0 \in \mathcal{I}_{der}$
Goal set	$\vec{X}_G$	Subsets of original X_G
Cost functional	$\vec{L}$	Derived from original L

Figure 12.1: The derived I-space can be treated as an ordinary state space on which planning with perfect state information can be performed.

discussion from the end of Section 11.1.3, this can be achieved by treating the I-space as a big state space in which each state is an I-state in the original problem formulation. Some of the components were given previously, but here a complete formulation is given.

Suppose that a problem has been specified using Formulation 11.1, resulting in the usual components: X, U, Θ, f, Y, h, x_I, X_G, and L. The following concepts will work for any sufficient I-map; however, the presentation will be limited to two important cases: κ_{ndet} and κ_{prob}, which yield derived I-spaces $\mathcal{I}_{ndet}$ and $\mathcal{I}_{prob}$, respectively (recall Sections 11.2.2 and 11.2.3).

The components of Formulation 10.1 will now be specified using components of the original problem. To avoid confusion between the two formulations, an arrow will be placed above all components of the new formulation. Figure 12.1 summarizes the coming definitions. The new state space, $\vec{X}$, is defined as $\vec{X} = \mathcal{I}_{der}$, and a state, $\vec{x} \in \vec{X}$, is a derived I-state, $\vec{x} = \eta_{der}$. Under nondeterministic uncertainty, $\vec{x}_k$ means $X_k(\eta_k)$, in which η_k is the history I-state. Under probabilistic uncertainty, $\vec{x}_k$ means $P(x_k|\eta_k)$. The action space remains the same: $\vec{U} = U$.

The strangest part of the formulation is the new nature action space, $\vec{\Theta}(\vec{x}, \vec{u})$. The observations in Formulation 11.1 behave very much like nature actions because they are not selected by the robot, and, as will be seen shortly, they are the only unpredictable part of the new state transition equation. Therefore, $\vec{\Theta}(\vec{x}, \vec{u}) \subseteq Y$, the original observation space. A new nature action, $\vec{\theta} \in \vec{\Theta}$, is just an observation, $\vec{\theta}(\vec{x}, \vec{u}) = y$. The set $\vec{\Theta}(\vec{x}, \vec{u})$ generally depends on $\vec{x}$ and $\vec{u}$ because some observations may be impossible to receive from some states. For example, if a sensor that measures a mobile robot position is never wrong by more than 1 meter, then observations that are further than 1 meter from the true robot position are impossible.

A derived state transition equation is defined with $\vec{f}(\vec{x}_k, \vec{u}_k, \vec{\theta}_k)$ and yields a new state, $\vec{x}_{k+1}$. Using the original notation, this is just a function that uses $\kappa(\eta_k)$, u_k, and y_k to compute the next derived I-state, $\kappa(\eta_{k+1})$, which is allowed because we are working with sufficient I-maps, as described in Section 11.2.1.

Initial states and goal sets are optional and can be easily formulated in the new representation. The initial I-state, η_0, becomes the new initial state, $\vec{x}_I = \eta_0$. It is assumed that η_0 is either a subset of X or a probability distribution, depending on whether planning occurs in $\mathcal{I}_{ndet}$ or $\mathcal{I}_{prob}$. In the nondeterministic case, the new goal set $\vec{X}_G$ can be derived as

$$\vec{X}_G = \{X(\eta) \in \mathcal{I}_{ndet} \mid X(\eta) \subseteq X_G\}, \tag{12.1}$$

which is the set of derived I-states for which it is *guaranteed* that the true state lies in X_G. A probabilistic version can be made by requiring that all states assigned nonzero probability by $P(x|\eta)$ lie in X_G. Instead of being nonzero, a threshold could be used. For example, the goal may require being only 98% certain that the goal is reached.

The only remaining portion of Formulation 10.1 is the cost functional. We will develop a cost model that uses only the state and action histories. A dependency on nature would imply that the costs depend directly on the observation, $y = \vec{\theta}$, which was not assumed in Formulation 11.1. The general K-stage cost functional from Formulation 10.1 appears in this context as

$$\vec{L}(\vec{x}_k, \vec{u}_k) = \sum_{k=1}^{K} \vec{l}(\vec{x}_k, \vec{u}_k) + \vec{l}_F(\vec{x}_F), \tag{12.2}$$

with the usual cost assumptions regarding the termination action.

The cost functional $\vec{L}$ must be derived from the cost functional L of the original problem. This is expressed in terms of states, which are unknown. First consider the case of $\mathcal{I}_{prob}$. The state x_k at stage k follows the probability distribution $P(x_k|\eta_k)$, as derived in Section 11.2.3. Using $\vec{x}_k$, an expected cost is assigned as

$$\vec{l}(\vec{x}_k, \vec{u}_k) = \vec{l}(\eta_k, u_k) = \sum_{x_k \in X} P(x_k|\eta_k) l(x_k, u_k) \tag{12.3}$$

and

$$\vec{l}_F(\vec{x}_F) = \vec{l}_F(\eta_F) = \sum_{x_F \in X} P(x_F|\eta_K) l_F(x_F). \tag{12.4}$$

Ideally, we would like to make analogous expressions for the case of $\mathcal{I}_{ndet}$; however, there is one problem. Formulating the worst-case cost for each stage is too pessimistic. For example, it may be possible to obtain high costs in two consecutive stages, but each of these may correspond to following different paths in X. There is nothing to constrain the worst-case analysis to the *same* path. In the probabilistic case there is no problem because probabilities can be assigned to paths. For the nondeterministic case, a cost functional can be defined, but the stage-additive property needed for dynamic programming is destroyed in general. Under some restrictions on allowable costs, the stage-additive property is preserved.

The state x_k at stage k is known to lie in $X_k(\eta_k)$, as derived in Section 11.2.2. For every history I-state, $\eta_k = \vec{x}_k$, and $u_k \in U$, assume that $l(x_k, u_k)$ is invariant over all $x_k \in X_k(\eta_k)$. In this case,

$$\vec{l}(\vec{x}_k, \vec{u}_k) = \vec{l}(\eta_k, u_k) = l(x_k, u_k), \tag{12.5}$$

in which $x_k \in X_k(\eta_k)$, and

$$\vec{l}_F(\vec{x}_F) = \vec{l}_F(\eta_F) = l_F(x_F), \tag{12.6}$$

in which $x_F \in X_F(\eta_F)$.

A plan on the derived I-space, $\mathcal{I}_{ndet}$ or $\mathcal{I}_{prob}$, can now also be considered as a plan on the new state space $\vec{X}$. Thus, state feedback is now possible, but in a larger state space $\vec{X}$ instead of X. The outcomes of actions are still generally unpredictable due to the observations. An interesting special case occurs when there are no observations. In this case, the I-state is predictable because it is derived only from actions that are chosen by the robot. In this case, the new formulation does not need nature actions, which reduces it down to Formulation 2.3. Due to this, feedback is no longer needed if the initial I-state is given. A plan can be expressed once again as a sequence of actions. Even though the *original* states are not predictable, the future *information* states are! This means that the state trajectory in the new state space is completely predictable as well.

12.1.2 Algorithms for Nondeterministic I-Spaces

Now that the problem of planning in $\mathcal{I}_{ndet}$ has been expressed using Formulation 10.1, the methods of Section 10.2 directly apply. The main limitation of their use is that the new state space $\vec{X}$ is exponentially larger than X. If X contains n states, then $\vec{X}$ contains $2^n - 1$ states. Thus, even though some methods in Section 10.2 can solve problems in practice that involve a million states, this would only be about 20 states in the original state space. Handling substantially larger problems requires developing application-specific methods that exploit some special structure of the I-space, possibly by defining an I-map that leads to a smaller derived I-space.

Value iteration The value-iteration method from Section 10.2.1 can be applied without modification. In the first step, initialize G^*_F using (12.6). Using the notation for the new problem, the dynamic programming recurrence, (10.39), becomes

$$G^*_k(\vec{x}_k) = \min_{\vec{u}_k \in U} \Big\{ \max_{\vec{\theta}_k} \Big\{ \vec{l}(\vec{x}_k, \vec{u}_k) + G^*_{k+1}(\vec{x}_{k+1}) \Big\} \Big\}, \tag{12.7}$$

in which $\vec{x}_{k+1} = \vec{f}(\vec{x}_k, \vec{u}_k, \vec{\theta}_k)$.

The main difficulty in evaluating (12.7) is to determine the set $\vec{\Theta}(\vec{x}_k, \vec{u}_k)$, over which the maximization occurs. Suppose that a state-nature sensor mapping is used, as defined in Section 11.1.1. From the I-state $\vec{x}_k = X_k(\eta_k)$, the action $\vec{u}_k = u_k$ is applied. This yields a forward projection $X_{k+1}(\eta_k, u_k)$. The set of all possible observations is

$$\vec{\Theta}(\vec{x}_k, \vec{u}_k) = \{ y_{k+1} \in Y \mid \exists x_{k+1} \in X_{k+1}(\eta_k, u_k) \text{ and } \exists \psi_{k+1} \in \Psi \text{ such that } y_{k+1} = h(x_{k+1}, \psi_{k+1}) \}. \tag{12.8}$$

Without using forward projections, a longer, equivalent expression is obtained:

$$\vec{\Theta}(\vec{x}_k, \vec{u}_k) = \{ y_{k+1} \in Y \mid \exists x_k \in X_k(\eta_k), \exists \theta_k \in \Theta, \text{ and } \exists \psi_{k+1} \in \Psi \text{ such that } y_{k+1} = h(f(x_k, u_k, \theta_k), \psi_{k+1}) \}. \tag{12.9}$$

Other variants can be formulated for different sensing models.

Policy iteration The policy iteration method of Section 10.2.2 can be applied in principle, but it is unlikely to solve challenging problems. For example, if $|X| = 10$, then each iteration will require solving matrices that have 1 million entries! At least they are likely to be sparse in many applications.

Graph-search methods The methods from Section 10.2.3, which are based on backprojections, can also be applied to this formulation. These methods must initially set $S = \vec{X}_G$. If S is initially nonempty, then backprojections can be attempted using the general algorithm in Figure 10.6. Dijkstra's algorithm, as given in Figure 10.8, can be applied to yield a plan that is worst-case optimal.

The sensorless case If there are no sensors, then better methods can be applied because the formulation reduces from Formulation 10.1 to Formulation 2.3. The simpler value iterations of Section 2.3 or Dijkstra's algorithm can be applied to find a solution. If optimality is not required, then any of the search methods of Section 2.2 can even be applied. For example, one can even imagine performing a bidirectional search on $\vec{X}$ to attempt to connect $\vec{x}_I$ to some $\vec{x}_G$.

12.1.3 Algorithms for Probabilistic I-Spaces (POMDPs)

For the probabilistic case, the methods of Section 10.2 cannot be applied because $\mathcal{I}_{prob}$ is a continuous space. Dynamic programming methods for continuous state spaces, as covered in Section 10.6, are needed. The main difficulty is that the dimension of $\vec{X}$ grows linearly with the number of states in X. If there are n states in X, the dimension of $\vec{X}$ is $n-1$. Since the methods of Section 10.6 suffer from the curse of dimensionality, the general dynamic programming techniques are limited to problems in which X has only a few states.

Approximate value iteration The continuous-space methods from Section 10.6 can be directly applied to produce an approximate solution by interpolating over $\vec{X}$ to determine cost-to-go values. The initial cost-to-go value G^*_F over the collection of samples is obtained by (12.6). Following (10.46), the dynamic programming recurrence is

$$G^*_k(\vec{x}_k) = \min_{\vec{u}_k \in \vec{U}} \left\{ \vec{l}(\vec{x}_k, \vec{u}_k) + \sum_{\vec{x}_{k+1} \in \vec{X}} G^*_{k+1}(\vec{x}_{k+1}) P(\vec{x}_{k+1} | \vec{x}_k, \vec{u}_k) \right\}. \tag{12.10}$$

If $\vec{\Theta}(\vec{x}, \vec{u})$ is finite, the probability mass is distributed over a finite set of points, $y = \vec{\theta} \in \vec{\Theta}(\vec{x}, \vec{u})$. This in turn implies that $P(\vec{x}_{k+1}|\vec{x}_k, \vec{u}_k)$ is also distributed over a finite subset of $\vec{X}$. This is somewhat unusual because $\vec{X}$ is a continuous space, which ordinarily requires the specification of a probability density function. Since the set of future states is finite, this enables a sum to be used in (12.10) as opposed to an integral over a probability density function. This technically yields a probability *density* over $\vec{X}$, but this density must be expressed using Dirac functions.[1] An approximation is still needed, however, because the x_{k+1} points may not be exactly the sample points on which the cost-to-go function G^*_{k+1} is represented.

Exact methods If the total number of stages is small, it is possible in practice to compute exact representations. Some methods are based on an observation that the cost-to-come is piecewise linear and convex [136]. A linear-programming problem results, which can be solved using the techniques that were described for finding randomized saddle points of zero-sum games in Section 9.3. Due to the numerous constraints, methods have been proposed that dramatically reduce the number that need to be considered in some circumstances (see the suggested reading on POMDPs at the end of the chapter).

An exact, discrete representation can be computed as follows. Suppose that the initial condition space $\mathcal{I}_0$ consists of one initial condition, η_0 (or a finite number of initial conditions), and that there are no more than K stages at which decisions are made. Since $\Theta(x, u)$ and $\Psi(x)$ are assumed to be finite, there is a finite number of possible final I-states, $\eta_F = (\eta_0, \tilde{u}_K, \tilde{y}_F)$. For each of these, the distribution $P(x_F|\eta_F)$ can be computed, which is alternatively represented as $\vec{x}_F$. Following this, (12.4) is used to compute $G^*(\vec{x}_F)$ for each possible $\vec{x}_F$. The number of these states is unfortunately exponential in the total number of stages, but at least there are finitely many. The dynamic programming recurrence (12.10) can be applied for $k = K$ to roll back one stage. It is known that each possible $\vec{x}_{k+1}$ will be a point in $\vec{X}$ at which a value was computed because values were computed for possible all I-states. Therefore, interpolation is not necessary. Equation 12.10 can be applied repeatedly until the first stage is reached. In each iteration, no interpolation is needed because the cost-to-go G^*_{k+1} was computed for each possible next I-state. Given the enormous size of $\mathcal{I}$, this method is practical only for very small problems.

[1]These are single points that are assigned a nonzero probability mass, which is not allowed, for example, in the construction of a continuous probability density function.

The sensorless case In the case of having no observations, the path through $\mathcal{I}_{prob}$ becomes predictable. Suppose that a feasible planning problem is formulated. For example, there are complicated constraints on the probability distributions over X that are permitted during the execution of the plan. Since $\vec{X} = \mathcal{I}_{prob}$ is a continuous space, it is tempting to apply motion planning techniques from Chapter 5 to find a successful path. The adaptation of such techniques may be possible, but they must be formulated to use actions and state transition functions, which was not done in Chapter 5. Such adaptations of these methods, however, will be covered in Chapter 14. They could be applied to this problem to search the I-space and produce a sequence of actions that traverses it while satisfying hard constraints on the probabilities.

12.2 Localization

Localization is a fundamental problem in robotics. Using its sensors, a mobile robot must determine its location within some map of the environment. There are both passive and active versions of the localization problem:

> **Passive localization:** The robot applies actions, and its position is inferred by computing the nondeterministic or probabilistic I-state. For example, if the Kalman filter is used, then probabilistic I-states are captured by mean and covariance. The mean serves as an estimate of the robot position, and the covariance indicates the amount of uncertainty.
>
> **Active localization:** A plan must be designed that attempts to reduce the localization uncertainty as much as possible. How should the robot move so that it can figure out its location?

Both versions of localization will be considered in this section.

In many applications, localization is an incremental problem. The initial configuration may be known, and the task is to maintain good estimates as motions occur. A more extreme version is the *kidnapped-robot problem*, in which a robot initially has no knowledge of its initial configuration. Either case can be modeled by the appropriate initial conditions. The kidnapped-robot problem is more difficult and is assumed by default in this section.

12.2.1 Discrete Localization

Many interesting lessons about realistic localization problems can be learned by first studying a discrete version of localization. Problems that may or may not be

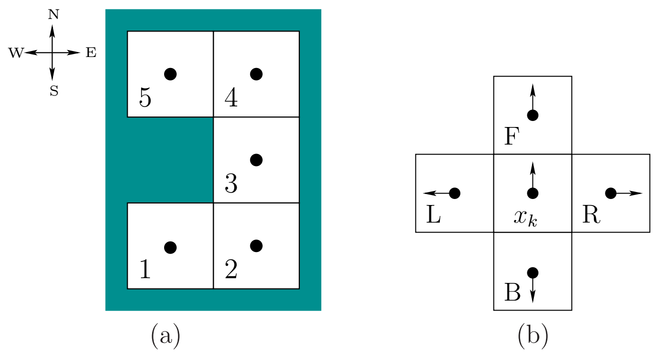


Figure 12.2: (a) This map is given to the robot for localization purposes. (b) The four possible actions each take one step, if possible, and reorient the robot as shown.

solvable can be embedded in more complicated problems, which may even involve continuous state spaces. The discrete case is often easier to understand, which motivates its presentation here. To simplify the presentation, only the nondeterministic I-space $\mathcal{I}_{ndet}$ will be considered; see Section 12.2.3 for the probabilistic case.

Suppose that a robot moves on a 2D grid, which was introduced in Example 2.1. It has a map of the grid but does not know its initial location or orientation within the grid. An example is shown in Figure 12.2a.

To formulate the problem, it is helpful to include in the state both the position of the robot and its orientation. Suppose that the robot may be oriented in one of four directions, which are labeled N, E, W, and S, for "north," "east," "west," and "south," respectively. Although the robot is treated as a point, its orientation is important because it does not have a compass. If it chooses to move in a particular direction, such as straight ahead, it does not necessarily know which direction it will be heading with respect to the four directions.

Thus, a state, $x \in X$, is written as $x = (p, d)$, in which p is a position and d is one of the four directions. A set of states at the same position will be denoted with special superscripts that point in the possible directions. For example, $3^{\llcorner}$ indicates the set of states for which $p = 3$ and the direction may be north (N) or east (E), because the superscript points in the north and east directions.

The robot is given four actions,

$$U = \{\mathrm{F}, \mathrm{B}, \mathrm{R}, \mathrm{L}\}, \tag{12.11}$$

which represent "forward," "backward," "right motion," and "left motion," respectively. These motions occur with respect to the current orientation of the robot, which may be unknown. See Figure 12.2b. For the F action, the robot moves forward one grid element and maintains its orientation. For the B action,

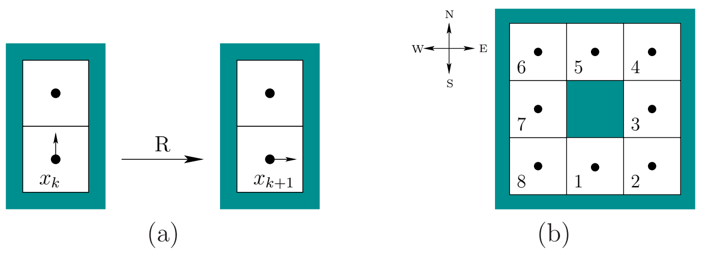


Figure 12.3: (a) If a direction is blocked because of an obstacle, then the orientation changes, but the position remains fixed. In this example, the R action is applied. (b) Another map is given to the robot for localization purposes. In this case, the robot cannot localize itself exactly.

the robot changes its orientation by 180 degrees and then moves forward one grid element. For the R action, the robot turns right by 90 degrees and then moves forward one grid element. The L action behaves similarly. If it is not possible to move because of an obstacle, it is assumed that the robot changes its orientation (in the case of B, R, or L) but does not change its position. This is depicted in Figure 12.3a.

The robot has one simple sensor that can only detect whether it was able to move in the direction that was attempted. The sensor space is $Y = \{0, 1\}$, and the sensor mapping is $h : X \times X \to Y$. This yields $y = h(x_{k-1}, x_k) = 1$ if x_{k-1} and x_k place the robot at different positions, and $h(x_{k-1}, x_k) = 0$ otherwise. Thus, the sensor indicates whether the robot has moved after the application of an action.

Nondeterministic uncertainty will be used, and the initial I-state η_0 is always assumed to be X (this can easily be extended to allow starting with any nonempty subset of X). A history I-state at stake k in its general form appears as

$$\eta_0 = (X, \tilde{u}_{k-1}, y_2, \ldots, y_k). \tag{12.12}$$

One special adjustment was made in comparison to (11.14). There is no observation y_1 because the sensor mapping requires a previous state to report a value. Thus, the observation history starts with y_2. An example history I-state for stage $k = 5$ is

$$\eta_5 = (X, \mathrm{R}, \mathrm{R}, \mathrm{F}, \mathrm{L}, 1, 0, 1, 1), \tag{12.13}$$

in which $\eta_0 = X$, $\tilde{u}_4 = (\mathrm{R}, \mathrm{R}, \mathrm{F}, \mathrm{L})$, and $(y_2, y_3, y_4, y_5) = (1, 0, 1, 1)$.

The *passive localization* problem starts with a given map, such as the one shown in Figure 12.2a, and a history I-state, η_k, and computes the nondeterministic I-state $X_k(\eta_k) \subseteq X$. The *active localization problem* is to compute some k and sequence of actions, $(u_1, \ldots, u_{k-1})$, such that the nondeterministic I-state is as small as possible. In the best case, $X_k(\eta_k)$ might become a singleton set, which means that the robot knows its position and orientation on the map. However, due to *symmetries*, which will be presented shortly in an example, it might not be possible.

Solving the passive localization problem The passive problem requires only that the nondeterministic I-states are computed correctly as the robot moves. A couple of examples of this are given.

Example 12.1 (An Easy Localization Problem) Consider the example given in Figure 12.2a. Suppose that the robot is initially placed in position 1 facing east. The initial condition is $\eta_0 = X$, which can be represented as

$$\eta_0 = 1^{+} \cup 2^{+} \cup 3^{+} \cup 4^{+} \cup 5^{+}, \tag{12.14}$$

the collection of all 20 states in X. Suppose that the action sequence $(\mathrm{F}, \mathrm{L}, \mathrm{F}, \mathrm{L})$ is applied. In each case, a motion occurs, which results in the observation history $(y_2, y_3, y_4, y_5) = (1, 1, 1, 1)$.

After the first action, $u_1 = \mathrm{F}$, the history I-state is $\eta_2 = (X, \mathrm{F}, 1)$. The nondeterministic I-state is

$$X_2(\eta_2) = 1^{\leftarrow} \cup 2^{\rightarrow\downarrow} \cup 3^{\updownarrow} \cup 4^{\uparrow\rightarrow} \cup 5^{\leftarrow}, \tag{12.15}$$

which means that any position is still possible, but the successful forward motion removed some orientations from consideration. For example, $1^{\uparrow}$ is not possible because the previous state would have to be directly south of 1, which is an obstacle.

After the second action, $u_2 = \mathrm{L}$,

$$X_3(\eta_3) = 3^{\uparrow} \cup 5^{\leftarrow}, \tag{12.16}$$

which yields only two possible current states. This can be easily seen in Figure 12.2a by observing that there are only two states from which a forward motion can be followed by a left motion. The initial state must have been either $1^{\leftarrow}$ or $3^{\uparrow}$.

After $u_3 = \mathrm{F}$ is applied, the only possibility remaining is that x_3 must have been $3^{\uparrow}$. This yields

$$X_4(\eta_4) = 4^{\uparrow}, \tag{12.17}$$

which exactly localizes the robot: It is at position 4 facing north. After the final action $u_4 = \mathrm{L}$ is applied it is clear that

$$X_5(\eta_5) = 5^{\leftarrow}, \tag{12.18}$$

which means that in the final state, x_5, the robot is at position 1 facing west. Once the exact robot state is known, no new uncertainty will accumulate because the effects of all actions are predictable. Although it was not shown, it is also possible to prune the possible states by the execution of actions that do not produce motions. ■

Example 12.2 (A Problem that Involves Symmetries) Now extend the map from Figure 12.2a so that it forms a loop as shown in Figure 12.2b. In this case, it is impossible to determine the precise location of the robot. For simplicity, consider only actions that produce motion (convince yourself that allowing the other actions cannot fix the problem).

Suppose that the robot is initially in position 1 facing east. If the action sequence $(\mathrm{F}, \mathrm{L}, \mathrm{F}, \mathrm{L}, \ldots)$ is executed, the robot will travel around in cycles. The problem is that it is also possible to apply the same action sequence from position 3 facing north. Every action successfully moves the robot, which means that, to the robot, the information appears identical. The other two cases in which this sequence can be applied to travel in cycles are 1) from 5 heading west, and 2) from 7 heading south. A similar situation occurs from 2 facing east, if the sequence $(\mathrm{L}, \mathrm{F}, \mathrm{L}, \mathrm{F}, \ldots)$ is applied. Can you find the other three starting states from which this sequence moves the robot at every stage? Similar symmetries exist when traveling in clockwise circles and making right turns instead of left turns.

The state space for this problem contains 32 states, obtained from four directions at each position. After executing some motions, the nondeterministic I-state can be reduced down to a *symmetry class* of no more than four possible states. How can this be proved? One way is to use the algorithm that is described next. ■

Solving the active localization problem From the previous two examples, it should be clear how to compute nondeterministic I-states and therefore solve the passive localization problem on a grid. Now consider constructing a plan that solves the active localization problem. Imagine using a computer to help in this task. There are two general approaches:

Precomputed Plan: In this approach, a planning algorithm running on a computer accepts a map of the environment and computes an information-feedback plan that immediately indicates which action to take based on all possible I-states that could result (a derived I-space could be used). During execution, the actions are immediately determined from the stored, precomputed plan.

Lazy Plan: In this case the map is still given, but the appropriate action is computed just as it is needed during each stage of execution. The computer runs on-board of the robot and must compute which action to take based on the current I-state.

The issues are similar to those of the sampling-based roadmap in Section 5.6. If faster execution is desired, then the precomputed plan may be preferable. If it would consume too much time or space, then a lazy plan may be preferable.

Using either approach, it will be helpful to recall the formulation of Section 12.1.1, which considers $\mathcal{I}_{ndet}$ as a new state space, $\vec{X}$, in which state feedback can be used. Even though there are no nature sensing actions, the observations are not predictable because the state is generally unknown. This means that $\vec{\theta}$ is unknown, and future new states, $\vec{x}_{k+1}$, are unpredictable once $\vec{x}_k$ and $\vec{u}_k$ are given. A plan must therefore use feedback, which means that it needs information learned during execution to solve the problem. The state transition function $\vec{f}$ on the new state space was illustrated for the localization problem in Examples 12.1 and 12.2. The initial state $\vec{x}_I$ is the set of all original states. If there are no symmetries, the goal set $\vec{X}_G$ is the set of all singleton subsets of X; otherwise, it is the set of all smallest possible I-states that are reachable (this does not need to be constructed in advance). If desired, cost terms can be defined to produce an optimal planning problem. For example, $\vec{l}(\vec{x},\vec{u}) = 2$ if a motion occurs, or $\vec{l}(\vec{x},\vec{u}) = 1$ otherwise.

Consider the approach of precomputing a plan. The methods of Section 12.1.2 can generally be applied to compute a plan, $\pi : \vec{X} \rightarrow U$, that solves the localization problem from any initial nondeterministic I-state. The approach may be space-intensive because an action must be stored for every state in $\vec{X}$. If there are n grid tiles, then $|\vec{X}| = 2^n - 1$. If the initial I-state is always X, then it may be possible to restrict π to a much smaller portion of $\vec{X}$. From any $\vec{x} \in \vec{X}_G$, a search based on backprojections can be conducted. If the initial I-state is added to S, then the partial plan will reliably localize the robot. Parts of $\vec{X}$ for which π is not specified will never be reached and can therefore be ignored.

Now consider the lazy approach. An algorithm running on the robot can perform a kind of search by executing actions and seeing which I-states result. This leads to a directed graph over $\vec{X}$ that is incrementally revealed through the robot's motions. The graph is directed because the information regarding the state generally improves. For example, once the robot knows its state (or symmetry class of states), it cannot return to an I-state that represents greater uncertainty. In many cases, the robot may get lucky during execution and localize itself using much less memory than would be required for a precomputed plan.

The robot needs to recognize that the same positions have been reached in different ways, to ensure a systematic search. Even though the robot does not necessarily know its position on the map, it can usually deduce whether it has been to some location previously. One way to achieve this is to assign (i,j) coordinates to the positions already visited. It starts with $(0,0)$ assigned to the initial position. If F is applied, then suppose that position $(1,0)$ is reached, assuming the robot moves to a new grid cell. If R is applied, then $(0,1)$ is reached if the robot is not blocked. The point $(2,1)$ may be reachable by $(\mathrm{F},\mathrm{F},\mathrm{R})$ or $(\mathrm{R},\mathrm{F},\mathrm{F})$. One way to interpret this is that a local coordinate frame in $\mathbb{R}^2$ is attached to the robot's initial position. Let this be referred to as the *odometric coordinates*. The orientation between this coordinate frame and the map is not known in the beginning, but a transformation between the two can be computed if the robot is able to localize itself exactly.

A variety of search algorithms can now be defined by starting in the initial state $\vec{x}_I$ and trying actions until a goal condition is satisfied (e.g., no smaller nondeterministic I-states are reachable). There is, however, a key difference between this search and the search conducted by the algorithms in Section 2.2.1. Previously, the search could continue from any state that has been explored previously without any additional cost. In the current setting, there are two issues:

Reroute paths: Most search algorithms enable new states to be expanded from any previously considered states at any time. For the lazy approach, the robot must move to a state and apply an action to determine whether a new state can be reached. The robot is capable of returning to any previously considered state by using its odometric coordinates. This induces a cost that does not exist in the previous search problem. Rather than being able to jump from place to place in a search tree, the search is instead a long, continuous path that is traversed by the robot. Let the jump be referred to as a *reroute path*. This will become important in Section 12.3.2.

Information improvement: The robot may not even be able to return to a previous nondeterministic I-state. For example, if the robot follows $(\mathrm{F},\mathrm{F},\mathrm{R})$ and then tries to return to the same state using $(\mathrm{B},\mathrm{L},\mathrm{F})$, it will indeed know that it returned to the same state, but the state remains unknown. It might be the case, however, that after executing $(\mathrm{F},\mathrm{F},\mathrm{R})$, it was able to narrow down the possibilities for its current state. Upon returning using $(\mathrm{B},\mathrm{L},\mathrm{F})$, the nondeterministic I-state will be different.

The implication of these issues is that the search algorithm should take into account the cost of moving the robot and that the search graph is directed. The second issue is really not a problem because even though the I-state may be different when returning to the same position, it will always be at least as good as the previous one. This means that if η_1 and η_2 are the original and later history I-states from the same position, it will always be true that $X(\eta_2) \subseteq X(\eta_1)$. Information always improves in this version of the localization problem. Thus, while trying to return to a previous I-state, the robot will find an improved I-state.

Other information models The model given so far in this section is only one of many interesting alternatives. Suppose, for example, that the robot carries a compass that always indicates its direction. In this case, there is no need to keep track of the direction as part of the state. The robot can use the compass to specify actions directly with respect to global directions. Suppose that $U = \{\mathrm{N},\mathrm{E},\mathrm{W},\mathrm{S}\}$, which denote the directions, "north," "east," "west," and "south," respectively. Examples 12.1 and 12.2 now become trivial. The first one is solved by applying the action sequence (E,N). The symmetry problems vanish for Example 12.2, which can also be solved by the sequence (E,N) because $(1,2,3)$ is the only sequence of positions that is consistent with the actions and compass readings.

Other interesting models can be made by giving the robot less information. In the models so far, the robot can easily infer its current position relative to its starting position. Even though it is not necessarily known where this starting position lies on the map, it can always be expressed in relative coordinates. This is because the robot relies on different forms of odometry. For example, if the direction is E and the robot executes the sequence (L, L, L), it is known that the direction is S because three lefts make a right. Suppose that instead of a grid, the robot must explore a graph. It moves discretely from vertex to vertex by applying an action that traverses an edge. Let this be a planar graph that is embedded in $\mathbb{R}^2$ and is drawn with straight line segments. The number of available actions can vary at each vertex. We can generally define $U = \mathbb{S}^1$, with the behavior that the robot only rotates without translating whenever a particular direction is blocked (this is a generalization of the grid case). A sensor can be defined that indicates which actions will lead to translations from the current vertex. In this case, the model nicely generalizes the original model for the grid. If the robot knows the angles between the edges that arrive at a vertex, then it can use angular odometry to make a local coordinate system in $\mathbb{R}^2$ that keeps track of its relative positions.

The situation can be made very confusing for the robot. Suppose that instead of $U = \mathbb{S}^1$, the action set at each vertex indicates which edges can be traversed. The robot can traverse an edge by applying an action, but it does not know anything about the direction relative to other edges. In this case, angular odometry can no longer be used. It could not, for example, tell the difference between traversing a rhombus, trapezoid, or a rectangle. If angular odometry is possible, then some symmetries can be avoided by noting the angles between the edges at each vertex. However, the new model does not allow this. All vertices that have the same degree would appear identical.

12.2.2 Combinatorial Methods for Continuous Localization

Now consider localization for the case in which X is a continuous region in $\mathbb{R}^2$. Assume that X is bounded by a simple polygon (a closed polygonal chain; there are no interior holes). A map of X in $\mathbb{R}^2$ is given to the robot. The robot velocity $\dot{x}$ is directly commanded by the action u, yielding a motion model $\dot{x} = u$, for which U is a unit ball centered at the origin. This enables a plan to be specified as a continuous path in X, as was done throughout Part II. Therefore, instead of specifying velocities using u, a path is directly specified, which is simpler. For models of the form $\dot{x} = u$ and the more general form $\dot{x} = f(x, u)$, see Section 8.4 and Chapter 13, respectively.

The robot uses two different sensors:

1. **Compass:** A perfect compass solves all orientation problems that arose in Section 12.2.1.

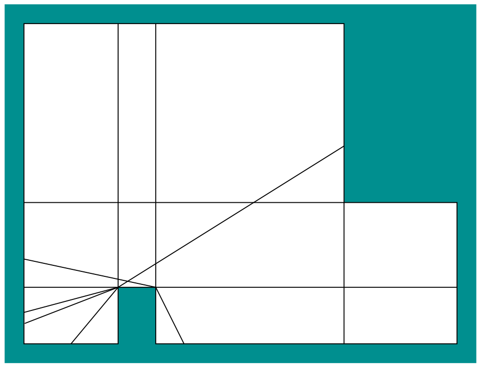


Figure 12.4: An example of the visibility cell decomposition. Inside of each cell, the visibility polygon is composed of the same edges of ∂X.

2. **Visibility:** The visibility sensor, which was shown in Figure 11.15, provides perfect distance measurements in all directions.

There are no nature sensing actions for either sensor.

As in Section 12.2.1, localization involves computing nondeterministic I-states. In the current setting there is no need to represent the orientation as part of the state space because of the perfect compass and known orientation of the polygon in $\mathbb{R}^2$. Therefore, the nondeterministic I-states are just subsets of X. Imagine computing the nondeterministic I-state for the example shown in Figure 11.15, but without any history. This is $H(y) \subseteq X$, which was defined in (11.6). Only the current sensor reading is given. This requires computing states from which the distance measurements shown in Figure 11.15b could be obtained. This means that a translation must be found that perfectly overlays the edges shown in Figure 11.15b on top of the polygon edges that are shown in Figure 11.15a. Let ∂X denote the boundary of X. The distance measurements from the visibility sensor must correspond exactly to a subset of ∂X. For the example, these could only be obtained from one state, which is shown in Figure 11.15a. Therefore, the robot does not even have to move to localize itself for this example.

As in Section 8.4.3, let the *visibility polygon* $V(x)$ refer to the set of all points visible from x, which is shown in Figure 11.15a. To perform the required computations efficiently, the polygon must be processed to determine the different ways in which the visibility polygon could appear from various points in X. This involves carefully determining which edges of ∂X could appear on $\partial V(x)$. The state space X can be decomposed into a finite number of cells, and over each region the invariant is that same set of edges is used to describe $V(x)$ [38, 120]. An example is shown in Figure 12.4. Two different kinds of rays must be extended to make the decomposition. Figure 12.5 shows the case in which a pair of vertices is mutually visible and an outward ray extension is possible. The other case is

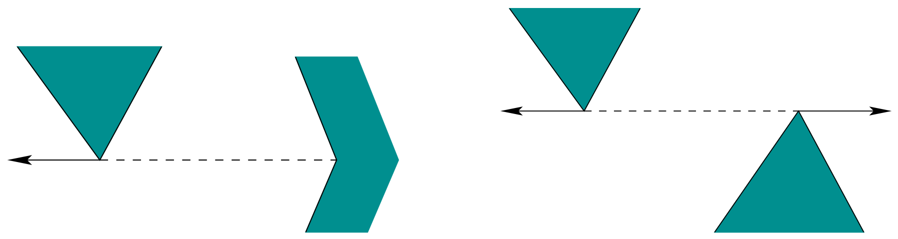


Figure 12.5: Rays are extended outward, whenever possible, from each pair of mutually visible vertices. The case on the right is a bitangent, as shown in Figure 6.10; however, here the edges extend outward instead of inward as required for the visibility graph.

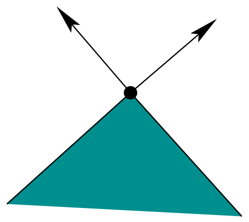


Figure 12.6: A reflex vertex: If the interior angle at a vertex is greater than π, then two outward rays are extended from the incident edges.

shown in Figure 12.6, in which rays are extended outward at every reflex vertex (a vertex whose interior angle is more than π, as considered in Section 6.2.4). The resulting decomposition generates $O(n^2 r)$ cells in the worse case, in which n is the number of edges that form ∂X and r is the number of reflex vertices (note that $r < n$). Once the measurements are obtained from the sensor, the cell or cells in which the edges or distance measurements match perfectly need to be computed to determine $H(y)$ (the set of points in X from which the current distance measurements could be obtained). An algorithm based on the idea of a *visibility skeleton* is given in [120], which performs these computations in time $O(m + \lg n + s)$ and uses $O(n^5)$ space, in which n is the number of vertices in ∂X, m is the number of vertices in $V(x)$, and $s = |H(y)|$, the size of the nondeterministic I-state. This method assumes that the environment is preprocessed to perform rapid queries during execution; without preprocessing, $H(y)$ can be computed in time $O(mn)$.

What happens if there are multiple states that match the distance data from the visibility sensor? Since the method in [120] only computes $H(y) \subseteq X$, some robot motions must be planned to further reduce the uncertainty. This provides yet another interesting illustration of the power of I-spaces. Even though the state space is continuous, an I-state in this case is used to disambiguate the state from a finite collection of possibilities.

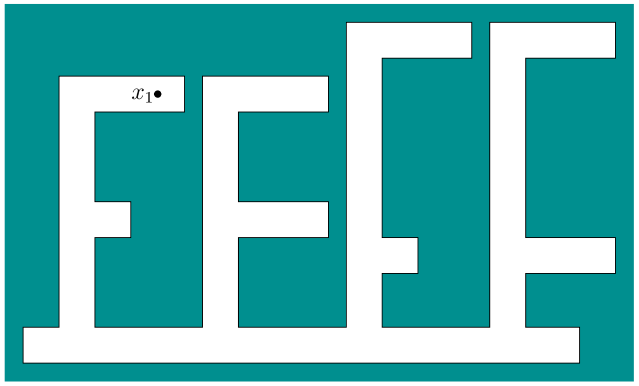


Figure 12.7: Consider this example, in which the initial state is not known [85].

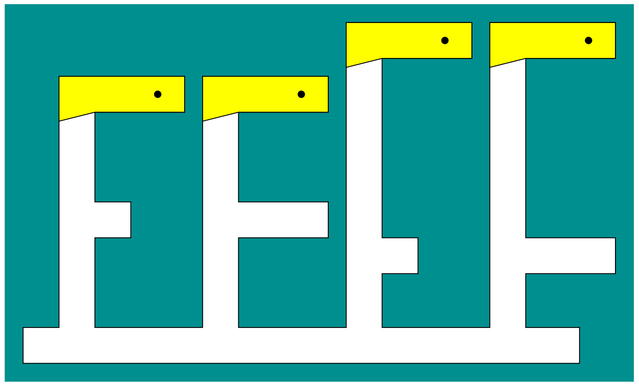


Figure 12.8: The four possible initial positions for the robot in Figure 12.7 based on the visibility sensor.

The following example is taken from [85].

Example 12.3 (Visibility-Based Localization) Consider the environment shown in Figure 12.7, with the initial state as shown. Based on the visibility sensor observation, the initial state could be any one of the four possibilities shown in Figure 12.8. Thus, $H(y_1)$ contains four states, in which y_1 is the initial sensor observation. Suppose that the motion sequence shown in Figure 12.9 is executed. After the first step, the position of the robot is narrowed down to two possibilities, as shown in Figure 12.10. This occurs because the corridor is longer for the remaining two possibilities. After the second motion, the state is completely determined because the short side corridor is detected. ■

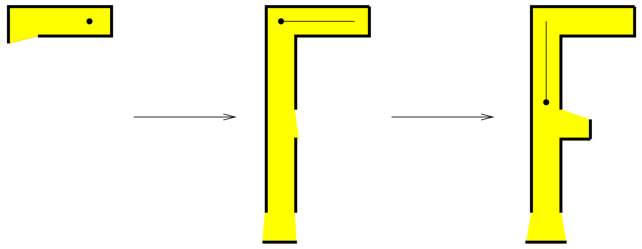


Figure 12.9: These motions completely disambiguate the state.

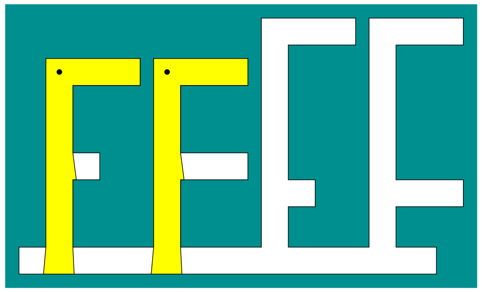


Figure 12.10: There are now only two possible states.

The localization problem can be solved in general by using the visibility cell decomposition, as shown in Figure 12.4. Initially, $X_1(\eta_1) = H(y_1)$ is computed from the initial visibility polygon, which can be efficiently performed using the visibility skeleton [120]. Suppose that $X_1(\eta_1)$ contains k states. In this case, k translated copies of the map are overlaid so that all of the possible states in $X_1(\eta_1)$ coincide. A motion is then executed that reduces the amount of uncertainty. This could be performed, by example, by crossing a cell boundary in the overlay that corresponds to one or more, but not all, of the k copies. This enables some possible states to be eliminated from the next I-state, $X_2(\eta_2)$. The overlay is used once again to obtain another disambiguating motion, which results in $X_3(\eta_3)$. This process continues until the state is known. In [85], a motion plan is given that enables the robot to localize itself by traveling no more than k times as far as the optimal distance that would need to be traveled to verify the given state. This particular localization problem might not seem too difficult after seeing Example 12.3, but it turns out that the problem of localizing using optimal motions is NP-hard if any simple polygon is allowed. This was proved in [85] by showing that every abstract decision tree can be realized as a localization problem, and the abstract decision tree problem is already known to be NP-hard.

Many interesting variations of the localization problem in continuous spaces can be constructed by changing the sensing model. For example, suppose that the robot can only measure distances up to a limit; all points beyond the limit cannot

be seen. This corresponds to many realistic sensing systems, such as infrared sensors, sonars, and range scanners on mobile robots. This may substantially enlarge $H(y)$. Suppose that the robot can take distance measurements only in a limited number of directions, as shown in Figure 11.14b. Another interesting variant can be made by removing the compass. This introduces the orientation confusion effects observed in Section 12.2.1. One can even consider interesting localization problems that have little or no sensing [205, 206], which yields I-spaces that are similar to that for the tray tilting example in Figure 11.28.

12.2.3 Probabilistic Methods for Localization

The localization problems considered so far have involved only nondeterministic uncertainty. Furthermore, it was assumed that nature does not interfere with the state transition equation or the sensor mapping. If nature is involved in the sensor mapping, then future I-states are not predictable. For the active localization problem, this implies that a localization plan must use information feedback. In other words, the actions must be conditioned on I-states so that the appropriate decisions are taken after new observations are made. The passive localization problem involves computing probabilistic I-states from the sensing and action histories. The formulation and solution of localization problems that involve nature and nondeterministic uncertainty will be left to the reader. Only the probabilistic case will be covered here.

Discrete problems First consider adding probabilities to the discrete grid problem of Section 12.2.1. A state is once again expressed as $x = (p, d)$. The initial condition is a probability distribution, $P(x_1)$, over X. One reasonable choice is to make $P(x_1)$ a uniform probability distribution, which makes each direction and position equally likely. The robot is once again given four actions, but now assume that nature interferes with state transitions. For example, if $u_k = F$, then perhaps with high probability the robot moves forward, but with low probability it may move right, left, or possibly not move at all, even if it is not blocked.

The sensor mapping from Section 12.2.1 indicated whether the robot moved. In the current setting, nature can interfere with this measurement. With low probability, it may incorrectly indicate that the robot moved, when in fact it remained stationary. Conversely, it may also indicate that the robot remained still, when in fact it moved. Since the sensor depends on the previous two states, the mapping is expressed as

$$y_k = h(x_k, x_{k-1}, \psi_k). \qquad (12.19)$$

With a given probability model, $P(\psi_k)$, this can be expressed as $P(y_k|x_k, x_{k-1})$.

To solve the passive localization problem, the expressions from Section 11.2.3 for computing the derived I-states are applied. If the sensor mapping used only the current state, then (11.36), (11.38), and (11.39) would apply without modification. However, since h depends on both x_k and x_{k-1}, some modifications are needed. Recall that the observations start with y_2 for this sensor. Therefore, $P(x_1|\eta_1) = P(x_1|y_1) = P(x_1)$, instead of applying (11.36).

After each stage, $P(x_{k+1}|\eta_{k+1})$ is computed from $P(x_k|\eta_k)$ by first applying (11.38) to take into account the action u_k. Equation (11.39) takes into account the sensor observation, y_{k+1}, but $P(y_{k+1}|x_{k+1}, \eta_k, u_k)$ is not given because the sensor mapping also depends on x_{k-1}. It reduces using marginalization as

$$P(y_k|\eta_{k-1}, u_{k-1}, x_k) = \sum_{x_{k-1} \in X} P(y_k|\eta_{k-1}, u_{k-1}, x_{k-1}, x_k) P(x_{k-1}|\eta_{k-1}, u_{k-1}, x_k). \tag{12.20}$$

The first factor in the sum can be reduced to the sensor model,

$$P(y_k|\eta_{k-1}, u_{k-1}, x_{k-1}, x_k) = P(y_k|x_{k-1}, x_k), \tag{12.21}$$

because the observations depend only on x_{k-1}, x_k, and the nature sensing action, ψ_k. The second term in (12.20) can be computed using Bayes' rule as

$$P(x_{k-1}|\eta_{k-1}, u_{k-1}, x_k) = \frac{P(x_k|\eta_{k-1}, u_{k-1}, x_{k-1}) P(x_{k-1}|\eta_{k-1}, u_{k-1})}{\displaystyle\sum_{x_{k-1} \in X} P(x_k|\eta_{k-1}, u_{k-1}, x_{k-1}) P(x_{k-1}|\eta_{k-1}, u_{k-1})}, \tag{12.22}$$

in which $P(x_k|\eta_{k-1}, u_{k-1}, x_{k-1})$ simplifies to $P(x_k|u_{k-1}, x_{k-1})$. This is directly obtained from the state transition probability, which is expressed as $P(x_{k+1}|x_k, u_k)$ by shifting the stage index forward. The term $P(x_{k-1}|\eta_{k-1}, u_{k-1})$ is given by (11.38). The completes the computation of the probabilistic I-states, which solves the passive localization problem.

Solving the active localization problem is substantially harder because a search occurs on $\mathcal{I}_{prob}$. The same choices exist as for the discrete localization problem. Computing an information-feedback plan over the whole I-space $\mathcal{I}_{prob}$ is theoretically possible but impractical for most environments. The search-based idea that was applied to incrementally grow a directed graph in Section 12.2.1 could also be applied here. The success of the method depends on clever search heuristics developed for this particular problem.

Continuous problems Localization in a continuous space using probabilistic models has received substantial attention in recent years [64, 124, 172, 235, 256, 288]. It is often difficult to localize mobile robots because of noisy sensor data, modeling errors, and high demands for robust operation over long time periods. Probabilistic modeling and the computation of probabilistic I-states have been quite successful in many experimental systems, both for indoor and outdoor mobile robots. Figure 12.11 shows localization successfully being solved using sonars only. The vast majority of work in this context involves passive localization because the

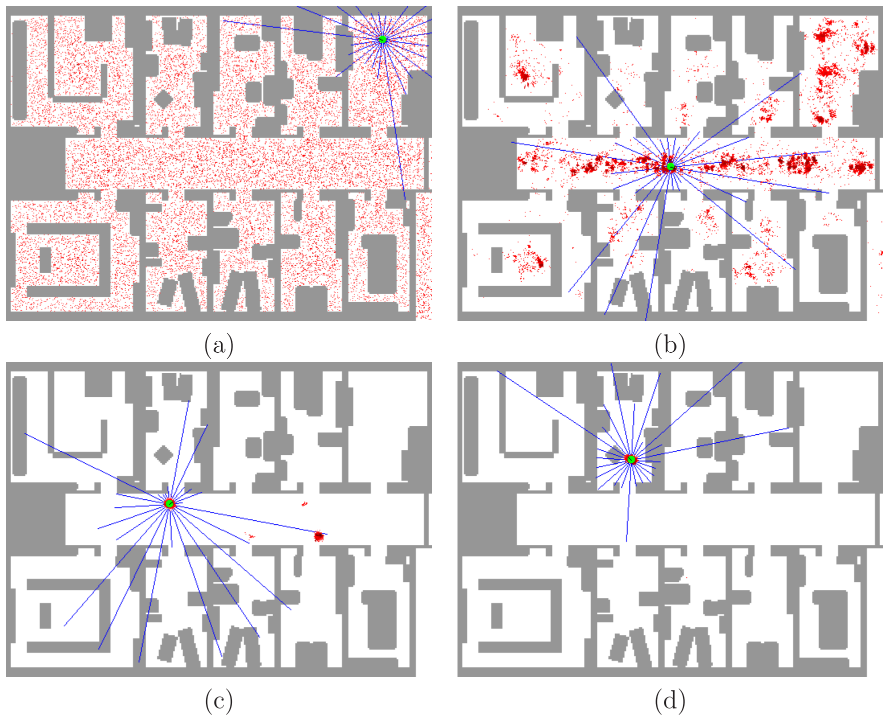


Figure 12.11: Four frames from an animation that performs probabilistic localization of an indoor mobile robot using sonars [105].

robot is often completing some other task, such as reaching a particular part of the environment. Therefore, the focus is mainly on *computing* the probabilistic I-states, rather than performing a difficult search on $\mathcal{I}_{prob}$.

Probabilistic localization in continuous spaces most often involves the definition of the probability densities $p(x_{k+1}|x_k, u_k)$ and $p(y_k|x_k)$ (in the case of a state sensor mapping). If the stages represent equally spaced times, then these densities usually remain fixed for every stage. The state space is usually $X = SE(2)$ to account for translation and rotation, but it may be $X = \mathbb{R}^2$ for translation only. The density $p(x_{k+1}|x_k, u_k)$ accounts for the unpredictability that arises when controlling a mobile robot over some fixed time interval. A method for estimating this distribution for nonholonomic robots by solving stochastic differential equations appears in [301].

The density $p(y_k|x_k)$ indicates the relative likelihood of various measurements when given the state. Most often this models distance measurements that are obtained from a laser range scanner, an array of sonars, or even infrared sensors. Suppose that a robot moves around in a 2D environment and takes depth measurements at various orientations. In the robot body frame, there are n angles at which a depth measurement is taken. Ideally, the measurements should look like those in Figure 11.15b; however, in practice, the data contain substantial noise. The observation $y \in Y$ is an n-dimensional vector of noisy depth measurements.

One common way to define $p(y|x)$ is to assume that the error in each distance measurement follows a Gaussian density. The mean value of the measurement can easily be calculated as the true distance value once x is given, and the variance should be determined from experimental evaluation of the sensor. If it is assumed that the vector of measurements is modeled as a set of independent, identically distributed random variables, a simple product of Guassian densities is obtained for $p(y|x)$.

Once the models have been formulated, the computation of probabilistic I-states directly follows from Sections 11.2.3 and 11.4.1. The initial condition is a probability density function, $p(x_1)$, over X. The marginalization and Bayesian update rules are then applied to construct a sequence of density functions of the form $p(x_k|\eta_k)$ for every stage, k.

In some limited applications, the models used to express $p(x_{k+1}|x_k, u_k)$ and $p(y_k|x_k)$ may be linear and Gaussian. In this case, the Kalman filter of Section 11.6.1 can be easily applied. In most cases, however, the densities will not have this form. Moment-based approximations, as discussed in Section 11.4.3, can be used to approximate the densities. If second-order moments are used, then the so-called *extended Kalman filter* is obtained, in which the Kalman filter update rules can be applied to a *linear-Gaussian* approximation to the original problem. In recent years, one of the most widely accepted approaches in experimental mobile robotics is to use sampling-based techniques to directly compute and estimate the probabilistic I-states. The particle-filtering approach, described in general in Section 11.6.2, appears to provide good experimental performance when applied to localization. The application of particle filtering in this context is often referred to as *Monte Carlo localization*; see the references at the end of this chapter.

12.3 Environment Uncertainty and Mapping

After reading Section 12.2, you may have already wondered what happens if the map is not given. This leads to a fascinating set of problems that are fundamental to robotics. If the state represents configuration, then the I-space allows tasks to be solved without knowing the exact configuration. If, however, the state also represents the environment, then the I-space allows tasks to be solved without even having a complete representation of the environment! This is obviously very powerful because building a representation of a robot's environment is very costly and subject to errors. Furthermore, it is likely to become quickly outdated.

12.3.1 Grid Problems

To gain a clear understanding of the issues, it will once again be helpful to consider discrete problems. The discussion here is a continuation of Section 12.2.1. In that section, the state represented a position, p, and a direction, d. Now suppose that the state is represented as (p, d, e), in which e represents the particular environment that contains the robot. This will require defining a space of environments, which is rarely represented explicitly. It is often expressed as a constraint on the types of environments that can exist. For example, the set of environments could be defined as all connected 2D grid-planning problems. The set of simply connected grid-planning problems is even further constrained.

One question immediately arises: When are two maps of an environment equivalent? Recall the maps shown in Figures 12.2a and 12.3b. The map in Figure 12.3b appears the same for every 90-degree rotation; however, the map in Figure 12.2a appears to be different. Even if it appears different, it should still be the same environment, right? Imagine mapping a remote island without having a compass that indicates the direction to the north pole. An orientation (which way is up?) for the map can be chosen arbitrarily without any harm. If a map of the environment is made by "drawing" on $\mathbb{R}^2$, it should seem that two maps are equivalent if a transformation in $SE(2)$ (i.e., translation and rotation) can be applied to overlay one perfectly on top of the other.

When defining an environment space, it is important to clearly define what it means for two environments to be equivalent. For example, if we are required to build a map by exploration, is it required to also provide the exact translation and orientation? This may or may not be required, but it is important to specify this in the problem description. Thus, we will allow any possibility: If the maps only differ by a transformation in $SE(2)$, they may or may not be defined as equivalent, depending on the application.

To consider some examples, it will be convenient to define some finite or infinite sets of environments. Suppose that planning on a 2D grid is once again considered. In this section, assume that each grid point p has integer coordinates $(i, j) \in \mathbb{Z} \times \mathbb{Z}$, as defined in Section 2.1. Let E denote a set of environments. Once again, there are four possible directions for the robot to face; let D denote this set. The state space is

$$X = \mathbb{Z} \times \mathbb{Z} \times D \times E. \tag{12.23}$$

Assume in general that an environment, $e \in E$, is specified by indicating a subset of $\mathbb{Z} \times \mathbb{Z}$ that corresponds to the positions of all of the *white* tiles on which the robot can be placed. All other tiles are *black*, which means that they are obstacles. If any subset of $\mathbb{Z} \times \mathbb{Z}$ is allowed, then $E = \text{pow}(\mathbb{Z} \times \mathbb{Z})$. This includes many useless maps, such as a checkerboard that spans the entire plane; this motivates some restrictions on E. For example, E can be restricted to be the subset of $\text{pow}(\mathbb{Z} \times \mathbb{Z})$ that corresponds to all maps that include a white tile at the origin, $(0, 0)$, and for which all other white tiles are reachable from it and lie within a bounded region.

Examples will be given shortly, but first think about the kinds of problems that can be formulated:

1. **Map building:** The task is to visit every reachable tile and construct a map. Depending on how E is defined, this may identify a particular environment in E or a set of environments that are consistent with the exploration. This may also be referred to as *simultaneous localization and mapping*, or *SLAM*, because constructing a complete map usually implies that the robot position and orientation are eventually known [131, 292]. Thus, the complete state, $x \in X$, as given in (12.23) is determined by the map-building process. For the grid problem considered here, this point is trivial, but the problem becomes more difficult for the case of probabilistic uncertainty in a continuous environment. See Section 12.3.5 for this case.

2. **Determining the environment:** Imagine that a robot is placed into a building at random and then is switched on. The robot is told that it is in one of a fixed (i.e., 10) number of buildings. It must move to determine which one. As the number of possible environments is increased, the problem appears to be more like map building. In fact, map building can be considered as a special case in which little or no constraints are initially imposed on the set of possible environments.

3. **Navigation:** In this case, a goal position is to be reached, even though the robot has no map. The location of the goal relative to the robot can be specified through a sensor. The robot is allowed to solve this problem without fully exploring the environment. Thus, the final nondeterministic I-state after solving the task could contain numerous possible environments. Only a part of the environment is needed to solve the problem.

4. **Searching:** In this case, a goal state can only be identified when it is reached (or detected by a short-range sensor). There are no additional sensors to help in the search. The environment must be systematically explored, but the search may terminate early if the goal is found. A map does not necessarily have to be constructed. Searching can be extended to *pursuit-evasion*, which is covered in Section 12.4.

Simple examples of determining the environment and navigation will now be given.

Example 12.4 (Determining the Environment) Suppose that the robot is told that it was placed into one of the environments shown in Figure 12.12. Let the initial position of the robot be $(0, 0)$, which is shown as a white circle. Let the initial direction be east and the environment be e_3. These facts are unknown to the robot. Use the same actions and state transition model as in Section 12.2.1. The current state space includes the environment, but the environment never changes. Only information regarding which environment the robot is in will change. The

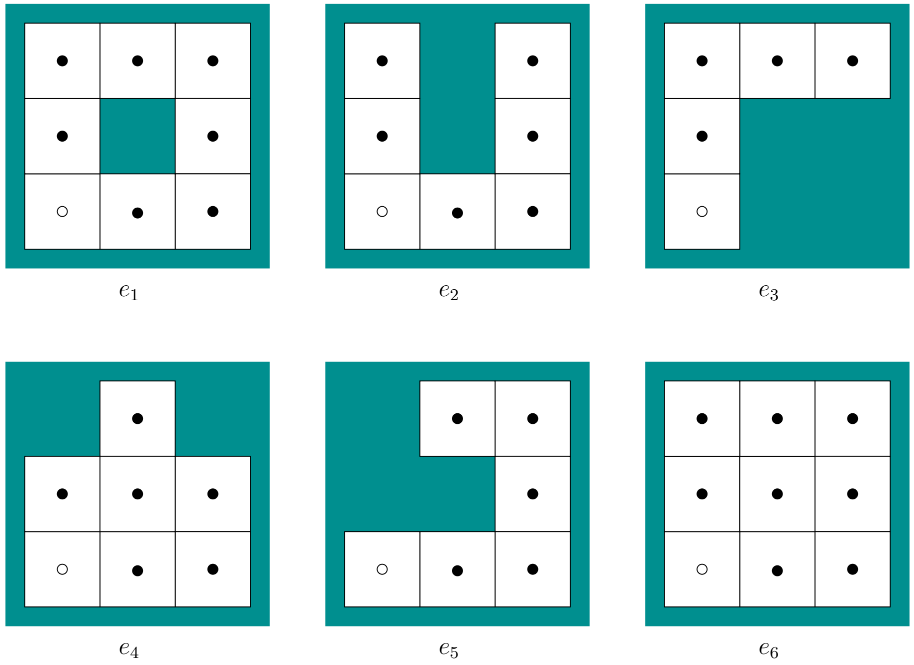


Figure 12.12: A set of possible 2D grid environments. In each case, the "up" direction represents north and the white circle represents the origin, $p = (0, 0)$.

sensing model again only indicates whether the robot has changed its position from the application of the last action.

The initial condition is X, because any position, orientation, and environment are possible. Some nondeterministic I-states will now be determined. Let $(u_1, u_2, u_3) = (\mathrm{F}, \mathrm{R}, \mathrm{R})$. From this sequence of actions, the sensor observations (y_2, y_3, y_4) report that the robot has not yet changed its position. The orientation was changed to west, but this is not known to the robot (it does, however, know that it is now pointing in the opposite direction with respect to its initial orientation). What can now be inferred? The robot has discovered that it is on a tile that is bounded on three sides by obstacles. This means that e_1 and e_6 are ruled out as possible environments. In the remaining four environments, the robot deduces that it must be on one of the end tiles: 1) the upper left of e_2, 2) the upper right of e_2, 3) the bottom of e_3, 4) the rightmost of e_3, 5) the top of e_4, 6) the lower left of e_5, or 7) the upper left of e_5. It can also make strong inferences regarding its orientation. It even knows that the action $u_4 = \mathrm{R}$ would cause it to move because all four directions cannot be blocked.

Apply $(u_4, u_5) = (\mathrm{R}, \mathrm{F})$. The robot should move two times, to arrive in the upper left of e_3 facing north. In this case, any of e_2, e_3, e_4, or e_5 are still possible;

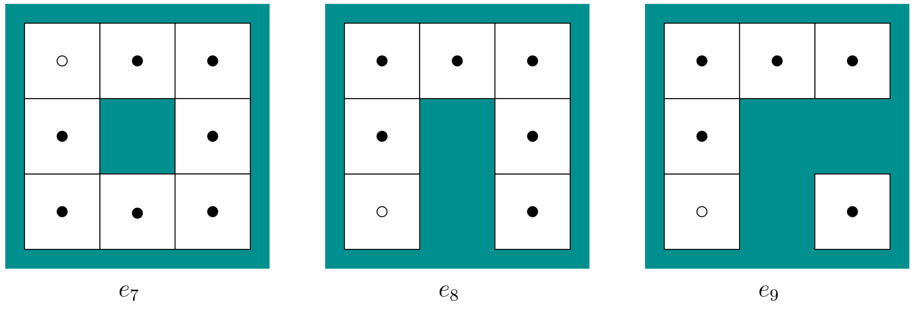


Figure 12.13: Add these environments to the set depicted in Figure 12.12. Each is essentially equivalent to an environment already given and generally does not affect the planning problem.

however, it now knows that its position at stage 4 could not have been in the upper left of e_5. If the robot is in e_3, it knows that it must be in the upper left, but it still does not know its orientation (it could be north or west). The robot could also be in the lower left or lower right of e_2.

Now let $(u_6, u_7) = (\mathrm{R}, \mathrm{F})$, which moves the robot twice. At this point, e_4 and e_5 are ruled out, and the set of possible environments is $\{e_2, e_3\}$ (one orientation from e_2 is also ruled out). If $u_8 = \mathrm{R}$ is applied, then the sensor observation y_9 reports that the robot does not move. This rules out e_2. Finally, the robot can deduce that it is in the upper right of e_3 facing south. It can also deduce that in its initial state it was in the lower left of e_3 facing east. Thus, all of the uncertainty has been eliminated through the construction of the nondeterministic I-states.

Now consider adding the environments shown in Figure 12.13 to the set and starting the problem over again. Environment e_7 is identical to e_1, except that the origin is moved, and e_8 is identical to e_2, except that it is rotated by 180 degrees. In these two cases, there exist no inputs that enable the robot to distinguish between e_1 and e_7 or between e_2 and e_8. It is reasonable to declare these environments to be pairwise equivalent. The only distinction between them is the way that the map is drawn.

If the robot executes the same action sequence as given previously, then it will also not be able to distinguish e_3 from e_9. It is impossible for the robot to deduce whether there is a white tile somewhere that is not reachable. A general environment space may include such variations, and this will prevent the robot from knowing the precise environment. However, this usually presents no additional difficulty in solving a planning problem. Therefore, it might make sense to declare e_3 and e_9 to be equivalent. The fact that tasks can be achieved without knowing the precise environment is very important. In a sense, the environment is observed at some "resolution" that is sufficient for solving a problem; further details beyond that are unimportant. Since the robot can ignore unnecessary details, cheaper and more reliable systems can often be built. ■

Example 12.5 (Reaching a Goal State) Suppose once again that the set of environments shown in Figure 12.12 is given. This time, also assume that the position $p = (0, 0)$ and orientation east are known. The environment is e_4, but it is unknown to the robot. The task is to reach the position $(2, 0)$, which means that the robot must move two tiles to the east. The plan $(u_1, u_2) = (\mathrm{F}, \mathrm{F})$ achieves the goal without providing much information about the environment. After $u_1 = \mathrm{F}$ is applied, it is known that the environment is not e_3; however, after this, no additional information is gathered regarding the environment because it is not relevant to solving the problem. If the goal had been to reach $(2, 2)$, then more information would be obtained regarding the environment. For example, if the plan is $(\mathrm{F}, \mathrm{L}, \mathrm{R}, \mathrm{L})$, then it is known that the environment is e_6. ■

Algorithms for determining the environment To determine the environment (which includes the map-building problem), it is sufficient to reach and remember all of the tiles. If the robot must determine its environment from a small set of possibilities, an optimal worst-case plan can be precomputed. This can be computed on $\vec{X} = \mathcal{I}_{ndet}$ by using value iteration or the nondeterministic version of Dijkstra's algorithm from Section 10.2.3. When the robot is dropped into the environment, it applies the optimal plan to deduce its position, orientation, and environment. If the set of possible environments is too large (possibly infinite), then a lazy approach is most suitable. This includes the map-building problem, for which there may be little or no assumptions about the environment. A lazy approach to the map-building problem simply has to ensure that every tile is visited. One additional concern may be to minimize the amount of reroute paths, which were mentioned in Section 12.2.1. A simple algorithm that solves the problem while avoiding excessive rerouting is depth-first search, from Section 2.2.2.

Algorithms for navigation The navigation task is to reach a prescribed goal, even though no environment map is given. It is assumed that the goal is expressed in coordinates relative to the robot's initial position and orientation (these are odometric coordinates). If the goal can only be identified when the robot is on the goal tile, then searching is required, which is covered next. As seen in Example 12.5, the robot is not required to learn the whole environment to solve a navigation problem. The search algorithms of Section 2.2 may be applied. For example, the A^* method will find the optimal route to the goal, and a reasonable heuristic underestimate of the cost-to-go can be defined by assuming that all tiles are empty. Although such a method will work, the reroute costs are not being taken into account. Thus, the optimal path eventually computed by A^* may be meaningless unless other robots will later use this information to reach the same goal in the same environment. For the unfortunate robot that went first, a substantial amount of exploration steps might have been wasted because A^* is not designed for exploration during execution. Even though the search algorithms in Section 2.2 assumed that the search graph was gradually revealed during execution, as opposed to being given in advance, they allow the current state in the search to jump around arbitrarily. In the current setting, this would require teleporting the robot to different parts of the environment. Section 12.3.2 covers a navigation algorithm that extends Dijkstra's algorithm to work correctly when the costs are discovered during execution. It can be nicely applied to the grid-based navigation problem presented in this section, even when the environment is initially unknown.

Algorithms for maze searching A fascinating example of using an I-map to dramatically reduce the I-space was given a long time ago by Blum and Kozen [35]. Map building requires space that is linear in the number of tiles; however, it is possible to ensure that the environment has been systematically searched using much less space. For 2D grid environments, the searching problem can be solved without maintaining a complete map. It must systematically visit every tile; however, this does not imply that it must remember all of the places that it has visited. It is important only to ensure that the robot does not become trapped in an infinite loop before covering all tiles. It was shown in [35] that any maze can be searched using space that is only logarithmic in the number of tiles. This implies that many different environments have the same representation in the machine. Essentially, an I-map was developed that severely collapses $\mathcal{I}_{ndet}$ down to a smaller derived I-space.

Assume that the robot motion model is the same as has been given so far in this section; however, no map of the environment is initially given. Whatever direction the robot is facing initially can be declared to be north without any harm. It is assumed that any planar 2D grid is possible; therefore, there are identical maps for each of the four orientations. The north direction of one of these maps might be mislabeled by arbitrarily declaring the initial direction to be north, but this is not critical for the coming approach. It is assumed that the robot is a finite automaton that carries a binary counter. The counter will be needed because it can store values that are arbitrarily large, which is not possible for the automaton alone.

To keep the robot from wandering around in circles forever, two important pieces of information need to be maintained:

1. The *latitude*, which is the number of tiles in the north direction from the robot's initial position.

2. When a loop path is executed, it needs to know its *orientation*, which means whether the loop travels clockwise or counterclockwise.

Both of these can be computed from the history I-state, which takes the same form as in (12.12), except in the current setting, X is given by (12.23) and E is the set

of all bounded environments (bounded means that the white tiles can be contained in a large rectangle). From the history I-state, let $\tilde{u}'_k$ denote the subsequence of the action history that corresponds to actions that produce motions. The latitude, $l(\tilde{u}'_k)$, can be computed by counting the number of actions that produce motions in the north direction and subtracting those that produce motions in the south direction. The loop orientation can be determined by angular odometry (which is equivalent to having a compass in this problem [79]). Let the value $r(\tilde{u}'_k)$ give the number of right turns in $\tilde{u}'_k$ minus the number of left turns in $\tilde{u}'_k$. Note that making four rights yields a clockwise loop and $r(\tilde{u}'_k) = 4$. Making four lefts yields a counterclockwise loop and $r(\tilde{u}'_k) = -4$. In general, it can be shown that for any loop path that does not intersect itself, either $r(\tilde{u}'_k) = 4$, which means that it travels clockwise, or $r(\tilde{u}'_k) = -4$, which means that it travels counterclockwise.

It was stated that a finite automaton and a binary counter are needed. The counter is used to keep track of $l(\tilde{u}'_k)$ as the robot moves. It turns out that an additional counter is not needed to measure the angular odometry because the robot can instead perform mod-3 arithmetic when counting right and left turns. If the result is $r(\tilde{u}'_k) = 1 \mod 3$ after forming a loop, then the robot traveled counterclockwise. If the result is $r(\tilde{u}'_k) = 2 \mod 3$, then the robot traveled clockwise. This observation avoids using an unlimited number of bits, contrary to the case of maintaining latitude. The construction so far can be viewed as part of an I-map that maps the history I-states into a much smaller derived I-space.

The plan will be described in terms of the example shown in Figure 12.14. For any environment, there are obstacles in the interior (this example has six), and there is an outer boundary. Using the latitude and orientation information, a unique point can be determined on the boundary of each obstacle and on the outer boundary. The *unique point* is defined as the westernmost vertex among the southernmost vertices of the obstacle. These are shown by small discs in Figure 12.15. By using the latitude and orientation information, the unique point can always be found (see Exercise 4).

To solve the problem, the robot moves to a boundary and traverses it by performing *wall following*. The robot can use its sensing information to move in a way that keeps the wall to its left. Assuming that the robot can always detect a unique point along the boundary, it can imagine that the obstacles are connected as shown in Figure 12.15. There is a fictitious thin obstacle that extends southward from each unique point. This connects the obstacles together in a way that appears to be an extension of the outer boundary. In other words, imagine that the obstacles are protruding from the walls, as opposed to "floating" in the interior. By refusing to cross these fictitious obstacles, the robot moves around the boundary of all obstacles in a single closed-loop path. The strategy so far does not ensure that every cell will be visited. Therefore, the modification shown in Figure 12.16 is needed to ensure that every tile is visited by zig-zag motions. It is interesting to compare the solution to the spanning-tree coverage planning approach in Section 7.6, which assumed a complete map was given and the goal

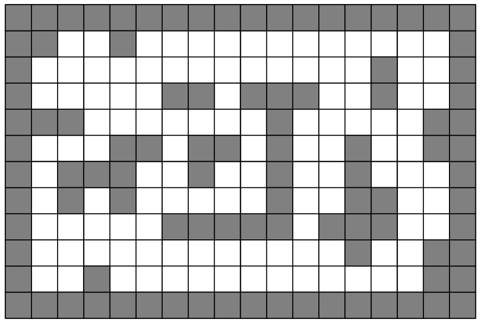


Figure 12.14: An example that has six obstacles.

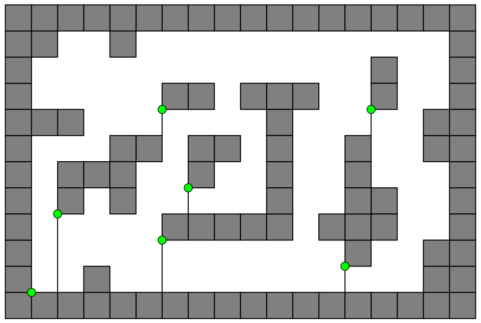


Figure 12.15: The obstacles are connected together by extending a thin obstacle downward from their unique points.

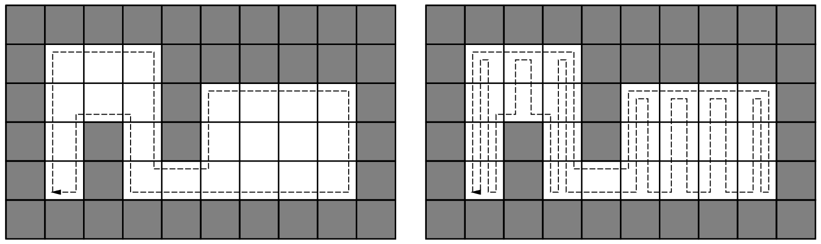
was to optimize the distance traveled.

If there is some special object in the environment that can be detected when reached by the robot, then the given strategy is always guaranteed to find it, even though at the end, it does not even have a map!

The resulting approach can be considered as an information-feedback plan on the I-space. In this sense, Blum and Kozen were the "planner" that found a plan that solves any problem. Alternative plans do not need to be computed from the problem data because the plan can handle all possible environments without modification. This is the power of working directly with an I-space over the set of environments, as opposed to requiring state estimation.

12.3.2 Stentz's Algorithm (D*)

Imagine exploring an unknown planet using a robotic vehicle. The robot moves along the rugged terrain while using a range scanner to make precise measurements of the ground in its vicinity. As the robot moves, it may discover that some parts



(a) (b)

Figure 12.16: (a) A clockwise loop produced by wall following. (b) An alternative loop that visits all of the tiles in the interior.

Figure 12.17: The Automated Cross-Country Unmanned Vehicle (XUV) is equipped with laser radar and other sensors, and uses Stentz's algorithm to navigate (courtesy of General Dynamics Robotic Systems).

were easier to traverse than it originally thought. In other cases, it might realize that some direction it was intending to go is impassable due to a large bolder or a ravine. If the goal is to arrive at some specified coordinates, this problem can be viewed as a navigation problem in an unknown environment. The resulting solution is a lazy approach, as discussed in Section 12.2.1.

This section presents *Stentz's algorithm* [267], which has been used in many outdoor vehicle navigation applications, such as the vehicle shown in Figure 12.17. The algorithm can be considered as a dynamic version of the backward variant of Dijkstra's algorithm. Thus, it maintains cost-to-go values, and the search grows outward from the goal, as opposed to cost-to-come values from x_I in the version of Dijkstra's algorithm in Section 2.3.3. The method applies to any optimal planning problem. In terms of the state transition graph, it is assumed that the costs of edge transitions are unknown (equivalently, each cost $l(x,u)$ is unknown). In the navigation problem, a positive cost indicates the difficulty of traveling from state x to state $x' = f(x,u)$.

To work with a concrete problem, imagine that a planet surface is partitioned into a high-resolution grid. The state space is simply a bounded set of grid tiles; hence, $X \subseteq \mathbb{Z} \times \mathbb{Z}$. Each grid tile is assigned a positive, real value, $c(x)$, that indicates the difficulty of its traversal. The actions $U(x)$ at each grid point can be chosen using standard grid neighbors (e.g., four-neighbors or eight-neighbors). This now defines a state transition graph over X. From any $x' \in X$ and $u' \in U(x')$ such that $x = f(x', u')$, the cost term is assigned using c as $l(x', u') = c(x)$. This model is a generalization of the grid in Section 12.3.1, in which the tiles were either empty or occupied; here any positive real value is allowed. In the coming explanation, the costs may be more general than what is permitted by starting from $c(x)$, and the state transition graph does not need to be derived from a grid. Some initial values are assigned arbitrarily for all $l(x, u)$. For example, in the planetary exploration application, the cost of traversing a level, unobstructed surface may be uniformly assumed.

The task is to navigate to some goal state, x_G. The method works by initially constructing a feedback plan, π, on a subset of X that includes both x_I and x_G. The plan, π, is computed by iteratively applying the procedure in Figure 12.18 until the optimal cost-to-go is known at x_I. A priority queue, Q, is maintained as in Dijkstra's algorithm; however, Stentz's algorithm allows the costs of elements in Q to be modified due to information sensed during execution. Let $G_{best}(x)$ denote the lowest cost-to-go associated with x during the time it spends in Q. Assume that Q is sorted according to G_{best}. Let $G_{cur}(x)$ denote its current cost-to-go value, which may actually be more than $G_{best}(x)$ if some cost updates caused it to increase. Suppose that some $u \in U(x)$ can be applied to reach a state $x' = f(x, u)$. Let $G_{via}(x, x')$ denote the cost-to-go from x by traveling via x',

$$G_{via}(x, x') = G_{cur}(x') + l(x, u). \tag{12.24}$$

If $G_{via}(x, x') < G_{cur}(x)$, then it indicates that $G_{cur}(x)$ could be reduced. If $G_{cur}(x') \leq G_{best}(x)$, then it is furthermore known that $G_{cur}(x')$ is optimal. If both of these conditions are met, then $G_{cur}(x)$ is updated to $G_{via}(x, x')$.

After the iterations of Figure 12.18 finish, the robot executes π, which generates a sequence of visited states. Let x_k denote the current state during execution. If it is discovered that if $\pi(x_k) = u_k$ would be applied, the received cost would not match the cost $l(x_k, u_k)$ in the current model, then the costs need to be updated. More generally, the robot may have to be able to update costs within a region around x_k that corresponds to the sensor field of view. For the description below, assume that an update, $l(x_k, u_k)$, is obtained for x_k only (the more general case is handled similarly). First, $l(x_k, u_k)$ is updated to the newly measured value. If x_k happened to be dead (visited, but no longer in Q), then it is inserted again into Q, with cost $G_{cur}(x_k)$. The steps in Figure 12.18 are performed until $G_{cur}(x_k) \leq G_{best}(x)$ for all $x \in Q$. Following this, the plan execution continues until either the goal is reached or another cost mismatch is discovered. At any time during execution, the robot motions are optimal given the current information about the costs [267].

STENTZ'S ALGORITHM

1. Remove x from Q, which is the state with the lowest $G_{best}(x)$ value.

2. If $G_{best}(x) < G_{cur}(x)$, then x has increased its value while on Q. If x can improve its cost by traveling via a neighboring state for which the optimal cost-to-go is known, it should do so. Thus, for every $u \in U(x)$, test for $x' = f(x, u)$ whether $G_{via}(x, x') < G_{cur}(x)$ and $G_{cur}(x') \leq G_{best}(x)$. If so, then update $G_{cur}(x) := G_{via}(x, x')$ and $\pi(x) := u$.

3. This and the remaining steps are repeated for each x' such that there exists $u' \in U(x')$ for which $x = f(x', u')$. If x' is unvisited, then assign $\pi(x') := u'$, and place x' onto Q with cost $G_{via}(x', x)$.

4. If the cost-to-go from x' appears incorrect because $\pi(x') = u'$ but $G_{via}(x', x) \neq G_{cur}(x')$, then an update is needed. Place x' onto Q with cost $G_{via}(x', x)$.

5. If $\pi(x') \neq u'$ but $G_{via}(x', x) < G_{cur}(x')$, then from x' it is better to travel via x than to use $\pi(x')$. If $G_{cur}(x) = G_{best}(x)$, then $\pi(x') := u'$ and x' is inserted into Q because the optimal cost-to-go for x is known. Otherwise, x (instead of x') is inserted into Q with its current value, $G_{cur}(x)$.

6. One final condition is needed to avoid generating cycles in π. If x' is dead (visited, but no longer in Q), it may need to be inserted back into Q with cost $G_{cur}(x')$. This must be done if $\pi(x') \neq u'$, $G_{via}(x, x') < G_{cur}(x)$, and $G_{cur}(x) > G_{best}(x)$

Figure 12.18: Stentz's algorithm, often called D^* (pronounced "dee star"), is a variant of Dijkstra's algorithm that dynamically updates cost values as the cost terms are learned during execution. The steps here are only one iteration of updating the costs after a removal of a state from Q.

Figure 12.19 illustrates the execution of the algorithm. Figure 12.19a shows a synthetic terrain that was generated by a stochastic fractal. Darker gray values indicate higher cost. In the center, very costly terrain acts as a barrier, for which an escape route exists in the downward direction. The initial state is the middle of the left edge of the environment, and the goal state is the right edge. The robot initially plans a straight-line path and then incrementally updates the path in each step as it moves. In Figure 12.19b, the robot has encountered the costly center and begins to search for a way around. Finally, the goal is reached, as shown in Figure 12.19c. The executed path is actually the result of executing a series of optimal paths, each of which is based on the known information at the time a single action is applied.

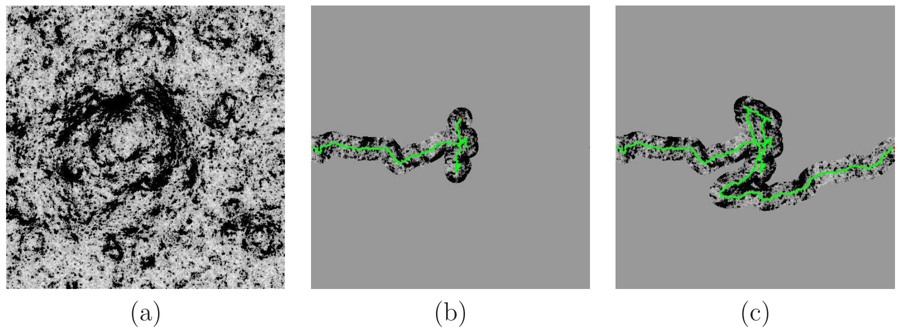


Figure 12.19: An example of executing Stentz's algorithm (courtesy of Tony Stentz).

Interpretation in terms of I-spaces An alternative formulation will now be given to help understand the connection to I-spaces of a set of environments. The state space, as defined previously, could instead be defined as a *configuration space*, $\mathcal{C} = \mathbb{Z} \times \mathbb{Z}$. Let $q \in \mathcal{C}$ denote a configuration. Suppose that each possible environment corresponds to one way to assign costs to all of the edges in a configuration transition graph. The set E of all possible environments for this problem seems to be all possible ways to assign costs, $l(q, u)$. The state space can now be defined as $\mathcal{C} \times E$, and for each state, $x = (q, e) \in X$, the configuration and complete set of costs are specified. Initially, it is guessed that the robot is in some particular $e \in E$. If a cost mismatch is discovered, this means that a different environment model is now assumed because a transition cost is different from what was expected. The costs should actually be written as $l(x, u) = l(q, e, u)$, which indicates the dependency of the costs on the particular environment is assumed.

A nondeterministic I-state corresponds to a set of possible cost assignments, along with their corresponding configurations. Since the method requires assigning costs that have not yet been observed, it takes a guess and assumes that one particular environment in the nondeterministic I-state is the correct one. As cost mismatches are discovered, it is realized that the previous guess lies outside of the updated nondeterministic I-state. Therefore, the guess is changed to incorporate the new cost information. As this process evolves, the nondeterministic I-state continues to shrink. Note, however, that in the end, the robot may solve the problem while being incorrect about the precise $e \in E$. Some tiles are never visited, and their true costs are therefore unknown. A default assumption about their costs was made to solve the problem; however, the true $e \in E$ can only be known if all tiles are visited. It is only true that the final assumed default values lie within the final nondeterministic I-state.

12.3.3 Planning in Unknown Continuous Environments

We now move from discrete to continuous environments but continue to use nondeterministic uncertainty. First, several *bug algorithms* [139, 180, 140] are presented, which represent a family of motion plans that solve planning problems using ideas that are related in many ways to the maze exploration ideas of Section 12.3.1. In addition to bug algorithms, the concept of competitive ratios is also briefly covered.

The following model will be used for bug algorithms. Suppose that a point robot is placed into an unknown 2D environment that may contain any finite number of bounded obstacles. It is assumed that the boundary of each obstacle and the outer boundary (if it exists) are piecewise-analytic (here, *analytic* implies that each piece is smooth and switches its curvature sign only a finite number of times). Thus, the obstacles could be polygons, smooth curves, or some combination of curved and linear parts. The set E of possible environments is overwhelming, but it will be managed by avoiding its explicit construction. The robot configuration is characterized by its position and orientation.

There are two main sensors:[2]

1. A *goal sensor* indicates the current Euclidean distance to the goal and the direction to the goal, expressed with respect to an absolute "north."

2. A *local visibility sensor* provides the exact shape of the boundary within a small distance from the robot. The robot must be in contact or almost in contact to observe part of the boundary; otherwise, the sensor provides no useful information.

The goal sensor essentially encodes the robot's position in polar coordinates (the goal is the origin). Therefore, unique (x, y) coordinates can be assigned to any position visited by the robot. This enables it to incrementally trace out obstacle boundaries that it has already traversed. The local visibility sensor provides just enough information to allow wall-following motions; the range of the sensor is very short so that the robot cannot learn anything more about the structure of the environment.

Some strategies will now be considered for the robot. Each of these can be considered as an information-feedback plan on a nondeterministic I-space.

The Bug1 strategy A strategy called *Bug1* was developed in [180] and is illustrated in Figure 12.20. The execution is as follows:

1. Move toward the goal until an obstacle or the goal is encountered. If the goal is reached, then stop.

[2]This is just one possible sensing model. Alternative combinations of sensors may be used, provided that they enable the required motions and decisions to be executed in the coming motion strategies.

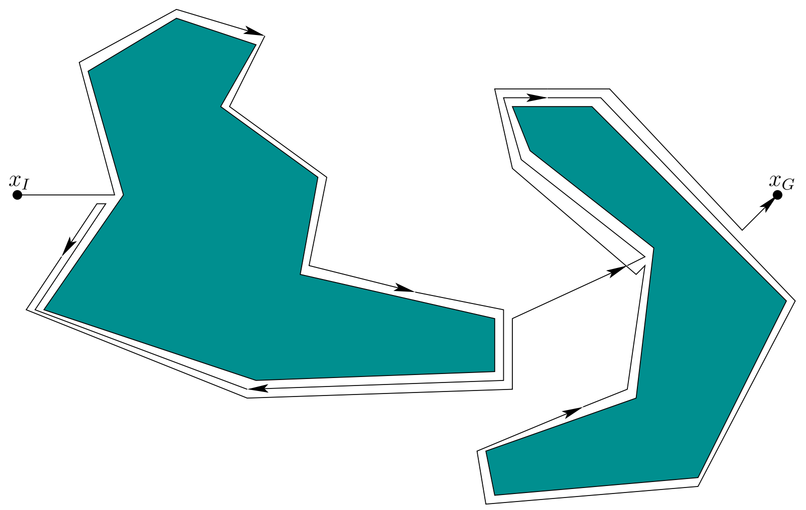


Figure 12.20: An illustration of the Bug1 strategy.

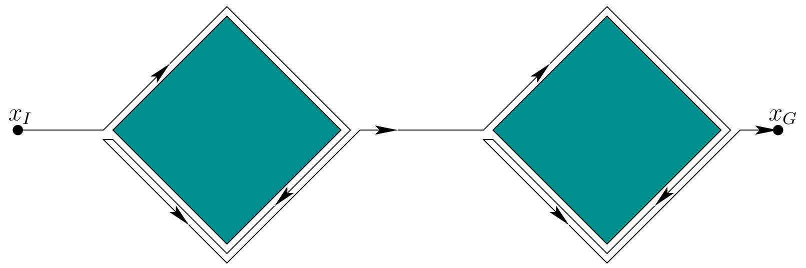


Figure 12.21: A bad example for Bug1. The perimeter of each obstacle is spanned one and a half times.

2. Turn left and follow the entire perimeter of the contacted obstacle. Once the full perimeter has been visited, then return to the point at which the goal was closest, and go to Step 1.

Determining that the entire perimeter has been traversed may seem to require a pebble or marker; however, this can be inferred by finding the point at which the goal sensor reading repeats.

The worst case is conceptually simple to understand. The total distance traveled by the robot is no greater than

$$d + \frac{3}{2} \sum_{i=1}^{M} p_i, \tag{12.25}$$

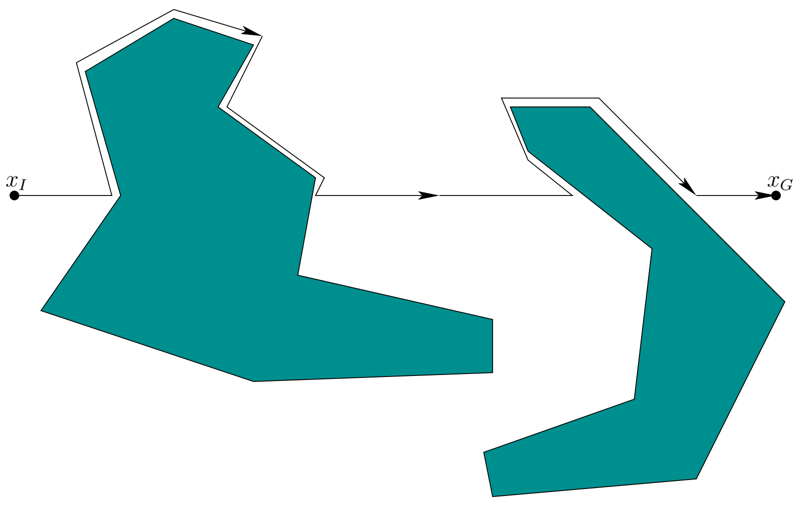


Figure 12.22: An illustration of the Bug2 strategy.

in which d is the Euclidean distance from the initial position to the goal position, p_i is the perimeter of the ith obstacle, and M is the number of obstacles. This means that the boundary of each obstacle is followed no more than 3/2 times. Figure 12.21 shows an example in which each obstacle is traversed 3/2 times. This bound relies on the fact that the robot can always recall the shortest path along the boundary to the point from which it needs to leave. This seems reasonable because the robot can infer its distance traveled along the boundary from the goal sensor. If this was not possible, then the 3/2 would have to be replaced by 2 because the robot could nearly traverse the full boundary twice in the worst case.

The Bug2 strategy An alternative to Bug1 is the *Bug2 strategy*, which is illustrated in Figure 12.22. The robot always attempts to move along a line that connects the initial and goal positions. When the robot is on this line, the goal direction will be either the same as from the initial state or it will differ by π radians (if the robot is on the other side of the goal). The first step is the same as for Bug1. In the second step, the robot follows the perimeter only until the line is reached and it is able to move in the direction toward the goal. From there, it goes to Step 1. As expressed so far, it is possible that infinite cycles occur. Therefore, a small modification is needed. The robot remembers the distance to the goal from the last point at which it departed from the boundary, and only departs from the boundary again if the candidate point that is closer to the goal. This is applied iteratively until the goal is reached or it is deemed to be impossible.

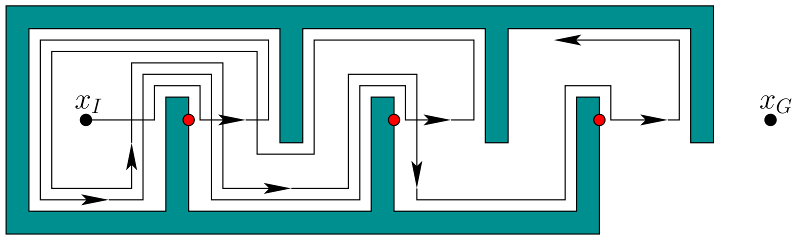


Figure 12.23: A bad case for Bug2. Only part of the resulting path is shown. Points from which the robot can leave the boundary are indicated.

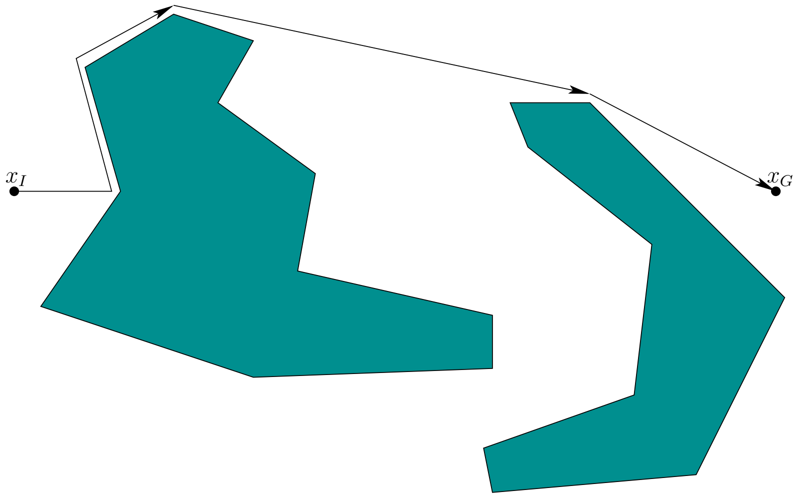


Figure 12.24: An illustration of the VisBug strategy with unlimited radius.

For the Bug2 strategy, the total distance traveled is no more than

$$d + \frac{1}{2} \sum_{i=1}^{M} n_i p_i, \tag{12.26}$$

in which n_i is the number of times the ith obstacle crosses the line segment between the initial position and the goal position. An example that illustrates the trouble caused by the crossings is shown in Figure 12.23.

Using range data The *VisBug* [179] and *TangentBug* [140, 160] strategies incorporate distance measurements made by a range or visibility sensor to improve the efficiency. The TangentBug strategy will be described here and is illustrated in Figure 12.24. Suppose that in addition to the sensors described previously, it

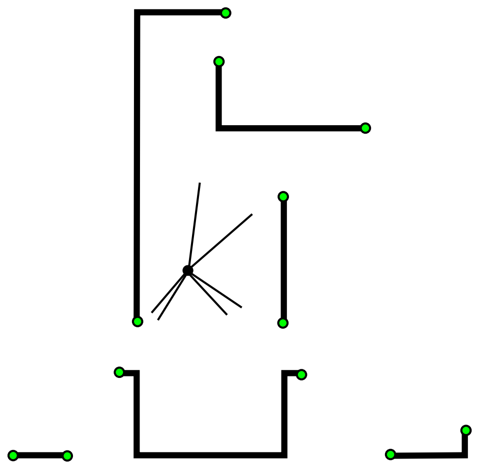


Figure 12.25: The candidate motions with respect to the range sensor are the directions in which there is a discontinuity in the depth map. The distances from the robot to the small circles are used to select the desired motion.

is also equipped with a sensor that produces measurements as shown in Figure 12.25. The strategy is as follows:

1. Move toward the goal, either through the interior of the space or by wall following, until it is realized that the robot is trapped in a local minimum or the goal is reached. This is similar to the gradient-descent motion of the potential-field planner of Section 5.4.3. If the goal is reached, then stop; otherwise, go to the next step.

2. Execute motions along the boundary. First, pick a direction by comparing the previous heading to the goal direction. While moving along the boundary, keep track of two distances: d_f and d_r. The distance d_f is the minimal distance from the goal, observed while traveling along the boundary. The distance d_r is the length of the shortest path from the current position to the goal, assuming that the only obstacles are those visible by the range sensor. The robot stops following the boundary if $d_r < d_f$. In this case, go to Step 1. If the robot loops around the entire obstacle without this condition occurring, then the algorithm reports that the goal is not reachable.

A one-parameter family of TangentBug algorithms can be made by setting a depth limit for the range sensor. As the maximum depth is decreased, the robot becomes more short-sighted and performance degrades. It is shown in [140] that the distance traveled is no greater than

$$d + \sum_{i=1}^{M} p_i + \sum_{i=1}^{M} p_i m_i, \tag{12.27}$$

in which m_i is the number of local minima for the ith obstacle and d is the initial distance to the goal. The bound is taken over M obstacles, which are assumed to intersect a disc of radius d, centered at the goal (all others can be ignored). A variant of the TangentBug, called WedgeBug, was developed in [160] for planetary rovers that have a limited field of view.

Competitive ratios A popular way to evaluate algorithms that utilize different information has emerged from the algorithms community. The idea is to compute a *competitive ratio*, which places an *on-line algorithm* in competition with an algorithm that receives more information [185, 261]. The idea can generally be applied to plans. First a cost is formulated, such as the total distance that the robot travels to solve a navigation task. A competitive ratio can then be defined as

$$\max_{e \in E} \frac{\text{Cost of executing the plan that does not know } e \text{ in advance.}}{\text{Cost of executing the plan that knows } e \text{ in advance}}. \tag{12.28}$$

The maximum is taken over all $e \in E$, which is usually an infinite set, as in the case of the bug algorithms. A competitive ratio for a navigation problem can be made by comparing the optimal distance to the total distance traveled by the robot during the execution of the on-line algorithm. Since E is infinite, many plans fail to produce a finite competitive ratio. The bug algorithms, while elegant, represent such an example. Imagine a goal that is very close, but a large obstacle boundary needs to be explored. An obstacle boundary can be made arbitrarily large while making the optimal distance to the goal very small. When evaluated in (12.28), the result over all environments is unbounded. In some contexts, the ratio may still be useful if expressed as a function of the representation. For example, if E is a polygon with n edges, then an $O(\sqrt{n})$ competitive ratio means that (12.28) is bounded over all n by $c\sqrt{n}$ for some $c \in \mathbb{R}$. For competitive ratio analysis in the context of bug algorithms, see [108].

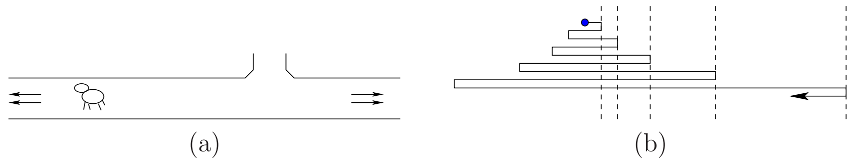


Figure 12.26: (a) A lost cow must find its way to the gate, but it does not know in which direction the gate lies. (b) If there is no bound on the distance to the gate, then a doubling spiral strategy works well, producing a competitive ratio of 9.

A nice illustration of competitive ratio analysis and issues is provided by the *lost-cow problem* [10]. As shown in Figure 12.26a, a short-sighted cow is following along an infinite fence and wants to find the gate. This makes a convenient one-dimensional planning problem. If the location of the gate is given, then the cow can reach it by traveling directly. If the cow is told that the gate is exactly distance 1 away, then it can move one unit in one direction and return to try the other direction if the gate has not been found. The competitive ratio in this case (the set of environments corresponds to all gate placements) is 3. What if the cow is told only that the gate is at least distance 1 away? In this case, the best strategy is a *spiral search*, which is to zig-zag back and forth while iteratively doubling the distance traveled in each direction, as shown in Figure 12.26b. In other words: left one unit, right one unit, left two units, right two units, left four units, and so on. The competitive ratio for this strategy turns out to be 9, which is optimal. This approach resembles iterative deepening, which was covered in Section 2.2.2.

12.3.4 Optimal Navigation Without a Geometric Model

This section presents *gap navigation trees* (GNTs) [280, 282], which are a data structure and associated planning algorithm for performing optimal navigation in the continuous environments that were considered in Section 12.3.3. It is assumed in this section that the robot is equipped with a gap sensor, as depicted in Figure 11.16 of Section 11.5.1. At every instant in time, the robot has available one action for each gap that is visible in the gap sensor. If an action is applied, then the robot moves toward the corresponding gap. This can be applied over continuous time, which enables the robot to "chase" a particular gap. The robot has no other sensing information: It has no compass and no ability to measure distances. Therefore, it is impossible to construct a map of the environment that contains metric information.

Assume that the robot is placed into an unknown but simply connected planar environment, X. The GNT can be extended to the case of multiply connected environments; however, in this case there are subtle issues with distinguishability, and it is only possible to guarantee optimality within a homotopy class of paths [281]. By analyzing the way that critical events occur in the gap sensor, a tree representation can be built that indicates how to move optimally in the environment, even though precise measurements cannot be taken. Since a gap sensor cannot even measure distances, it may seem unusual that the robot can move along shortest paths without receiving any distance (or metric) information. This will once again illustrate the power of I-spaces.

The appearance of the environment *relative to the position of the robot* is encoded as a tree that indicates how the gaps change as the robot moves. It provides the robot with sufficient information to move to any part of the environment while traveling along the shortest path. It is important to understand that the tree does not correspond to some static map of the environment. It expresses how the environment appears relative to the robot and may therefore change as the robot moves in the environment.

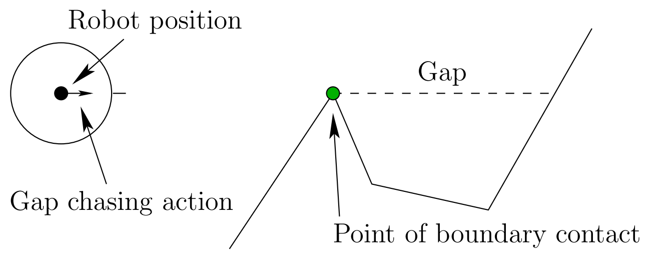


Figure 12.27: A gap-chasing action is applied, which moves the robot straight in the direction of the gap until the boundary is contacted. Once this occurs, a new part of the environment becomes visible.

The root of the tree represents the gap sensor. For each gap that currently appears in the sensor, an edge is connected to the root. Let these edges be called *root edges*. Each root edge corresponds to an action that can be applied by the robot. By selecting a root edge, the action moves the robot along a straight line toward that gap. Thus, there is a simple control model that enables the robot to move precisely toward a particular point along the boundary, ∂X, as shown in Figure 12.27.

Let $V(x)$ be the *visibility region*, which is the set of all points in X that are visible from x. Let $X \setminus V(x)$ be called the *shadow region*, which is the set of all points *not* visible from x. Let each connected component of the shadow region be called a *shadow component*. Every gap in the gap sensor corresponds to a line segment in X that touches ∂X in two places (for example, see Figure 11.15a). Each of these segments forms a boundary between the visibility region and a shadow component. If the robot would like to travel to this shadow component, the shortest way is to move directly to the gap. When moving toward a gap, the robot eventually reaches ∂X, at which point a new action must be selected.

Critical gap events As the robot moves, several important events can occur in the gap sensor:

1. **Disappear:** A gap disappears because the robot crosses an *inflection ray* as shown in Figure 12.28. This means that some previous shadow component is now visible.

2. **Appear:** A gap appears because the robot crosses an inflection ray in the opposite direction. This means that a new shadow component exists, which represents a freshly hidden portion of the environment.

3. **Split:** A gap splits into two gaps because the robot crosses a *bitangent ray*, as shown in Figure 12.29 (this was also shown in Figure 12.5). This means that one shadow component splits into two shadow components.

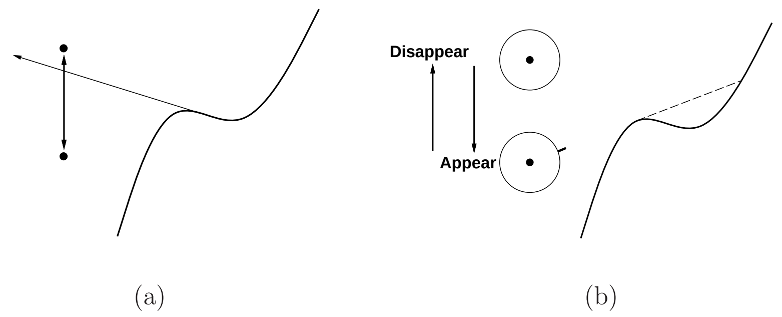


Figure 12.28: (a) The robot crosses a ray that extends from an inflectional tangent. (b) A gap appears or disappears from the gap sensor, depending on the direction.

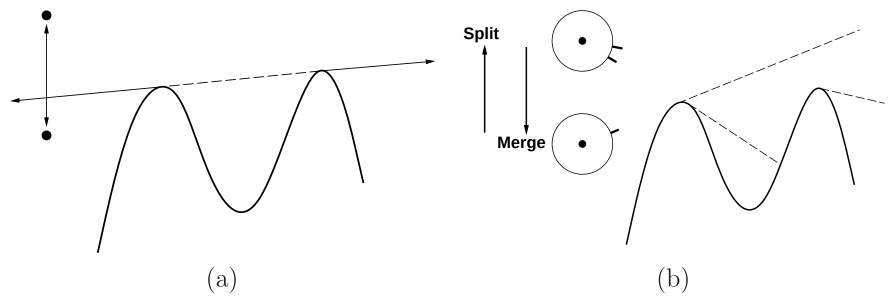


Figure 12.29: (a) The robot crosses a ray that extends from a bitangent. (b) Gaps split or merge, depending on the direction.

4. **Merge:** Two gaps merge into one because the robot crosses a bitangent ray in the oppose direction. In this case, two shadow components merge into one.

This is a complete list of possible events, under a *general position* assumption that precludes environments that cause degeneracies, such as three gaps that merge into one or the appearance of a gap precisely where two other gaps split.

As each of these gap events occurs, it needs to be reflected in the tree. If a gap disappears, as shown in Figure 12.30, then the corresponding edge and vertex are simply removed. If a merge event occurs, then an intermediate vertex is inserted as shown in Figure 12.31. This indicates that if that gap is chased, it will split into the two original gaps. If a split occurs, as shown in Figure 12.32, then the intermediate vertex is removed. The appearance of a gap is an important case, which generates a *primitive vertex* in the tree, as shown in Figure 12.33. Note that a primitive vertex can never split because chasing it will result in its disappearance.

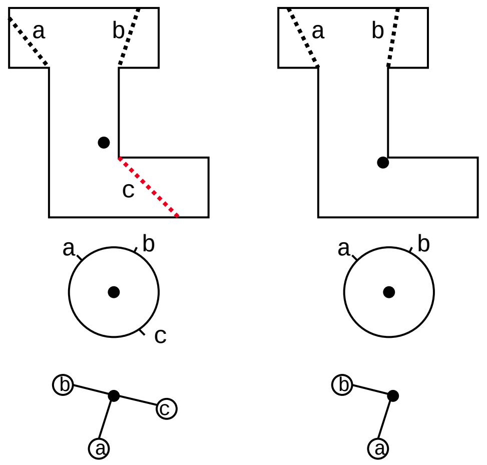


Figure 12.30: If a gap disappears, it is simply removed from the GNT.

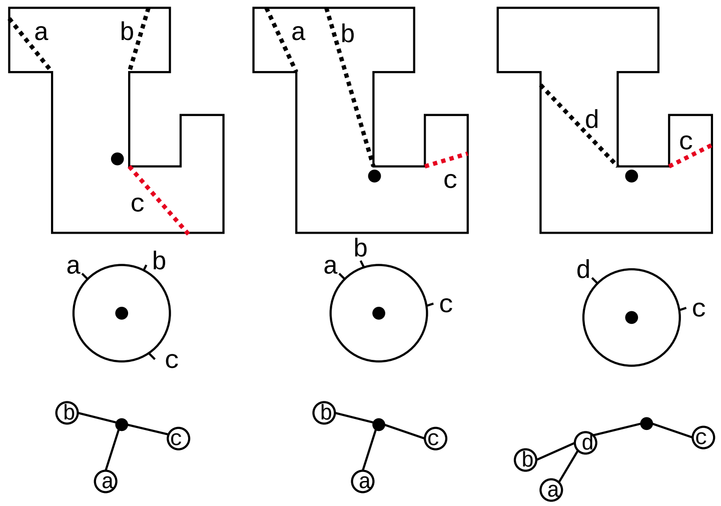


Figure 12.31: If two gaps merge, an intermediate vertex is inserted into the tree.

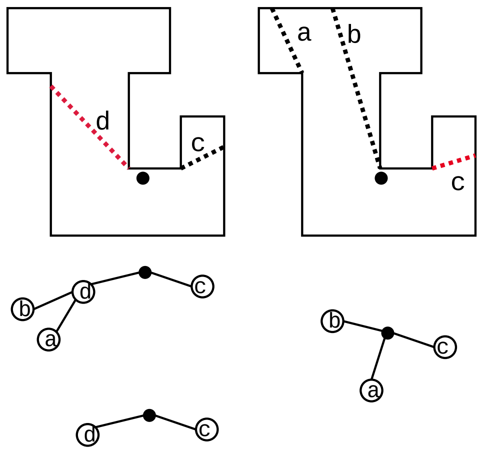


Figure 12.32: If two gaps split, the intermediate vertex is removed.

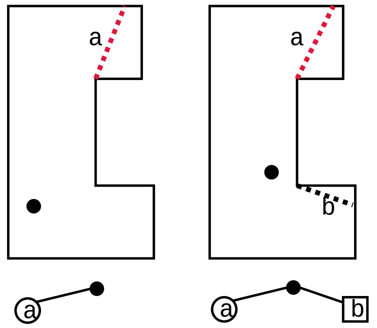


Figure 12.33: The appearance of a gap results in a primitive vertex, which is denoted by a square.

A simple example will now be considered.

Example 12.6 (Gap Navigation Tree) Suppose that the robot does not know the environment in Figure 12.34. It moves from cells 1 to 7 in order and then returns to cell 1. The following sequence of trees occurs: T_1, ..., T_7, T_6', ..., T_1', as shown in Figure 12.35. The root vertex is shown as a solid black disc. Vertices that are not known to be primitive are shown as circles; primitive vertices are squares. Note that if any leaf vertex is a circle, then it means that the shadow region of R that is hidden by that gap has not been completely explored. Note that once the robot reaches cell 5, it has seen the whole environment. This occurs precisely when all leaf vertices are primitive. When the robot returns to the first region, the tree is larger because it knows that the region on the right is composed

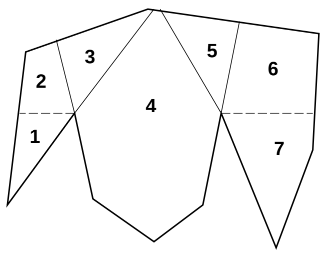


Figure 12.34: A simple environment for illustrating the gap navigation tree.

of two smaller regions to the right. If all leaves are squares, this means that the environment has been completely explored. ■

In the example, all of the interesting parts of the environment were explored. From this point onward, all leaf vertices will be primitive vertices because all possible splits have been discovered. In a sense, the environment has been completely learned, at the level of resolution possible with the gap sensor. A simple strategy for exploring the environment is to chase any gaps that themselves are nonprimitive leaf vertices or that have children that are nonprimitive leaf vertices. A leaf vertex in the tree can be *chased* by repeatedly applying actions that chase its corresponding gap in the gap sensor. This may cause the tree to incrementally change; however, there is no problem if the action is selected to chase whichever gap hides the desired leaf vertex, as shown in Figure 12.36. Every nonprimitive leaf vertex will either split or disappear. After all nonprimitive leaf vertices have been chased, all possible splits have been performed and only primitive leaves remain. In this case, the environment has been completely learned.

Using the GNTs for optimal navigation Since there is no precise map of the environment, it is impossible to express a goal state using coordinates in $\mathbb{R}^2$. However, a goal can be expressed in terms of the vertex that must be chased to make the state visible. For example, imagine showing the robot an object while it explores. At first, the object is visible, but a gap may appear that hides the object. After several merges, a vertex deep in the tree may correspond to the location from which the object is visible. The robot can navigate back to the object optimally by chasing the vertex that first hid the object by its appearance. Once this vertex and its corresponding gap disappear, the object becomes visible. At this time the robot can move straight toward the object (assuming an additional sensor that indicates the direction of the object). It was argued in [282] that when the robot chases a vertex in the GNT, it precisely follows the paths of the shortest-path roadmap, which was introduced in Section 6.2.4. Each pair of successive gap events corresponds to the traversal of a bitangent edge.

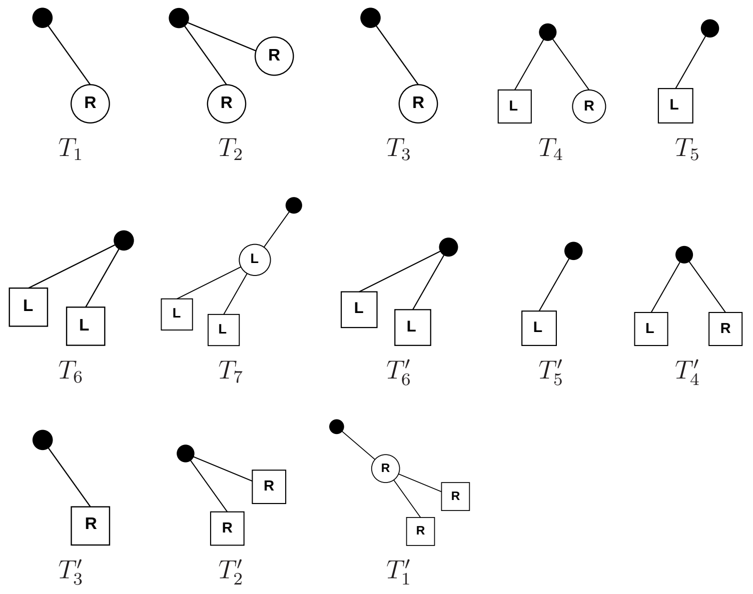


Figure 12.35: Building a representation of the environment in Figure 12.34 using the gap navigation tree. The sequence is followed from left to right. For convenience, the "R" or "L" inside of each vertex indicates whether the shadow component is to the right or left of the gap, respectively. This information is not needed by the algorithm, but it helps in understanding the representation.

I-space interpretation In terms of an I-space over the set of environments, the GNT considers large sets of environments to be equivalent. This means that an I-map was constructed on which the derived I-space is the set of possible GNTs. Under this I-map, many environments correspond to the same GNT. Due to this, the robot can accomplish interesting tasks without requesting further information. For example, if two environments differ only by rotation or scale, the GNT representations are identical. Surprisingly, the robot does not even need to be concerned about whether the environment boundary is polygonal or curved. The only important concern is how the gaps events occur. For example, the environments in Figure 12.37 all produce the same GNTs and are therefore indistinguishable to the robot. In the same way that the maze exploring algorithm of Section 12.3.1 did not need a complete map to locate an object, the GNT does not need one to perform optimal navigation.

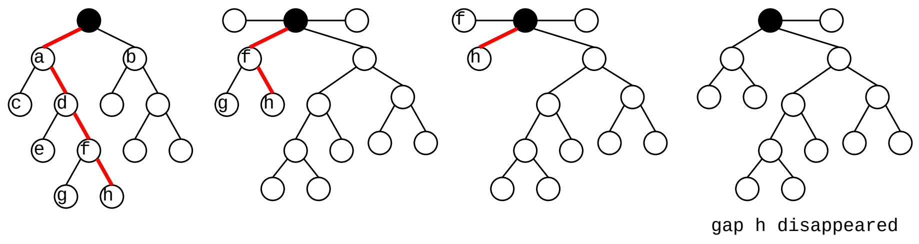


Figure 12.36: Optimal navigation to a specified part of the environment is achieved by "chasing" the desired vertex in the GNT until it disappears. This will make a portion of the environment visible. In the example, the gap labeled "h" is chased.

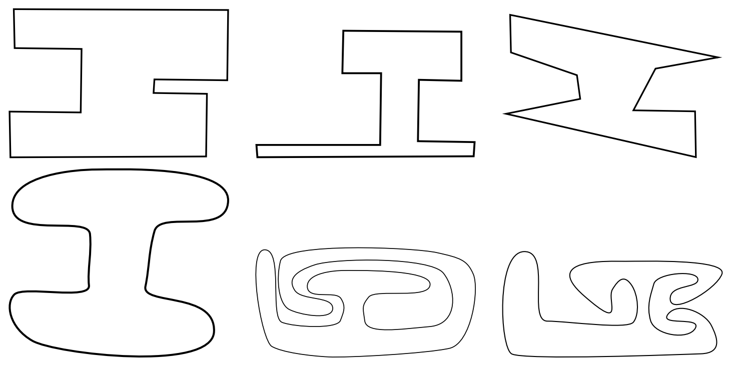


Figure 12.37: These environments yield the same GNTs and are therefore equivalent at the resolution of the derived I-space. The robot cannot measure distances and does not even care whether walls are straight or curved; it is not relevant to the navigation task. Nevertheless, it executes optimal motions in terms of the Euclidean distance traveled.

12.3.5 Probabilistic Localization and Mapping

The problems considered so far in Section 12.3 have avoided probabilistic modeling. Suppose here that probabilistic models exist for the state transitions and the observations. Many problems can be formulated by replacing the nondeterministic models in Section 12.3.1 by probabilistic models. This would lead to probabilistic I-states that represent distributions over a set of possible grids and a configuration within each grid. If the problem is left in its full generality, the I-space is enormous to the point that is seems hopeless to approach problems in the manner used to far. If optimality is not required, then in some special cases progress may be possible.

The current problem is to construct a map of the environment while simultaneously localizing the robot with the respect to the map. Recall Figure 1.7

from Section 1.2. The section covers a general framework that has been popular in mobile robotics in recent years (see the literature suggested at the end of the chapter). The discussion presented here can be considered as a generalization of the discussion from Section 12.2.3, which was only concerned with the localization portion of the current problem. Now the environment is not even known. The current problem can be interpreted as localization in a state space defined as

$$X = \mathcal{C} \times E, \tag{12.29}$$

in which $\mathcal{C}$ is a configuration space and E is the environment space. A state, x_k, is represented as $x_k = (q_k, e)$; there is no k subscript for e because the environment is assumed to be static). The history I-state provides the data to use in the process of determining the state. As for localization in Section 12.2, there are both passive and active versions of the problem. An incremental version of the active problem is sometimes called the *next-best-view problem* [11, 60, 227]. The difficulty is that the robot has opposing goals of: 1) trying to turn on the sensor at places that will gain as much new data as possible, and 2) this minimization of redundancy can make it difficult to fuse all of the measurements into a global map. The passive problem will be described here; the methods can be used to provide information for solving the active problem.

Suppose that the robot is a point that translates and rotates in $\mathbb{R}^2$. According to Section 4.2, this yields $\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$, which represents $SE(2)$. Let $q \in \mathcal{C}$ denote a configuration, which yields the position and orientation of the robot. Assume that configuration transitions are modeled probabilistically, which requires specifying a probability density, $p(q_{k+1}|q_k, u_k)$. This can be lifted to the state space to obtain $p(x_{k+1}|x_k, u_k)$ by assuming that the configuration transitions are independent of the environment (assuming no collisions ever occur). This replaces q_k and q_{k+1} by x_k and x_{k+1}, respectively, in which $x_k = (q_k, e)$ and $x_{k+1} = (q_{k+1}, e)$ for any $e \in E$.

Suppose that observations are obtained from a depth sensor, which ideally would produce measurements like those shown in Figure 11.15b; however, the data are assumed to be noisy. The probabilistic model discussed in Section 12.2.3 can be used to define $p(y|x)$. Now imagine that the robot moves to several parts of the environment, such as those shown in Figure 11.15a, and performs a sensor sweep in each place. If the configuration q_k is not known from which each sweep y_k was performed, how can the data sets be sewn together to build a correct, global map of the environment? This is trivial after considering the knowledge of the configurations, but without it the problem is like putting together pieces of a jigsaw puzzle. Thus, the important data in each stage form a vector, (y_k, q_k). If the sensor observations, y_k, are not tagged with a configuration, q_k, from which they are taken, then the jigsaw problem arises. If information is used to tightly constrain the possibilities for q_k, then it becomes easier to put the pieces together. This intuition leads to the following approach.

The EM algorithm The problem is often solved in practice by applying the *expectation-maximization* (EM) algorithm [29]. In the general framework, there are three different spaces:

1. A set of parameters, which are to be determined through some measurement and estimation process. In our problem, this represents E, because the main goal is to determine the environment.

2. A set of data, which provide information that can be used to estimate the parameter. In the localization and mapping problem, this corresponds to the history I-space $\mathcal{I}_K$. Each history I-state $\eta_K \in \mathcal{I}_K$ is $\eta_K = (p(x), \tilde{u}_{K-1}, \tilde{y}_K)$, in which $p(x)$ is a prior probability density over X.

3. A set of hidden variables, which are unknown but need to be estimated to complete the process of determining the parameters. In the localization and mapping problem, this is the configuration space $\mathcal{C}$.

Since both the parameters and the hidden variables are unknown, the choice between the two may seem arbitrary. It will turn out that expressions can be derived to nicely express the probability density for the hidden variables, but the parameters are much more complicated.

The EM algorithm involves an expectation step followed by a maximization step. The two steps are repeated as necessary until a solution with the desired accuracy is obtained. The method is guaranteed to converge under general conditions [71, 293, 294]. In practice, it appears to work well even under cases that are not theoretically guaranteed to converge [277].

From this point onward, let E, $\mathcal{I}_K$, and $\mathcal{C}$ denote the three spaces for the EM algorithm because they pertain directly to the problem. Suppose that a robot has moved in the environment for $K-1$ stages, resulting in a final stage, K. At each stage, $k \in \{1, \ldots, K\}$, an observation, y_k, is made using its sensor. This could, for example, represent a set of distance measurements made by sonars or a range scanner. Furthermore, an action, u_k, is applied for $k = 1$ to $k = K$. A prior probability density function, $p(x)$, is initially assumed over X. This leads to the history I-state, η_k, as defined in (11.14).

Now imagine that K stages have been executed, and the task is to estimate e. From each q_k, a measurement, y_k, of part of the environment is taken. The EM algorithm generates a sequence of improved estimates of e. In each execution of the two EM steps, a new estimate of $e \in E$ is produced. Let $\hat{e}_i$ denote this estimate after the ith iteration. Let $\tilde{q}_K$ denote the configuration history from stage 1 to stage K. The expectation step computes the expected likelihood of η_K

given $\hat{e}_i$. This can be expressed as[3]

$$\begin{aligned} Q(e, \hat{e}_{i-1}) =& E\left[p(\eta_K, \tilde{q}_K |\, e) |\, \eta_K, \hat{e}_{i-1}\right] \\ =& \int_{\mathcal{C}} p(\eta_K, \tilde{q}_K |\, e) p(\tilde{q}_K |\, \eta_K, \hat{e}_{i-1}) d\tilde{q}_K, \end{aligned} \tag{12.30}$$

in which the expectation is taken over the configuration histories. Since η_K is given and the expectation removes $\tilde{q}_k$, (12.30) is a function only of e and $\hat{e}_{i-1}$. The term $p(\eta_K, \tilde{q}_K |\, e)$ can be expressed as

$$p(\eta_K, \tilde{q}_K |\, e) = p(\tilde{q}_K |\, \eta_K, e) p(\eta_K | e), \tag{12.31}$$

in which $p(\eta_K)$ is a prior density over the I-space, given nothing but the environment e. The factor $p(\tilde{q}_K |\, \eta_K, e)$ differs from the second factor of the integrand in (12.30) only by using e or $\hat{e}_{i-1}$. The main difficulty in evaluating (12.30) is to evaluate $p(\tilde{q}_k |\, \eta_K, \hat{e}_{i-1})$ (or the version that uses e). This is essentially a localization problem with a given map, as considered in Section 12.2.3. The information up to stage k can be applied to yield the probabilistic I-state $p(q_k |\, \eta_k, \hat{e}_{i-1})$ for each q_k; however, this neglects the information from the remaining stages. This new information can be used to make inferences about old configurations. For example, based on current measurements and memory of the actions that were applied, we have better information regarding the configuration several stages ago. In [278] a method of computing $p(q_k |\, \eta_k, \hat{e}_{i-1})$ is given that computes two terms: One is $p(q_k | \eta_k)$, and the other is a backward probabilistic I-state that starts at stage K and runs down to $k+1$.

Note that once determined, (12.30) is a function only of e and $\hat{e}_{i-1}$. The maximization step involves selecting an $\hat{e}_i$ that minimizes (12.30):

$$\hat{e}_i = \underset{e \in E}{\operatorname{argmax}}\, Q(e, \hat{e}_{i-1}). \tag{12.32}$$

This optimization is often too difficult, and convergence conditions exist if $\hat{e}_i$ is chosen such that $Q(\hat{e}_i, \hat{e}_{i-1}) > Q(\hat{e}_{i-1}, \hat{e}_{i-1})$. Repeated iterations of the EM algorithm result in a kind of gradient descent that arrives at a local minimum in E.

One important factor in the success of the method is in the representation of E. In the EM computations, one common approach is to use a set of landmarks, which were mentioned in Section 11.5.1. These are special places in the environment that can be identified by sensors, and if correctly classified, they dramatically improve localization. In [278], the landmarks are indicated by a user as the robot travels. Classification and positioning errors can both be modeled probabilistically and incorporated into the EM approach. Another idea that dramatically simplifies

[3]In practice, a logarithm is applied to $p(\eta_K, q_k |\, e)$ because densities that contain exponentials usually arise. Taking the logarithm makes the expressions simpler without affecting the result of the optimization. The log is not applied here because this level of detail is not covered.

the representation of E is to approximate environments with a fine-resolution grid. Probabilities are associated with grid cells, which leads to a data structure called an *occupancy grid* [88, 190, 244]. In any case, E must be carefully defined to ensure that reasonable prior distributions can be made for $p(e)$ to initialize the EM algorithm as the robot first moves.

12.4 Visibility-Based Pursuit-Evasion

This section considers *visibility-based pursuit-evasion* [167, 273], which was described in Section 1.2 as a game of hide-and-seek. The topic provides an excellent illustration of the power of I-space concepts.

12.4.1 Problem Formulation

The problem considered in this section is formulated as follows.

Formulation 12.1 (Visibility-Based Pursuit-Evasion)

1. A given, continuous environment region $R \subset \mathbb{R}^2$, which is an open set that is bounded by a simple closed curve. The boundary ∂R is often a polygon, but it may be any piecewise-analytic closed curve.

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

3. An *evader*, which is a moving point in R. The evader position $e(t)$ at time $t \in T$ is determined by a continuous *position function*, $\tilde{e} : [0, 1] \rightarrow R$.[4]

4. A *pursuer*, which is a moving point in R. The evader position function $\tilde{e}$ is unknown to the pursuer.

5. A *visibility sensor*, which defines a set $V(r) \subseteq R$ for each $r \in R$.

The task is to find a path, $\tilde{p} : [0, 1] \rightarrow R$, for the pursuer for which the evader is guaranteed to be detected, regardless of its position function. This means that $\exists t \in T$ such that $e(t) \in V(p(t))$. The speed of the pursuer is not important; therefore, the time domain may be lengthened as desired, if the pursuer is slow.

It will be convenient to solve the problem by verifying that there is no evader. In other words, find a path for the pursuer that upon completion guarantees that there are no remaining places where the evader could be hiding. This ensures that during execution of the plan, the pursuer will encounter any evader. In fact, there can be any number of evaders, and the pursuer will find all of them. The approach systematically eliminates any possible places where evaders could hide.

[4]Following from standard function notation, it is better to use $\tilde{e}(t)$ instead of $e(t)$ to denote the position at time t; however, this will not be followed.

The state yields the positions of the pursuer and the evader, $x = (p, e)$, which results in the state space $X = R \times R \subset \mathbb{R}^4$. Since the evader position is unknown, the current state is unknown, and I-spaces arise. The observation space Y is a collection of subsets of R. For each $p \in R$, the sensor yields a visibility polygon, $V(p) \subseteq R$ (this is denoted by $y = h(p, e)$ using notation of Section 11.1.1). Consider the history I-state at time t. The initial pursuer position $p(0)$ is given (any position can be chosen arbitrarily, if it is not given), and the evader may lie anywhere in R. The input history $\tilde{u}_t$ can be expressed as the pursuer history $\tilde{p}_t$.[5] Thus, the history I-state is

$$\eta_t = ((p(0), R), \tilde{p}_t, \tilde{y}_t), \tag{12.33}$$

in which $(p(0), R) \subset X$ reflects the initial condition in which $p(0)$ is known, and the evader position $e(0)$ may lie anywhere in R.

Consider the nondeterministic I-space, $\mathcal{I}_{ndet}$. Since the pursuer position is always known, the interesting part of R is the subset in which the evader may lie. Thus, the nondeterministic I-state can be expressed as $X_t(\eta_t) = (p(t), E(\eta_t))$, in which $E(\eta_t)$ is the set of possible evader positions given η_t. As usual for nondeterministic I-states, $E(\eta_t)$ is the smallest set that is consistent with all of the information in η_t.

Consider how $E(\eta_t)$ varies over time. After the first instant of time, $V(p(0))$ is observed, and it is known that the evader lies in $R \setminus V(p(0))$, which is the shadow region (defined in Section 12.3.4) from $p(0)$. As the pursuer moves, $E(\eta_t)$ varies. Suppose you are told that the pursuer is now at position $p(t)$, but you are not yet told η_t. What options seem possible for $E(\eta_t)$? These depend on the history, but the only interesting possibilities are that each shadow component may or may not contain the evader. For some of these components, we may be certain that it does not. For example, consider Figure 12.38. Suppose that the pursuer initially believes that the end of the corridor may contain the evader. If it moves along the smaller closed-loop path, the nondeterministic I-state gradually varies but returns to the same value when the loop is completed. However, if the pursuer traverses the larger loop, it becomes certain upon completing the loop that the corridor does not contain the evader. The dashed line that was crossed in this example may inspire you to think about cell decompositions based on critical boundaries, as in the algorithm in Section 6.3.4. This idea will be pursued shortly to develop a complete algorithm for solving this problem. Before presenting a complete algorithm, however, first consider some interesting examples.

Example 12.7 (When Is a Problem Solvable?) Figure 12.39 shows four similar problems. The evader position is never shown because the problem is solved

[5]To follow the notation of Section 11.4 more closely, the motion model $\dot{p} = u$ can be used, in which u represents the velocity of the pursuer. Nature actions can be used to model the velocity of the evader to obtain $\dot{e}$. By integrating $\dot{p}$ over time, $p(t)$ can be obtained for any t. This means that $\tilde{p}_t$ can be used as a simpler representation of the input history, instead of directly referring to velocities.

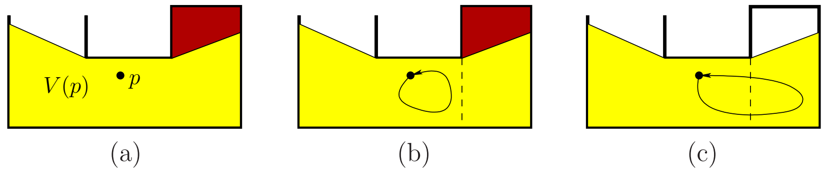


Figure 12.38: (a) Suppose the pursuer comes near the end of a contaminated corridor. (b) If the pursuer moves in a loop path, the nondeterministic I-state gradually changes, but returns to its original value. (c) However, if a critical boundary is crossed, then the nondeterministic I-state fundamentally changes.

by ensuring that no evader could be left hiding. Note that the speed of the pursuer is not relevant to the nondeterministic I-states. Therefore, a solution can be defined by simply showing the pursuer path. The first three examples are straightforward to solve. However, the fourth example does not have a solution because there are at least three distinct hiding places (can you find them?). Let $V(V(p))$ denote the set of all points visible from at least one point in $V(p)$. The condition that prevents the problem from being solved is that there exist three positions, p_1, p_2, p_3, such that $V(V(p_i)) \cap V(V(p_j)) = \emptyset$ for each $i, j \in \{1, 2, 3\}$ with $i \neq j$. As one hiding place is reached, the evader can sneak between the other two. In the worst case, this could result in an endless chase with the evader always eluding discovery. We would like an algorithm that systematically searches $\mathcal{I}_{ndet}$ and determines whether a solution exists. ■

Since one pursuer is incapable of solving some problems, it is tempting to wonder whether two pursuers can solve any problem. The next example gives an interesting sequence of environments that implies that for any positive integer k, there is an environment that requires exactly k pursuers to solve.

Example 12.8 (A Sequence of Hard Problems) Each environment in the sequence shown in Figure 12.40 requires one more pursuer than the previous one [119]. The construction is based on recursively ensuring there are three isolated hiding places, as in the last problem of Figure 12.39. Each time this occurs, another pursuer is needed. The sequence recursively appends three environments that require k pursuers, to obtain a problem that requires $k+1$. An extra pursuer is always needed to guard the junction where the three environments are attached together. The construction is based on the notion of 3-separability, from pursuit-evasion on a graph, which was developed in [219]. ■

The problem can be made more challenging by considering multiply connected environments (environments with holes). A single pursuer cannot solve any of the

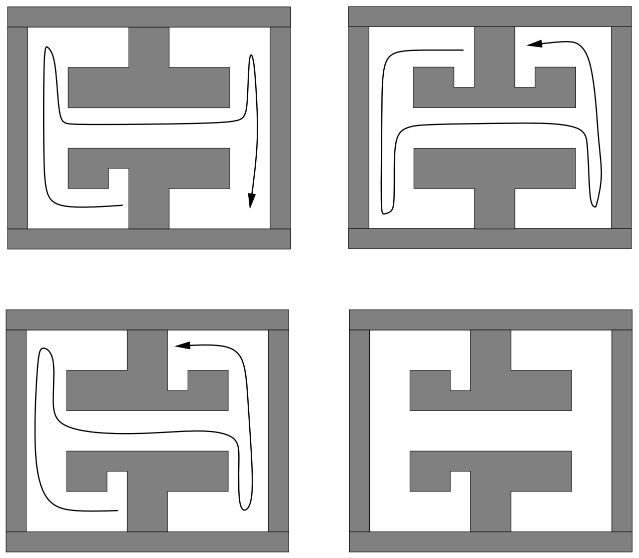


Figure 12.39: Three problems that can be easily solved with one pursuer, and a minor variant for which no solution exists.

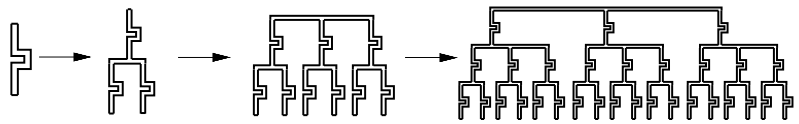


Figure 12.40: Each collection of corridors requires one more pursuer than the one before it because a new pursuer must guard the junction.

these problems. Determining the minimum number of pursuers required to solve such a problem is NP-hard [119].

12.4.2 A Complete Algorithm

Now consider designing a complete algorithm that solves the problem in the case of a single pursuer. To be *complete*, it must find a solution if one exists; otherwise, it correctly reports that no solution is possible. Recall from Figure 12.38 that the nondeterministic I-state changed in an interesting way only after a critical boundary was crossed. The pursuit-evasion problem can be solved by carefully analyzing all of the cases in which these critical changes can occur. It turns out that these are exactly the same cases as considered in Section 12.3.4: crossing inflection rays and bitangent rays. Figure 12.38 is an example of crossing an inflection ray. Figure 12.41 indicates the connection between the gaps of Section 12.3.4 and the parts of the environment that may contain the evader.

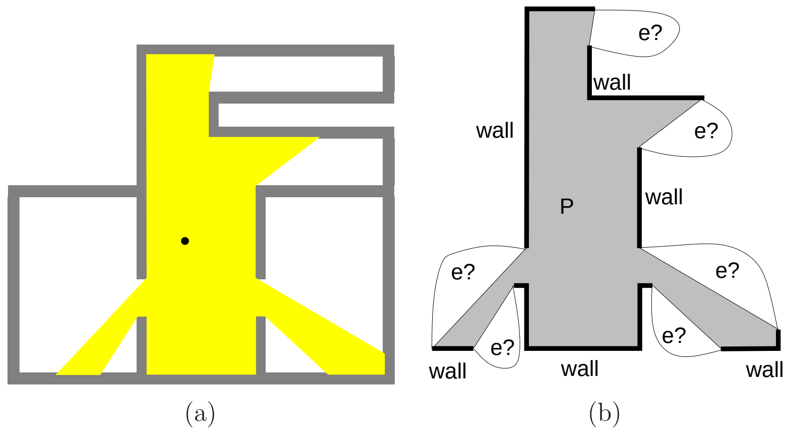


Figure 12.41: Recall Figure 11.15. Beyond each gap is a portion of the environment that may or may not contain the evader.

Recall that the shadow region is the set of all points not visible from some $p(t)$; this is expressed as $R \setminus V(p(t))$. Every critical event changes the number of shadow components. If an inflection ray is crossed, then a shadow component either appears or disappears, depending on the direction. If a bitangent ray is crossed, then either two components merge into one or one component splits into two. To keep track of the nondeterministic I-state, it must be determined whether each component of the shadow region is *cleared*, which means it certainly does not contain the evader, or *contaminated*, which means that it might contain the evader. Initially, all components are labeled as contaminated, and as the pursuer moves, cleared components can emerge. Solving the pursuit-evasion problem amounts to moving the pursuer until all shadow components are cleared. At this point, it is known that there are no places left where the evader could be hiding.

If the pursuer crosses an inflection ray and a new shadow component appears, it must always be labeled as cleared because this is a portion of the environment that was just visible. If the pursuer crosses a bitangent ray and a split occurs, then the labels are distributed across the two components: A contaminated shadow component splits into two contaminated components, and a cleared component splits into two cleared components. If the bitangent ray is crossed in the other direction, resulting in a merge of components, then the situation is more complicated. If one component is cleared and the other is contaminated, then the merged component is contaminated. The merged component may only be labeled as cleared if both of the original components are already cleared. Note that among the four critical cases, only the merge has the potential to undo the work of the pursuer. In other words, it may lead to *recontamination*.

Consider decomposing R into cells based on inflection rays and bitangent rays, as shown in Figure 12.42. These cells have the following *information-conservative*

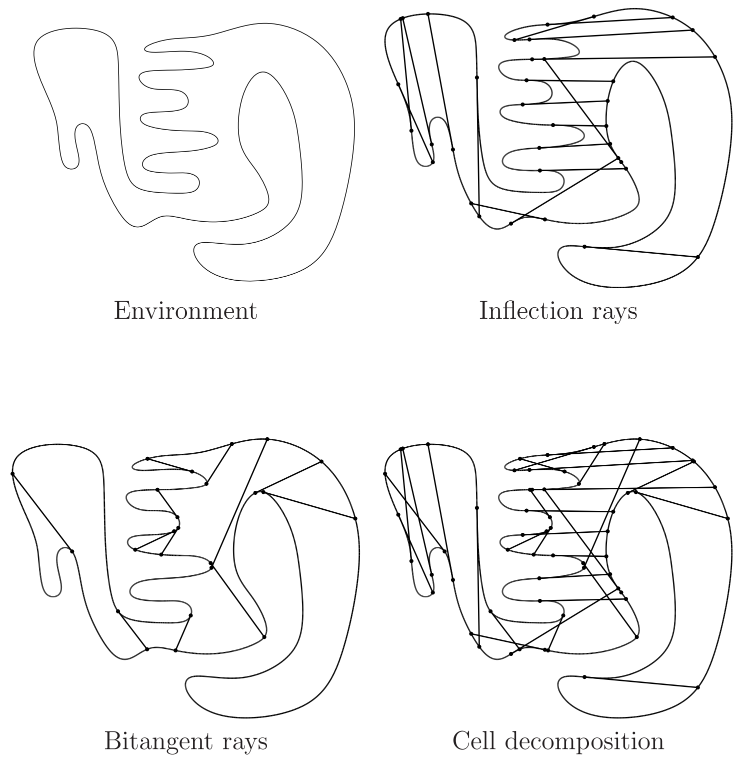


Figure 12.42: The environment is decomposed into cells based on inflections and bitangents, which are the only critical visibility events.

property: If the pursuer travels along any loop path that stays within a 2D cell, then the I-state remains the same upon returning to the start. This implies that the particular path taken by the pursuer through a cell is not important. A solution to the pursuit-evasion problem can be described as a sequence of adjacent 2D cells that must be visited. Due to the information-conservative property, the particular path through a sequence of cells can be chosen arbitrarily.

Searching the cells for a solution is more complicated than searching for paths in Chapter 6 because the search must be conducted in the I-space. The pursuer may visit the same cell in R on different occasions but with different knowledge about which components are cleared and contaminated. A directed graph, $\mathcal{G}_I$, can be constructed as follows. For each 2D cell in R and each possible labeling of shadow components, a vertex is defined in $\mathcal{G}_I$. For example, if the shadow region of a cell has three components, then there are $2^3 = 8$ corresponding vertices in $\mathcal{G}_I$. An edge exists in $\mathcal{G}_I$ between two vertices if: 1) their corresponding cells are adjacent, and 2) the labels of the components are consistent with the changes induced by crossing the boundary between the two cells. The second condition means that the labeling rules for an appear, disappear, split, or merge must be followed. For example, if crossing the boundary causes a split of a contaminated shadow component, then the new components must be labeled contaminated and all other components must retain the same label. Note that $\mathcal{G}_I$ is directed because many motions in the $\mathcal{I}_{ndet}$ are not reversible. For example, if a contaminated region disappears, it cannot reappear as contaminated by reversing the path. Note that the information in this directed graph does not improve monotonically as in the case of lazy discrete localization from Section 12.2.1. In the current setting, information is potentially worse when shadow components merge because contamination can spread.

To search $\mathcal{G}_I$, start with any vertex for which all shadow region components are labeled as contaminated. The particular starting cell is not important. Any of the search algorithms from Section 2.2 may be applied to find a goal vertex, which is any vertex of $\mathcal{G}_I$ for which all shadow components are labeled as cleared. If no such vertices are reachable from the initial state, then the algorithm can correctly declare that no solution exists. If a goal vertex is found, then the path in $\mathcal{G}_I$ gives the sequence of cells that must be visited to solve the problem. The actual path through R is then constructed from the sequence of cells. Some of the cells may not be convex; however, their shape is simple enough that a sophisticated motion planning algorithm is not needed to construct a path that traverses the cell sequence.

The algorithm presented here is conceptually straightforward and performs well in practice; however, its worst-case running time is exponential in the number of inflection rays. Consider a polygonal environment that is expressed with n edges. There can be as many as $O(n)$ inflections and $O(n^2)$ bitangents. The number of cells is bounded by $O(n^3)$ [118]. Unfortunately, $\mathcal{G}_I$ has an exponential number of vertices because there can be as many as $O(n)$ shadow components, and there are 2^n possible labelings if there are n components. Note that $\mathcal{G}_I$ does not need to be computed prior to the search. It can be revealed incrementally during the planning process. The most efficient complete algorithm, which is more complicated, solves the pursuit-evasion problem in time $O(n^2)$ and was derived by first proving that any problem that can be solved by a pursuer using the visibility polygon can be solved by a pursuer that uses only two beams of light [216]. This simplifies $V(p(t))$ from a 2D region in R to two rotatable rays that emanate from $p(t)$ and dramatically reduces the complexity of the I-space.

12.4.3 Other Variations

Numerous variations of the pursuit-evasion problem presented in this section can be considered. The problem becomes much more difficult if there are multiple pursuers. A cell decomposition can be made based on changing shadow components; however, some of the cell boundaries are algebraic surfaces due to complicated interactions between the visibility polygons of different pursuers. Thus, it is difficult to implement a complete algorithm. On the other hand, straightforward heuristics can be used to guide multiple pursuers. A single pursuer can use the complete algorithm described in this section. When this pursuer fails, it can move to some part of the environment and then wait while a second pursuer applies the complete single-pursuer algorithm on each shadow component. This idea can be applied recursively for any number of robots.

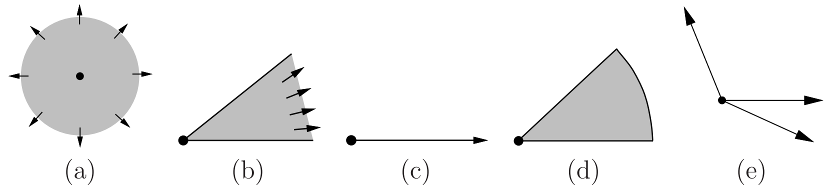


Figure 12.43: Several evader detection models: (a) omnidirectional sensing with unlimited distance; (b) visibility with a limited field of view; (c) a single visibility ray that is capable of rotating; (d) limited distance and a rotating viewing cone, which corresponds closely to a camera model; and (e) three visibility rays that are capable of rotating.

The problem can be made more complicated by placing a velocity bound on the evader. Even though this makes the pursuer more powerful, it is more difficult to design a complete algorithm that correctly exploits this additional information. No complete algorithms currently exist for this case.

Figure 12.43 shows several alternative detection models that yield different definitions of $V(p(t))$. Each requires different pursuit-evasion algorithms because the structure of the I-space varies dramatically across different sensing models. For example, using the model in Figure 12.43c, a single pursuer is required to move along the ∂X. Once it moves into the interior, the shadow region always becomes a single connected component. This model is sometimes referred to as a *flashlight*. If there are two flashlights, then one flashlight may move into the interior while the other protects previous work. The case of limited depth, as shown in Figure 12.43, is very realistic in practice, but unfortunately it is the most challenging. The number of required pursuers generally depends on metric properties of the environment, such as its minimum "thickness." The method presented in this section was extended to the case of a limited field of view in [110]; critical curves are obtained that are similar to those in Section 6.3.4. See the literature overview at the end of the chapter for more related material.

12.5 Manipulation Planning with Sensing Uncertainty

One of the richest sources of interesting I-spaces is manipulation planning. As robots interact with obstacles or objects in the world, the burden of estimating the state becomes greater. The classical way to address this problem is to highly restrict the way in which the robot can interact with obstacles. Within the manipulation planning framework of Section 7.3.2, this means that a robot must grasp and carry objects to their desired destinations. Any object must be lying in a stable configuration upon grasping, and it must be returned to a stable configuration after grasping.

As the assumptions on the classical manipulation planning framework are lifted, it becomes more difficult to predict how the robot and other bodies will behave. This immediately leads to the challenges of uncertainty in predictability, which was the basis of Chapter 10. The next problem is to design sensors that enable plans to be achieved in spite of this uncertainty. For each sensing model, an I-space arises.

Section 12.5.1 covers the preimage planning framework [90, 177], under which many interesting issues covered in Chapters 10 and 11 are addressed for a specific manipulation planning problem. I-states, forward projections, backprojections, and termination actions were characterized in this context. Furthermore, several algorithmic complexity results regarding planning under uncertainty have been proved within this framework.

Section 12.5.2 covers methods that clearly illustrate the power of reasoning directly in terms of the I-space. The philosophy is to allow nonprehensile forms of manipulation (e.g., pushing, squeezing, throwing) and to design simple sensors, or even to avoid sensing altogether. This dramatically reduces the I-space while still allowing feasible plans to exist. This contradicts the intuition that more information is better. Using less information leads to greater uncertainty in the state, but this is not important in some problems. It is only important is that the I-space becomes simpler.

12.5.1 Preimage Planning

The *preimage planning* framework (or *LMT framework*, named after its developers, Lozano-Pérez, Mason, and Taylor) was developed as a general way to perform manipulation planning under uncertainty [90, 177]. Although the concepts apply to general configuration spaces, they will be covered here for the case in which $\mathcal{C} = \mathbb{R}^2$ and $\mathcal{C}_{obs}$ is polygonal. This is a common assumption throughout most of the work done within this framework. This could correspond to a simplified model of a robot hand that translates in $\mathcal{W} = \mathbb{R}^2$, while possibly carrying a part. A popular illustrative task is the *peg-in-hole problem*, in which the part is a peg that must be inserted into a hole that is slightly larger. This operation

is frequently performed as manufacturing robots assemble products. Using the configuration space representation of Section 4.3.2, the robot becomes a point moving in $\mathbb{R}^2$ among polygonal obstacles.

The distinctive features of the models used in preimage planning are as follows:

1. The robot can execute *compliant motions*, which means that it can slide along the boundary of $\mathcal{C}_{obs}$. This differs from the usual requirement in Part II that the robot must avoid obstacles.

2. There is nondeterministic uncertainty in prediction. An action determines a motion direction, but nature determines how much error will occur during execution. A bounded error model is assumed.

3. There is nondeterministic uncertainty in sensing, and the true state cannot be reliably estimated.

4. The goal region is usually an edge of $\mathcal{C}_{obs}$, but it may more generally be any subset of $\mathrm{cl}(\mathcal{C}_{free})$, the closure of $\mathcal{C}_{free}$.

5. A hierarchical planning model is used, in which the robot is issued a sequence of *motion commands*, each of which is terminated by applying u_T based on the I-state.

Each of these will now be explained in more detail.

Compliant motions It will be seen shortly that the possibility of executing *compliant motions* is crucial for reducing uncertainty in the robot position. Let $\mathcal{C}_{con}$ denote the obstacle boundary, $\partial\mathcal{C}_{obs}$ (also, $\mathcal{C}_{con} = \partial\mathcal{C}_{free}$). A model of robot motion while $q \in \mathcal{C}_{con}$ needs to be formulated. In general, this is complicated by friction. A simple *Coulomb friction* model is assumed here; see [189] for more details on modeling friction in the context of manipulation planning. Suppose that the net force F is applied by a robot at some $q \in \mathcal{C}_{con}$. The force could be maintained by using the *generalized damper model* of robot control [289].

The resulting motion is characterized using a *friction cone*, as shown in Figure 12.44a. A basic principle of Newtonian mechanics is that the obstacle applies a reaction force (it may be helpful to look ahead to Section 13.3, which introduces mechanics). If F points into the surface and is normal to it, then the reaction force provided by the obstacle will cancel F, and there will be no motion. If F is not perpendicular to the surface, then sliding may occur. At one extreme, F may be parallel to the surface. In this case, it must slide along the boundary. In general, F can be decomposed into parallel and perpendicular components. If the parallel component is too small relative to the perpendicular component, then the robot becomes stuck. The friction cone shown in Figure 12.44a indicates precisely the conditions under which motion occurs. The parameter α captures the amount of friction (more friction leads to larger α). Figure 12.44b indicates the behaviors

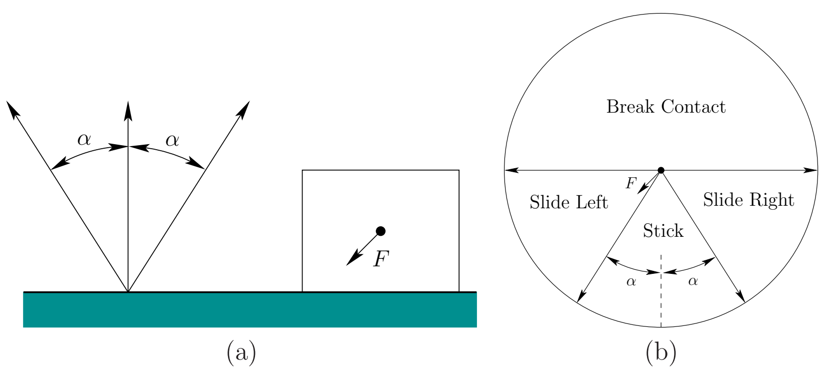


Figure 12.44: The compliant motion model. If a force F is applied by the robot at $q \in \mathcal{C}_{con}$, then it moves along the boundary only if $-F$ points outside of the friction cone.

that occur for various directions of F. The diagram is obtained by inverting the friction cone. If F points into the bottom region, then *sticking* occurs, which means that the robot cannot move. If F points away from the obstacle boundary, then contact is broken (this is reasonable, unless the boundary is sticky). For the remaining two cases, the robot slides along the boundary.

Sources of uncertainty Nature interferes with both the configuration transitions and with the sensor. Let $U = [0, 2\pi)$, which indicates the direction in $\mathbb{R}^2$ that the robot is commanded to head. Nature interferes with this command, and the actual direction lies within an interval of $\mathbb{S}^1$. As shown in Figure 12.45a, the forward projection (recall from Section 10.1.2) for a fixed action $u \in U$ yields a cone of possible future configurations. (A precise specification of the motion model is given using differential equations in Example 13.15.) The sensing model, shown in Figure 12.45b, was already given in Section 11.5.1. The nature sensing actions form a disc given by (11.67), and $y = q + \psi$, in which q is the true configuration, ψ is the nature sensing action, and y is the observation. The result appears in Figure 11.11.

Goal region Since contact with the obstacle is allowed, the goal region can be defined to include edges of $\mathcal{C}_{obs}$ in addition to points in $\mathcal{C}_{free}$. Most often, a single edge of $\mathcal{C}_{obs}$ is chosen as the goal region.

Motion commands The planning problem can now be described. It may be tempting to express the model using continuous time, as opposed to discrete stages. This is a viable approach, but leads to planning under differential constraints, which is the topic of Part IV and is considerably more complicated. In

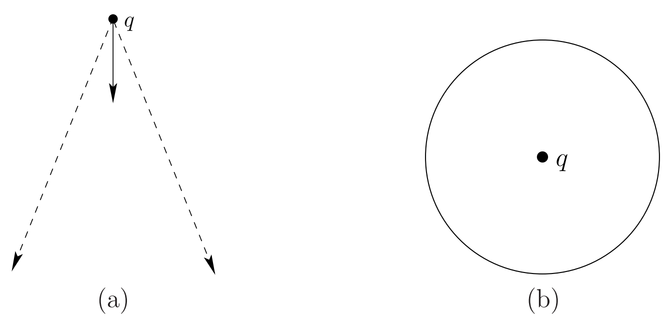


Figure 12.45: Nature interferes with both the configuration transitions and the sensor observations.

the preimage-planning framework, a hierarchical approach is taken. A restricted kind of plan called a *motion command*, μ, will be defined, and the goal is achieved by constructing a sequence of motion commands. This has the effect of converting the continuous-time decision-making problem into a planning problem that involves discrete stages. Each time a motion command is applied, the robot must apply a termination action to end it. At that point another motion command can be issued. Thus, imagine that a high-level module issues motion commands, and a low-level module executes each until a termination condition is met.

For some action $u \in U$, let $M_u = \{u, u_T\}$, in which u_T is the termination action. A *motion command* is a feedback plan, $\mu : \mathcal{I}_{hist} \rightarrow M_u$, in which $\mathcal{I}_{hist}$ is the standard history I-space, based on initial conditions, the action history, and the sensing history. The motion command is executed over continuous time. At $t = 0$, $\mu(\eta_0) = u$. Using a history I-state η gathered during execution, the motion command will eventually yield $\mu(\eta) = u_T$, which terminates it. If the goal was not achieved, then the high-level module can apply another motion command.

Preimages Now consider how to construct motion commands. Using the hierarchical approach, the main task of terminating in the goal region can be decomposed into achieving intermediate subgoals. The *preimage* $P(\mu, G)$ of a motion command μ and subgoal $G \subset \text{cl}(\mathcal{C}_{free})$ is the set of all history I-states from which μ is guaranteed to be achieved in spite of all interference from nature. Each motion command must recognize that the subgoal has been achieved so that it can apply its termination action. Once a subgoal is achieved, the resulting history I-state must lie within the required set of history I-states for the next motion command in the plan. Let $\mathcal{M}$ denote the set of all allowable motion commands that can be defined. This can actually be considered as an action space for the high-level module.

Planning with motion commands A high-level open-loop plan,[6]

$$\pi = (\mu_1, \mu_2, \ldots, \mu_k), \tag{12.34}$$

can be constructed, which is a sequence of k motion commands. Although the precise path executed by the robot is unpredictable, the sequence of motion commands is assumed to be predictable. Each motion command μ_i for $1 < i < k$ must terminate with an I-state $\eta \in P(\mu_{i+1}, G_{i+1})$. The preimage of μ_1 must include η_0, the initial I-state. The goal is achieved by the last motion command, μ_k.

More generally, the particular motion command chosen need not be predictable, and may depend on the I-state during execution. In this case, the high-level feedback plan $\pi : \mathcal{I}_{hist} \rightarrow \mathcal{M}$ can be developed, in which a motion command $\mu = \pi(\eta)$ is chosen based on the history I-state η that results after the previous motion command terminates. Such variations are covered in [75, 90, 158].

The high-level planning problem can be solved using discrete planning algorithms from Chapters 2 and 10. The most popular method within the preimage planning framework is to perform a backward search from the goal. Although this sounds simple enough, the set of possible motion commands is infinite, and it is difficult to sample μ in a way that leads to completeness. Another complication is that termination is based on the history I-state. Planning is therefore quite challenging. It was even shown in [90], by a reduction from the Turing machine halting problem [260], that the preimage in general is uncomputable by any algorithm. It was shown in [202] that the 3D version of preimage planning, in which the obstacles are polyhedral, is PSPACE-hard. It was then shown in [47] that it is even NEXPTIME-hard.[7]

Backprojections Erdmann proposed a practical way to compute effective motion commands by separating the reachability and recognizability issues [90, 91]. Reachability refers to characterizing the set of points that are guaranteed to be reachable. Recognizability refers to knowing that the subgoal has been reached based on the history I-state. Another way to interpret the separation is that the effects of nature on the configuration transitions is separated from the effects of nature on sensing.

For reachability analysis, the sensing uncertainty is neglected. The notions of forward projections and backprojections from Section 10.1.2 can then be used. The only difference here is that they are applied to continuous spaces and motion commands (instead of u). Let S denote a subset of $\text{cl}(\mathcal{C}_{free})$. Both weak backprojections, $\text{WB}(S, \mu)$, and strong backprojections, $\text{SB}(S, \mu)$, can be defined. Furthermore, *nondirectional backprojections* [76], $\text{WB}(S)$ and $\text{SB}(S)$, can be defined, which are analogous to (10.25) and (10.26), respectively.

[6]Note that this open-loop plan is composed of closed-loop motion commands. This is perfectly acceptable using hierarchical modeling.

[7]NEXPTIME is the complexity class of all problems that can be solved in nondeterministic exponential time. This is beyond the complexity classes shown in Figure 6.40.

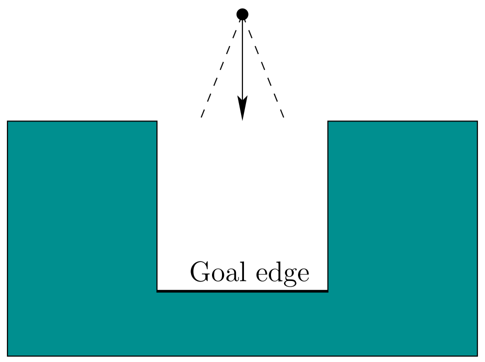


Figure 12.46: A simple example that resembles the peg-in-hole problem.

Figure 12.46 shows a simple problem in which the task is to reach a goal edge with a motion command that points downward. This is inspired by the peg-in-hole problem. Figure 12.47 illustrates several backprojections from the goal region for the problem in Figure 12.46. The action is $u = 3\pi/2$; however, the actual motion lies within the shown cone due to nature. First suppose that contact with the obstacle is not allowed, except at the goal region. The strong backprojection is given in Figure 12.47a. Starting from any point in the triangular region, the goal is guaranteed to be reached in spite of nature. The weak backprojection is the unbounded region shown in Figure 12.47b. This indicates configurations from which it is *possible* to reach the goal. The weak backprojection will not be considered further because it is important here to *guarantee* that the goal is reached. This is accomplished by the strong backprojection. From here onward, it will be assumed that *backprojection* by default means a strong backprojection. Using weak backprojections, it is possible to develop an alternative framework of *error detection and recovery* (EDR), which was introduced by Donald in [75].

Now assume that compliant motions are possible along the obstacle boundary. This has the effect of enlarging the backprojections. Suppose for simplicity that there is no friction ($\alpha = 0$ in Figure 12.44a). The backprojection is shown in Figure 12.47c. As the robot comes into contact with the side walls, it slides down until the goal is reached. It is not important to keep track of the exact configuration while this occurs. This illustrates the power of compliant motions in reducing uncertainty. This point will be pursued further in Section 12.5.2. Figure 12.47d shows the backprojection for a different motion command.

Now consider computing backprojections in a more general setting. The backprojection can be defined from any subset of $\text{cl}(\mathcal{C}_{free})$ and may allow a friction cone with parameter α. To be included in a backprojection, points from which sticking is possible must be avoided. Note that sticking is possible even if $\alpha = 0$. For example, in Figure 12.46, nature may allow the motion to be exactly perpendicular to the obstacle boundary. In this case, sticking occurs on horizontal edges

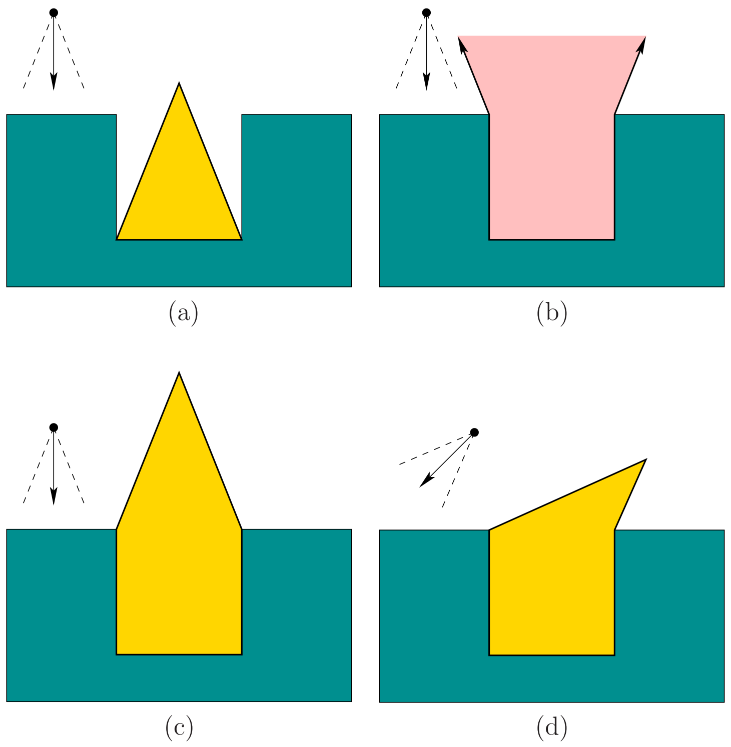


Figure 12.47: Several backprojections are shown for the peg-in-hole problem.

because there is no tangential motion. In general, it must be determined whether sticking is *possible* at each edge and vertex of $\mathcal{C}_{obs}$. Possible sticking from an edge depends on u, α, and the maximum directional error contributed by nature. The robot can become stuck at a vertex if it is possible to become stuck at either incident edge.

Computing backprojections Many algorithms have been developed to compute backprojections. The first algorithm was given in [90, 91]. Assume that the goal region is one or more segments contained in edges of $\mathcal{C}_{con}$. The algorithm proceeds for a fixed motion command, μ, which is based on a direction $u \in U$ as follows:

1. Mark every obstacle vertex at which sticking is possible. Also mark any point on the boundary of the goal region if it is possible to slide away from the goal.
2. For every marked vertex, extend two rays with directions based on the maximum possible deviations allowed by nature when executing u. This inverts the cone shown in Figure 12.45a. The extended rays are shown in Figure 12.48 for the frictionless case ($\alpha = 0$).

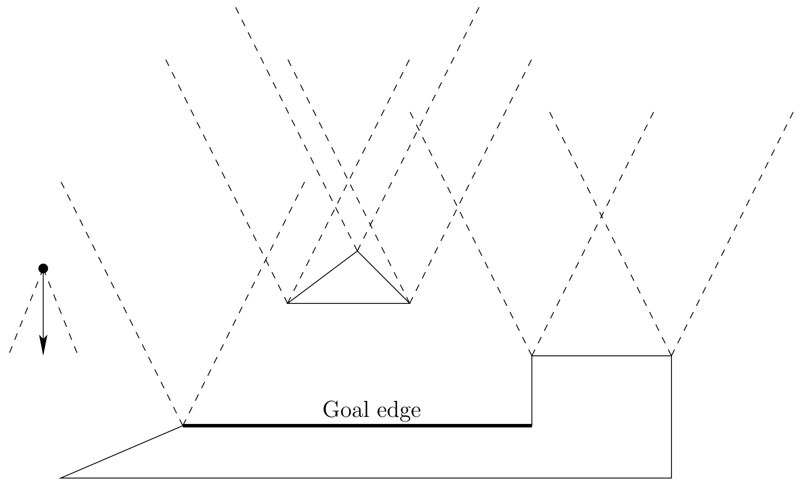


Figure 12.48: Erdmann's backprojection algorithm traces out the boundary after constructing cones based on friction.

3. Starting at every goal edge, trace out the boundary of the backprojection region. Every edge encountered defines a half-plane of configurations from which the robot is guaranteed to move into. In Figure 12.48, this corresponds to being below a ray. When tracing out the backprojection boundary, the direction at each intersection vertex is determined based on including the points in the half-plane.

The resulting backprojection is shown in Figure 12.49. A more general algorithm that applies to goal regions that include polygonal regions in $\mathcal{C}_{free}$ was given in [76] (some details are also covered in [158]). It uses the plane-sweep principle (presented in Section 6.2.2) to yield an algorithm that computes the backprojection in time $O(n \lg n)$, in which n is the number of edges used to define $\mathcal{C}_{obs}$. The backprojection itself has no more than $O(n)$ edges. Algorithms for computing nondirectional backprojections are given in [39, 76]. One difficulty in this case is that the backprojection boundary may be quite complicated. An incremental algorithm for computing a nondirectional backprojection of size $O(n^2)$ in time $O(n^2 \lg n)$ is given in [39].

Once an algorithm that computes backprojections has been obtained, it needs to be adapted to compute preimages. Using the sensing model shown in Figure 12.45b, a preimage can be obtained by shrinking the subgoal region G. Let ϵ denote the radius of the ball in Figure 12.45b. Let $G' \subset G$ denote a subset of the

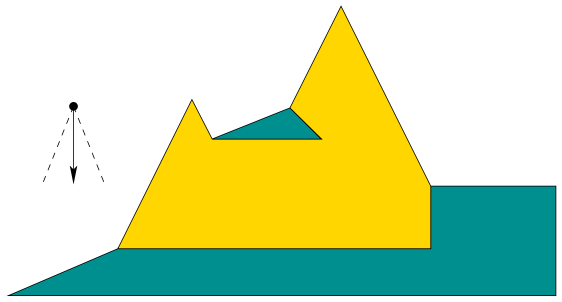


Figure 12.49: The computed backprojection. Sliding is guaranteed from the steeper edge of the triangle; hence, it is included in the backprojection. From the other top edge, sticking is possible.

subgoal in which a strip of thickness ϵ has been removed. If the sensor returns $y \in G'$, then it is guaranteed that $q \in G$. This yields a method of obtaining preimages by shrinking the subgoals. If ϵ is too large, however, this may fail to yield a successful plan, even though one exists.

The high-level plan can be found by performing a backward search that computes backprojections from the goal region (reduced by ϵ). There is still the difficulty of $\mathcal{M}$ being too large, which controls the branching factor in the search. One possibility is to compute nondirectional backprojections. Another possibility is to discretize $\mathcal{M}$. For example, in [158, 159], $\mathcal{M}$ is reduced to four principle directions, and plans are computed for complicated environments by using sticking edges as subgoals. Using discretization, however, it becomes more difficult to ensure the completeness of the planning algorithm.

The preimage planning framework may seem to apply only to a very specific model, but it can be extended and adapted to a much more general setting. It was extended to semi-algebraic obstacle models in [48], which gives a planning method that runs in time doubly exponential in the C-space dimension (based on cylindrical algebraic decomposition, which was covered in Section 6.4.2). In [43], probabilistic backprojections were introduced by assigning a uniform probability density function to the nature action spaces considered in this section. This was in turn generalized further to define backprojections and preimages as the level sets of optimal cost-to-go functions in [161, 166]. Dynamic programming methods can then be applied to compute plans.

12.5.2 Nonprehensile Manipulation

Manipulation by grasping is very restrictive. People manipulate objects in many interesting ways that do not involve grasping. Objects may be pushed, flipped, thrown, squeezed, twirled, smacked, blown, and so on. A classic example from the kitchen is flipping a pancake over by a flick of the wrist while holding the skillet.

These are all examples of *nonprehensile manipulation*, which means manipulation without grasping.

The temptation to make robots grasp objects arises from the obsession with estimating and controlling the state. This task is more daunting for nonprehensile manipulation because there are times at which the object appears to be out of direct control. This leads to greater uncertainty in predictability and a larger sensing burden. By planning in the I-space, however, it may be possible to avoid all of these problems. Several works have emerged which show that manipulation goals can be achieved with little or no sensing at all. This leads to a form of *minimalism* [49, 97, 189], in which the sensors are designed in a way that simplifies the I-space, as opposed to worrying about accurate estimation. The search for minimalist robotic systems is completely aligned with trying to find derived I-spaces that are as small as possible, as mentioned in Section 11.2.1. Sensing systems should be simple, yet still able to achieve the task. Preferably, completeness should not be lost. Most work in this area is concerned primarily with finding feasible solutions, as opposed to optimal solutions. This enables further simplifications of the I-space.

This section gives an example that represents an extreme version of this minimalism. A *sensorless manipulation* system is developed. At first this may seem absurd. From the forward projections in Section 10.1.2, it may seem that uncertainty can only grow if nature causes uncertainty in the configuration transitions and there are no sensors. To counter the intuition, compliant motions have the ability to reduce uncertainty. This is consistent with the discussion in Section 11.5.4. Simply knowing that some motion commands have been successfully applied may reduce the amount of uncertainty. In an early demonstration of sensorless manipulation, it was shown that an Allen wrench (L-shaped wrench) resting in a tray can be placed into a known orientation by simply tilting the tray in a few directions [97]. The same orientation is achieved in the end, regardless of the initial wrench configuration. Also, no sensors are needed. This can be considered as a more complicated extension of the ball rolling in a tray that was shown in Figure 11.29. This is also an example of compliant motions, as shown in Figure 12.44; however, in the present setting F is caused by gravity.

Squeezing parts Another example of sensorless manipulation will now be described, which was developed by Goldberg and Mason in [112, 113, 114]; see also [189]. A Java implementation of the algorithm appears in [36]. Suppose that convex, polygonal parts arrive individually along a conveyor belt in a factory. They are to be used in an assembly operation and need to be placed into a given orientation. Figure 12.50 shows a top view of a *parallel-jaw gripper*. The robot can perform a squeeze operation by bringing the jaws together. Figure 12.50a shows the part before squeezing, and Figure 12.50b shows it afterward. A simple model is assumed for the mechanics. The jaws move at constant velocity toward each other, and it is assumed that they move slowly enough so that dynamics can be

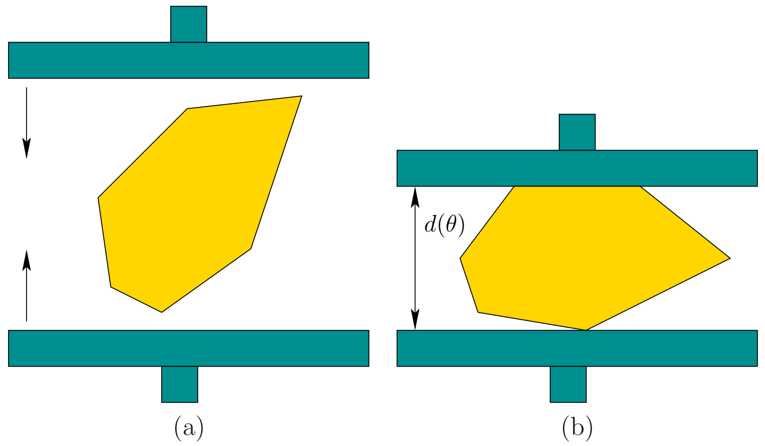


Figure 12.50: A parallel-jaw gripper can orient a part without using sensors.

neglected. To help slide the part into place, one of the jaws may be considered as a frictionless contact (this is a real device; see [49]). The robot can perform a squeeze operation at any orientation in $[0, 2\pi)$ (actually, only $[0, \pi)$ is needed due to symmetry). Let $U = [0, 2\pi)$ denote the set of all squeezing actions. Each squeezing action terminates on its own after the part can be squeezed no further (without crushing the part).

The planning problem can be modeled as a game against nature. The initial orientation, $x \in [0, 2\pi)$, of the part is chosen by nature and is unknown. The state space is $\mathbb{S}^1$. For a given part, the task is to design a sequence,

$$\pi = (u_1, u_2, \ldots, u_n), \tag{12.35}$$

of squeeze operations that leads to a known orientation for the part, regardless of its initial state. Note that there is no specific requirement on the final state. After i motion commands have terminated, the history I-state is the sequence

$$\eta = (u_1, u_2, \ldots, u_i) \tag{12.36}$$

of squeezes applied so far. The nondeterministic I-space $\mathcal{I}_{ndet}$ will now be used. The requirement can be stated as obtaining a singleton, nondeterministic I-state (includes only one possible orientation). If the part has symmetries, then the task is instead to determine a single symmetry class (which includes only a finite number of orientations)

Consider how a part in an unknown orientation behaves. Due to rotational symmetry, it will be convenient to describe the effect of a squeeze operation based on the relative angle between the part and the robot. Therefore, let $\alpha = u - x$, assuming arithmetic modulo 2π. Initially, α may assume any value in $[0, 2\pi)$. It

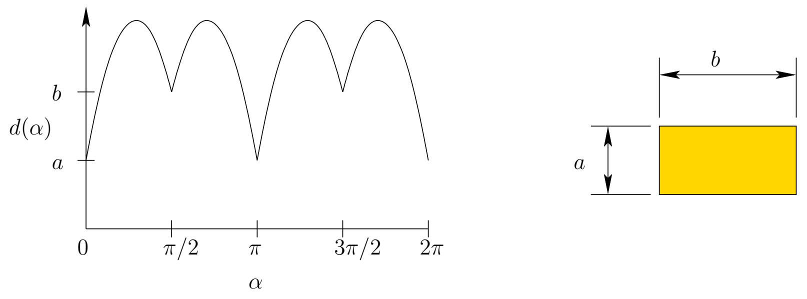


Figure 12.51: The diameter function for a rectangle.

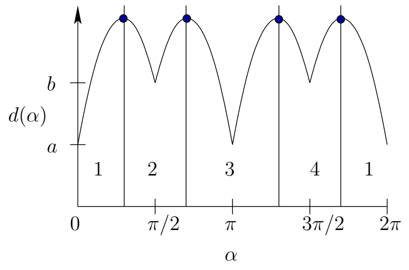


Figure 12.52: There are four regions of attraction, each of which represents an interval of orientations.

turns out that after one squeeze, α is always forced into one of a finite number of values. This can be explained by representing the *diameter function* $d(\alpha)$, which indicates the maximum thickness that can be obtained by taking a slice of the part at orientation α. Figure 12.51 shows the slice for a rectangle. The local minima of the distance function indicate orientations at which the part will stabilize as shown in Figure 12.50b. As the part changes its orientation during the squeeze operation, the α value changes in a way that gradually decreases $d(\alpha)$. Thus, $[0, 2\pi)$ can be divided into regions of attraction, as shown in Figure 12.52. These behave much like the funnels in Section 8.5.1.

The critical observation to solve the problem without sensors is that with each squeeze the uncertainty can grow no worse, and is usually reduced. Assume u is fixed. For the state transition equation $x' = f(x, u)$, the same x' will be produced for an interval of values for x. Due to rotational symmetry, it is best to express this in terms of α. Let $s(\alpha)$ denote relative orientation obtained after a squeeze. Since α is a function of x and u, this can be expressed as a *squeeze function*, $s : \mathbb{S}^1 \rightarrow \mathbb{S}^1$, defined as

$$s(\alpha) = f(x, u) - u. \tag{12.37}$$

The forward projection with respect to an interval, A, of α values can also be defined:

$$S(A) = \bigcup_{\alpha \in A} s(\alpha). \tag{12.38}$$

Any interval $A \subset [0, 2\pi)$ can be interpreted as a nondeterministic I-state, based on the history of squeezes that have been performed. It is defined, however, with respect to relative orientations, instead of the original states. The algorithms discussed in Section 12.1.2 can be applied to $\mathcal{I}_{ndet}$. A backward search algorithm is given in [113] that starts with a singleton, nondeterministic I-state. The planning proceeds by performing a backward search on $\mathcal{I}_{ndet}$. In each iteration, the interval, A, of possible relative orientations increases until eventually all of $\mathbb{S}^1$ is reached (or the period of symmetry, if symmetries exist).

The algorithm is greedy in the sense that it attempts to force A to be as large as possible in every step. Note from Figure 12.52 that the regions of attraction are maximal at the minima of the diameter function. Therefore, only the minima values are worth considering as choices for α. Let B denote the preimage of the function s. In the first step, the algorithm finds the α for which $B(\alpha)$ is largest (in terms of length in $\mathbb{S}^1$). Let α_0 denote this relative orientation, and let $A_0 = B(\alpha_0)$. For each subsequent iteration, let A_i denote the largest interval in $[0, 2\pi)$ that satisfies

$$|S(A_{i-1})| < |A_i|, \tag{12.39}$$

in which $|\cdot|$ denotes interval length. This implies that there exists a squeeze operation for which any relative orientation in $S(A_{i-1})$ can be forced into A_i by a single squeeze. This iteration is repeated, generating A_{-1}, A_{-2}, and so on, until the condition in (12.39) can no longer be satisfied. It was shown in [113] that for any polygonal part, the A_i intervals increase until all of $\mathbb{S}^1$ (or the period of symmetry) is obtained.

Suppose that the sequence $(A_{-k}, \ldots, A_0)$ has been computed. This must be transformed into a plan that is expressed in terms of a fixed coordinate frame for the robot. The k-step action sequence $(u_1, \ldots, u_k)$ is recovered from

$$u_i = s(\beta_{i-1}) - a_i - \tfrac{1}{2}(|A_{i-k}| - |S(A_{i-k-1})|) + u_{i-1} \tag{12.40}$$

and $u_{-k} = 0$ [113]. Each a_i in (12.40) is the left endpoint of A_i. There is some freedom of choice in the alignment, and the third term in (12.40) selects actions in the middle to improve robustness with respect to orientation errors. By exploiting a proof in [52] that no more than $O(n)$ squeeze operations are needed for a part with n edges, the complete algorithm runs in time $O(n^2)$.

Example 12.9 (Squeezing a Rectangle) Figure 12.53 shows a simple example of a plan that requires two squeezes to orient the rectangular part when placed in any initial orientation. Four different executions of the plan are shown, one in each column. After the first squeeze, the part orientation is a multiple of $\pi/2$. After the second squeeze, the orientation is known. Even though the execution looks different every time, no feedback is necessary because the I-state contains

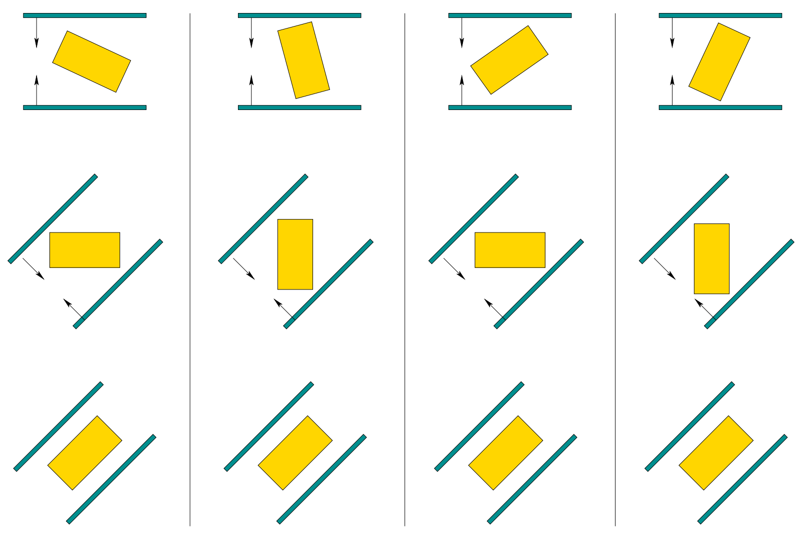


Figure 12.53: A two-step squeeze plan [113].

no sensor information. ■

Further Reading

The material from this chapter could easily be expanded into an entire book on planning under sensing uncertainty. Several key topics were covered, but numerous others remain. An incomplete set of suggestions for further reading is given here.

Since Section 12.1 involved converting the I-space into an ordinary state space, many methods and references in Chapter 10 are applicable. For POMDPs, a substantial body of work has been developed in operations research and stochastic control theory [151, 175, 194, 264] and more recently in artificial intelligence [136, 173, 174, 203, 218, 226, 229, 230, 238, 299, 300]. Many of these algorithms compress or approximate $\mathcal{I}_{prob}$, possibly yielding nonoptimal solutions, but handling problems that involve dozens of states.

Localization, the subject of Section 12.2, is one of the most fundamental problems in robotics; therefore, there are hundreds of related references. Localization in a graph has been considered [84, 102]. The combinatorial localization presentation was based on [85, 120]. Ambiguities due to symmetry also appeared in [14]. Combinatorial localization with very little sensing is presented in [206]. For further reading on probabilistic localization, see [4, 64, 122, 124, 133, 135, 149, 171, 172, 207, 235, 256, 257, 288]. In [275, 276], localization uncertainty is expressed in terms of a sensor-uncertainty field, which is a derived I-space.

Section 12.3 was synthesized from many sources. For more on the maze searching method from Section 12.3.1 and its extension to exploring a graph, see [35]. The issue of distinguishability and pebbles arises again in [19, 79, 80, 181, 241, 281]. For more on competitive ratios and combinatorial approaches to on-line navigation, see [34, 66, 72, 100, 108, 141, 144, 185, 214, 233].

For more on Stentz's algorithm and related work, see [146, 267]. A multi-resolution approach to terrain exploration appears in [211]. For material on bug algorithms, see [140, 154, 160, 179, 180, 231, 254]. Related sensor-based planning work based on generalized Voronoi diagrams appears in [54, 55]; also related is [236]. Gap navigation trees were introduced in [280, 281, 282]. For other work on minimal mapping, see [132, 234, 252]. Landmark-based navigation is considered in [107, 159, 255].

There is a vast body of literature on probabilistic methods for mapping and localization, much of which is referred to as SLAM [279]; see also [50, 56, 74, 196, 217, 296]. One of the earliest works is [263]. An early application of dynamic programming in this context appears in [157]. A well-known demonstration of SLAM techniques is described in [46]. For an introduction to the EM algorithm, see [29]; its convergence is addressed in [71, 293, 294]. For more on mobile robotics in general, see [37, 83].

The presentation of Section 12.4 was based mainly on [119, 167]. Pursuit-evasion problems in general were first studied in differential game theory [9, 123, 129]. Pursuit-evasion in a graph was introduced in [219], and related theoretical analysis appears in [28, 155, 195]. Visibility-based pursuit-evasion was introduced in [273], and the first complete algorithm appeared in [167]. An algorithm that runs in $O(n^2)$ for a single pursuer in a simple polygon was given in [216]. Variations that consider curved environments, beams of light, and other considerations appear in [53, 63, 87, 164, 170, 204, 258, 259, 272, 274, 295]. Pursuit-evasion in three dimensions is discussed in [169]. For versions that involve minimal sensing and no prior given map, see [121, 138, 231, 241, 298]. The problem of visually tracking a moving target both with [17, 116, 117, 163, 198, 199] and without [99, 127, 249] obstacles is closely related to pursuit-evasion. For a survey of combinatorial algorithms for computing visibility information, see [209]. Art gallery and sensor placement problems are also related [40, 208, 253]. The bitangent events also arise in the visibility complex [228] and in aspect graphs [224], which are related visibility-based data structures.

Section 12.5 was inspired mostly by the works in [76, 90, 97, 114, 177, 290]. Many works are surveyed in [189]. A probabilistic version of preimage planning was considered in [44, 45, 166]. Visual preimages are considered in [104]. Careful analysis of manipulation uncertainty appears in [41, 42]. For more on preimage planning, see [158, 159]. The error detection and recovery (EDR) framework uses many preimage planning ideas but allows more problems to be solved by permitting fixable errors to occur during execution [75, 77, 78]. Compliant motions are also considered in [39, 76, 134, 186, 188, 221]. The effects of friction in the C-space are studied in [95]. For more work on orienting parts, see [49, 98, 112, 113, 232, 291]. For more forms of nonprehensile manipulation, see [1, 2, 32, 96, 182, 183, 269]. A humorous paper, which introduces the concept of the "principle of virtual dirt," is [187]; the idea later appears in [240] and in the Roomba autonomous vacuum cleaner from the iRobot Corporation.

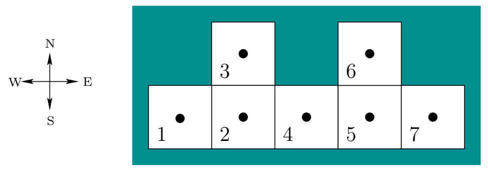


Figure 12.54: An environment for grid-based localization.

Exercises

1. For the environment in Figure 12.1a, give the nondeterministic I-states for the action sequence (L, L, F, B, F, R, F, F), if the initial state is the robot in position 3 facing north and the initial I-state is $\eta_0 = X$.

2. Describe how to apply the algorithm from Figure 10.6 to design an information-feedback plan that takes a map of a grid and performs localization.

3. Suppose that a robot operates in the environment shown in Figure 12.54 using the same motion and sensing model as in Example 12.1. Design an information-feedback plan that is as simple as possible and successfully localizes the robot, regardless of its initial state. Assume the initial condition $\eta_0 = X$.

4. Prove that the robot can use the latitude and orientation information to detect the unique point of each obstacle boundary in the maze searching algorithm of Section 12.3.1.

5. Suppose once again that a robot is placed into one of the six environments shown in Figure 12.12. It is initially in the upper right cell facing north; however, the initial condition is $\eta_0 = X$. Determine the sequence of sensor observations and nondeterministic I-states as the robot executes the action sequence (F, R, B, F, L, L, F).

6. Prove that the counter in the maze searching algorithm of Section 12.3.1 can be replaced by two pebbles, and the robot can still solve the problem by simulating the counter. The robot can place either pebble on a tile, detect them when the robot is on the same tile, and can pick them up to move them to other tiles.

7. Continue the trajectory shown in Figure 12.23 until the goal is reached using the Bug2 strategy.

8. Show that the competitive ratio for the doubling spiral motion applied to the lost-cow problem of Figure 12.26 is 9.

9. Generalize the lost-cow problem so that there are n fences that emanate from the current cow location ($n = 2$ for the original problem).

   (a) If the cow is told that the gate is along only one unknown fence and is no more than one unit away, what is the competitive ratio of the best plan that you can think of?

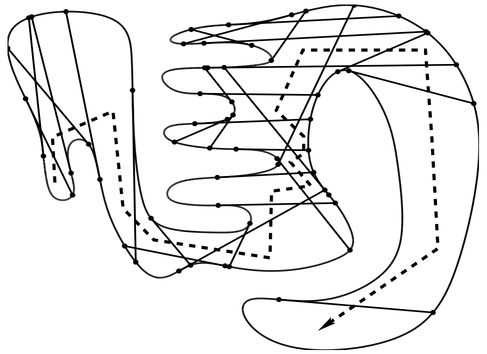


Figure 12.55: A path followed by the robot in an initially unknown environment. The robot finishes in the lower right.

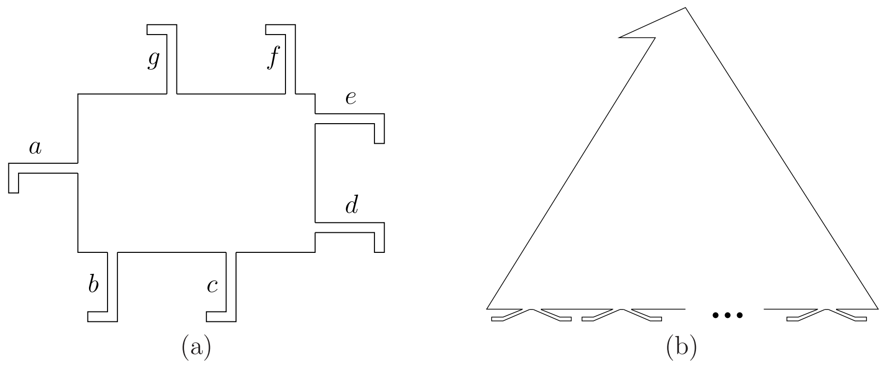


Figure 12.56: Two pursuit-evasion problems that involve recontamination.

(b) Suppose the cow does not know the maximum distance to the gate. Propose a plan that solves the problem and establish its competitive ratio.

10. Suppose a point robot is dropped into the environment shown in Figure 12.42. Indicate the gap navigation trees that are obtained as the robot moves along the path shown in Figure 12.55.

11. Construct an example for which the worst case bound, (12.25), for Bug1 is obtained.

12. Some environments are so complicated that in the pursuit-evasion problem they require the same region to be visited multiple times. Find a solution for a single pursuer with omnidirectional visibility to the problem in Figure 12.56a.

13. Find a pursuit-evasion solution for a single pursuer with omnidirectional visibility to the problem in Figure 12.56b, in which any number of pairs of "feet" may appear on the bottom of the polygon.

14. Prove that for a polygonal environment, if there are three points, p_1, p_2, and p_3, for which $V(V(p_1))$, $V(V(p_2))$, and $V(V(p_3))$ are pairwise-disjoint, then the problem requires more than one pursuer.

15. Prove that the diameter function for the squeezing algorithm in Section 12.5.2 has no more than $O(n^2)$ vertices. Give a sequence of polygons that achieves this bound. What happens for a regular polygon?

16. Develop versions of (12.8) and (12.9) for state-nature sensor mappings.

17. Develop versions of (12.8) and (12.9) for history-based sensor mappings.

18. Describe in detail the I-map used for maze searching in Section 12.3.1. Indicate how this is an example of dramatically reducing the size of the I-space, as described in Section 11.2. Is a sufficient I-map obtained?

19. Describe in detail the I-map used in the Bug1 algorithm. Is a sufficient I-map obtained?

20. Suppose that several teams of point robots move around in a simple polygon. Each robot has an omnidirectional visibility sensor and would like to keep track of information for each shadow region. For each team and shadow region, it would like to record one of three possibilities: 1) There are definitely no team members in the region; 2) there may possibly be one or more; 3) there is definitely at least one.

    (a) Define a nondeterministic I-space based on labeling gaps that captures the appropriate information. The I-space should be defined with respect to one robot (each will have its own).

    (b) Design an algorithm that keeps track of the nondeterministic I-state as the robot moves through the environments and observes others.

21. Recall the sequence of connected corridors shown in Figure 12.40. Try to adapt the polygons so that the same number of pursuers is needed, but there are fewer polygon edges. Try to use as few edges as possible.

Implementations

22. Solve the probabilistic passive localization problem of Section 12.2.3 for 2D grids. Implement your solution and demonstrate it on several interesting examples.

23. Implement the exact value-iteration method described in Section 12.1.3 to compute optimal cost-to-go functions. Test the implementation on several small examples. How large can you make K, Θ, and Ψ?

24. Develop and implement a graph search algorithm that searches on $\mathcal{I}_{ndet}$ to perform robot localization on a 2D grid. Test the algorithm on several interesting examples. Try developing search heuristics that improve the performance.

25. Implement the Bug1, Bug2, and VisBug (with unlimited radius) algorithms. Design a good set of examples for illustrating their relative strengths and weaknesses.

26. Implement software that computes probabilistic I-states for localization as the robot moves in a grid.

27. Implement the method of Section 12.3.4 for simply connected environments and demonstrate it in simulation for polygonal environments.

28. Implement the pursuit-evasion algorithm for a single pursuer in a simple polygon.

29. Implement the part-squeezing algorithm presented in Section 12.5.2.

Bibliography

[1] P. K. Agarwal, J.-C. Latombe, R. Motwani, and P. Raghavan. Nonholonomic path planning for pushing a disk among obstacles. In *Proceedings IEEE International Conference on Robotics & Automation*, 1997.

[2] S. Akella, W. H. Huang, K. M. Lynch, and M. T. Mason. Parts feeding on a conveyor with a one joint robot. *Algorithmica*, 26(3/4):313–344, March/April 2000.

[3] E. Alpaydin. *Machine Learning*. MIT Press, Cambridge, MA, 2004.

[4] K. Arras, N. Tomatis, B. Jensen, and R. Siegwart. Multisensor on-the-fly localization: Precision and reliability for applications. *Robotics and Autonomous Systems*, 34(2-3):131–143, 2001.

[5] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[6] R. B. Ash. *Information Theory*. Dover, New York, 1990.

[7] T. Başar. Game theory and H^∞-optimal control: The continuous-time case. In R. P. Hämäläinen and H. K. Ehtamo, editors, *Differential Games – Developments in Modelling and Computation*, pages 171–186. Springer-Verlag, Berlin, 1991.

[8] T. Başar and P. R. Kumar. On worst case design strategies. *Computers and Mathematics with Applications*, 13(1-3):239–245, 1987.

[9] T. Başar and G. J. Olsder. *Dynamic Noncooperative Game Theory, 2nd Ed.* Academic, London, 1995.

[10] R. A. Baeza, J. C. Culberson, and G. J. E. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.

[11] J. E. Banta, Y. Zhien, X. Z. Wang, G. Zhang, M. T. Smith, and M. A. Abidi. A "best-next-view" algorithm for three-dimensional scene reconstruction using range images. In *Proceedings SPIE, vol. 2588*, pages 418–29, 1995.

[12] J. Barraquand and P. Ferbach. Motion planning with uncertainty: The information space approach. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1341–1348, 1995.

[13] A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and sequential decision making. In M. Gabriel and J.W. Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 539–602. MIT Press, Cambridge, MA, 1990.

[14] K. Basye and T. Dean. Map learning with indistinguishable locations. In M. Henrion, L. N. Kanal, and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence 5*, pages 331–340. Elsevier Science, New York, 1990.

[15] K. Basye, T. Dean, J. Kirman, and M. Lejter. A decision-theoretic approach to planning, perception, and control. *IEEE Expert*, 7(4):58–65, August 1992.

[16] T. Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53, 1763.

[17] C. Becker, H. González-Baños, J.-C. Latombe, and C. Tomasi. An intelligent observer. In *Preprints of International Symposium on Experimental Robotics*, pages 94–99, 1995.

[18] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[19] M. A. Bender, A. Fernandez, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proceedings Annual Symposium on Foundations of Computer Science*, 1998.

[20] A. Benveniste, M. Metivier, and P. Prourier. *Adaptive Algorithms and Stochastic Approximations*. Springer-Verlag, Berlin, 1990.

[21] J. O. Berger. *Statistical Decision Theory*. Springer-Verlag, Berlin, 1980.

[22] D. P. Bertsekas. Convergence in discretization procedures in dynamic programming. *IEEE Transactions on Automatic Control*, 20(3):415–419, June 1975.

[23] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[24] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 1999.

[25] D. P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. I, 2nd Ed.* Athena Scientific, Belmont, MA, 2001.

[26] D. P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II, 2nd Ed.* Athena Scientific, Belmont, MA, 2001.

[27] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[28] D. Bienstock and P. Seymour. Monotonicity in graph searching. *Journal of Algorithms*, 12:239–245, 1991.

[29] J. Bilmes. A gentle tutorial on the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report ICSI-TR-97-021, International Computer Science Institute (ICSI), Berkeley, CA, 1997.

[30] R. L. Bishop and S. I. Goldberg. *Tensor Analysis on Manifolds.* Dover, New York, 1980.

[31] D. Blackwell and M. A. Girshik. *Theory of Games and Statistical Decisions.* Dover, New York, 1979.

[32] S. Blind, C. McCullough, S. Akella, and J. Ponce. Manipulating parts with an array of pins: A method and a machine. *International Journal of Robotics Research*, 20(10):808–818, December 2001.

[33] A. Bloch. *Murphy's Law and Other Reasons Why Things Go Wrong.* Price Stern Sloan Adult, New York, 1977.

[34] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrains. In *Proceedings ACM Symposium on Computational Geometry*, pages 494–504, 1991.

[35] M. Blum and D. Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *Proceedings Annual Symposium on Foundations of Computer Science*, pages 132–142, 1978.

[36] G. Boo and K. Goldberg. Orienting polygonal parts without sensors: An implementation in Java. Alpha Lab, UC Berkeley. Available from http://www.ieor.berkeley.edu/∼goldberg/feeder-S05/, 2005.

[37] J. Borenstein, B. Everett, and L. Feng. *Navigating Mobile Robots: Systems and Techniques.* A.K. Peters, Wellesley, MA, 1996.

[38] P. Bose, A. Lubiv, and J. I. Munro. Efficient visibility queries in simple polygons. In *Proceedings Canadian Conference on Computational Geometry*, pages 23–28, 1992.

[39] A. Briggs. An efficient algorithm for one-step compliant motion planning with uncertainty. In *Proceedings ACM Symposium on Computational Geometry*, 1989.

[40] A. J. Briggs and B. R. Donald. Robust geometric algorithms for sensor planning. In J.-P. Laumond and M. Overmars, editors, *Proceedings Workshop on Algorithmic Foundations of Robotics.* A.K. Peters, Wellesley, MA, 1996.

[41] R. C. Brost. Automatic grasp planning in the presence of uncertainty. *International Journal of Robotics Research*, 7(1):3–17, 1988.

[42] R. C. Brost. *Analysis and Planning of Planar Manipulation Tasks.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1991.

[43] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A research agenda. In *Proceedings IEEE International Conference on Robotics & Automation*, volume 3, pages 549–556, 1993.

[44] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A computational framework. Technical Report SAND92-2033, Sandia National Laboratories, Albuquerque, NM, January 1994.

[45] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A computational framework. *International Journal of Robotics Research*, 15(1):1–23, February 1996.

[46] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings AAAI National Conference on Artificial Intelligence*, pages 11–18, 1998.

[47] J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proceedings IEEE Symposium on Foundations of Computer Science*, pages 49–60, 1987.

[48] J. F. Canny. On computability of fine motion plans. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 177–182, 1989.

[49] J. F. Canny and K. Y. Goldberg. "RISC" industrial robots: Recent results and current trends. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1951–1958, 1994.

[50] J. Castellanos, J. Montiel, J. Neira, and J. Tardós. The SPmap: A probabilistic framework for simultaneous localization and mapping. *IEEE Transactions on Robotics & Automation*, 15(5):948–953, 1999.

[51] C.-T. Chen. *Linear System Theory and Design.* Holt, Rinehart, and Winston, New York, 1984.

[52] Y.-B. Chen and D. J. Ierardi. The complexity of oblivious plans for orienting and distinguishing polygonal parts. *Algorithmica*, 14:367–397, 1995.

[53] W.-P. Chin and S. Ntafos. Optimum watchman routes. *Information Processing Letters*, 28:39–44, 1988.

[54] H. Choset and J. Burdick. Sensor based motion planning: Incremental construction of the hierarchical generalized Voronoi graph. *International Journal of Robotics Research*, 19(2):126–148, 2000.

[55] H. Choset and J. Burdick. Sensor based motion planning: The hierarchical generalized Voronoi graph. *International Journal of Robotics Research*, 19(2):96–125, 2000.

[56] H. Choset and K. Nagatani. Topological simultaneous localization and mapping (T-SLAM). *IEEE Transactions on Robotics & Automation*, 17(2):125–137, 2001.

[57] K.-C. Chu. Team decision theory and information structures in optimal control problems – Part II. *IEEE Transactions on Automatic Control*, 17(1):22–28, February 1972.

[58] C. K. Chui and G. Chen. *Kalman Filtering*. Springer-Verlag, Berlin, 1991.

[59] F. S. Cohen and D. B. Cooper. Simple parallel hierarchical and relaxation algorithms for segmenting noncausal Markovian random fields. *IEEE Transactions Pattern Analysis Machine Intelligence*, 9(2):195–219, March 1987.

[60] C. I. Connolly. The determination of next best views. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 432–435, 1985.

[61] H. W. Corley. Some multiple objective dynamic programs. *IEEE Transactions on Automatic Control*, 30(12):1221–1222, December 1985.

[62] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, New York, 1991.

[63] D. Crass, I. Suzuki, and M. Yamashita. Searching for a mobile intruder in a corridor – The open edge variant of the polygon search problem. *International Journal Computational Geometry & Applications*, 5(4):397–412, 1995.

[64] F. Dallaert, D. Fox, W. Burgard, and S. Thrun. Monte Carlo localization for mobile robots. In *Proceedings IEEE International Conference on Robotics & Automation*, 1999.

[65] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[66] A. Datta, C. A. Hipke, and S. Schuierer. Competitive searching in polygons–beyond generalized streets. In J. Staples, P. Eades, N. Katoh, and A. Moffat, editors, *Algorithms and Computation, ISAAC '95*, pages 32–41. Springer-Verlag, Berlin, 1995.

[67] R. S. Datta. Using computer algebra to compute Nash equilibria. In *Proceedings International Symposium on Symbolic and Algebraic Computation*, 2003.

[68] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications, 2nd Ed.* Springer-Verlag, Berlin, 2000.

[69] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufman, San Francisco, CA, 1991.

[70] M. H. DeGroot. *Optimal Statistical Decisions*. McGraw-Hill, New York, 1970.

[71] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum-likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Ser. B.*, 39:1–38, 1977.

[72] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment I: The rectilinear case. Available from http://www.cs.berkeley.edu/∼christos/, 1997.

[73] P. A. Devijver and J. Kittler. *Pattern Recognition: A Statistical Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[74] G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localisation and map building (SLAM) problem. *IEEE Transactions on Robotics & Automation*, 17(3):229–241, 2001.

[75] B. R. Donald. *Error Detection and Recovery for Robot Motion Planning with Uncertainty*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[76] B. R. Donald. The complexity of planar compliant motion planning under uncertainty. In *Proceedings ACM Symposium on Computational Geometry*, pages 309–318, 1988.

[77] B. R. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artificial Intelligence Journal*, 37:223–271, 1988.

[78] B. R. Donald. Planning multi-step error detection and recovery strategies. *International Journal of Robotics Research*, 9(1):3–60, 1990.

[79] B. R. Donald. On information invariants in robotics. *Artificial Intelligence Journal*, 72:217–304, 1995.

[80] B. R. Donald and J. Jennings. Sensor interpretation and task-directed planning using perceptual equivalence classes. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 190–197, 1991.

[81] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, Berlin, 2001.

[82] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification, 2nd Ed.* Wiley, New York, 2000.

[83] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, Cambridge, U.K., 2000.

[84] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Map validation and self-location in a graph-like world. In *Proceedings AAAI National Conference on Artificial Intelligence*, pages 1648–1653, 1993.

[85] G. Dudek, K. Romanik, and S. Whitesides. Global localization: Localizing a robot with minimal travel. *SIAM Journal on Computing*, 27(2):583–604, April 1998.

[86] G. E. Dullerud and F. Paganini. *A Course in Robust Control Theory*. Springer-Verlag, Berlin, 2000.

[87] A. Efrat, L. J. Guibas, S. Har-Peled, D. C. Lin, J. S. B. Mitchell, and T. M. Murali. Sweeping simple polygons with a chain of guards. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[88] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *IEEE Computer*, 22(6):46–57, June 1989.

[89] G. Ellis. *Observers in Control Systems*. Elsevier, New York, 2002.

[90] M. A. Erdmann. On motion planning with uncertainty. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1984.

[91] M. A. Erdmann. Using backprojections for fine motion planning with uncertainty. *International Journal of Robotics Research*, 5(1):19–45, 1986.

[92] M. A. Erdmann. *On Probabilistic Strategies for Robot Tasks*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1989.

[93] M. A. Erdmann. Randomization in robot tasks. *International Journal of Robotics Research*, 11(5):399–436, October 1992.

[94] M. A. Erdmann. Randomization for robot tasks: Using dynamic programming in the space of knowledge states. *Algorithmica*, 10:248–291, 1993.

[95] M. A. Erdmann. On a representation of friction in configuration space. *International Journal of Robotics Research*, 13(3):240–271, 1994.

[96] M. A. Erdmann. An exploration of nonprehensile two-palm manipulation using two zebra robots. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 239–254. A.K. Peters, Wellesley, MA, 1997.

[97] M. A. Erdmann and M. T. Mason. An exploration of sensorless manipulation. *IEEE Transactions on Robotics & Automation*, 4(4):369–379, August 1988.

[98] M. A. Erdmann, M. T. Mason, and G. Vaněček. Mechanical parts orienting: The case of a polyhedron on a table. *Algorithmica*, 10:206–247, 1993.

[99] B. Espiau, F. Chaumette, and P. Rives. A new approach to visual servoing in robotics. *IEEE Transactions on Robotics & Automation*, 8(3):313–326, June 1992.

[100] S. P. Fekete, R. Klein, and A. Nüchter. Online searching with an autonomous robot. In *Proceedings Workshop on Algorithmic Foundations of Robotics*, Zeist, The Netherlands, July 2004.

[101] R. J. Firby. An investigation into reactive planning in complex domains. In *Proceedings AAAI National Conference on Artificial Intelligence*, 1987.

[102] R. Fleischer, K. Romanik, S. Schuierer, and G. Trippen. Optimal localization in trees. *Information and Computation*, 171(2):224–247, 2002.

[103] G. B. Folland. *Real Analysis: Modern Techniques and Their Applications*. Wiley, New York, 1984.

[104] A. Fox and S. Hutchinson. Exploiting visual constraints in the synthesis of uncertainty-tolerant motion plans. *IEEE Transactions on Robotics & Automation*, 1(11):56–71, February 1995.

[105] D. Fox, S. Thrun, W. Burgard, and F. Dallaert. Particle filters for mobile robot localization. In A. Doucet, N. de Freitas, and N. Gordon, editors, *Sequential Monte Carlo Methods in Practice*, pages 401–428. Springer-Verlag, Berlin, 2001.

[106] J. Fraden. *Handbook of Modern Sensors: Physics, Designs, and Applications*. Springer-Verlag, Berlin, 2003.

[107] T. Fukuda, S. Ito, N. Oota, F. Arai, Y. Abe, K. Tanake, and Y. Tanaka. Navigation system based on ceiling landmark recognition for autonomous mobile robot. In *Proceedings International Conference on Industrial Electronics, Control, and Instrumentation*, pages 1466–1471, 1993.

[108] Y. Gabriely and E. Rimon. Competitive complexity of mobile robot on line motion planning problems. In *Proceedings Workshop on Algorithmic Foundations of Robotics*, pages 249–264, 2004.

[109] D. Geman and S. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions Pattern Analysis Machine Intelligence*, 6(6):721–741, November 1984.

[110] B. Gerkey, S. Thrun, and G. Gordon. Clear the building: Pursuit-evasion with teams of robots. In *Proceedings AAAI National Conference on Artificial Intelligence*, 2004.

[111] P. J. Gmytrasiewicz, E. H. Durfee, and D. K. Wehe. A decision-theoretic approach to coordinating multi-agent interactions. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 62–68, 1991.

[112] K. Y. Goldberg. *Stochastic Plans for Robotic Manipulation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 1990.

[113] K. Y. Goldberg. Orienting polygonal parts without sensors. *Algorithmica*, 10:201–225, 1993.

[114] K. Y. Goldberg and M. T. Mason. Bayesian grasping. In *Proceedings IEEE International Conference on Robotics & Automation*, 1990.

[115] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd ed)*. Johns Hopkins University Press, Baltimore, MD, 1996.

[116] H. H. González-Baños, L. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, R. Motwani, and C. Tomasi. Motion planning with visibility constraints: Building autonomous observers. In Y. Shirai and S. Hirose, editors, *Proceedings Eighth International Symposium on Robotics Research*, pages 95–101. Springer-Verlag, Berlin, 1998.

[117] H. H. González-Baños, C.-Y. Lee, and J.-C. Latombe. Real-time combinatorial tracking of a target moving unpredictably among obstacles. In *Proceedings IEEE International Conference on Robotics & Automation*, 2002.

[118] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[119] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani. Visibility-based pursuit-evasion in a polygonal environment. *International Journal of Computational Geometry and Applications*, 9(5):471–494, 1999.

[120] L. J. Guibas, R. Motwani, and P. Raghavan. The robot localization problem. In K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, editors, *Algorithmic Foundations of Robotics*, pages 269–282. A.K. Peters, Wellesley, MA, 1995.

[121] L. Guilamo, B. Tovar, and S. M. LaValle. Pursuit-evasion in an unknown environment using gap navigation trees. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[122] J.-S. Gutmann, T. Weigel, and B. Nebel. A fast, accurate, and robust method for self-localization in polygonal environments using laser-range-finders. *Advanced Robotics*, 14(8):651–668, 2001.

[123] O. Hájek. *Pursuit Games*. Academic, New York, 1975.

[124] R. Hinkel and T. Knieriemen. Environment perception with a laser radar in a fast moving robot. In *Proceedings Symposium on Robot Control*, pages 68.1–68.7, Karlsruhe, Germany, 1988.

[125] Y.-C. Ho and K.-C. Chu. Team decision theory and information structures in optimal control problems-Part I. In *IEEE Transactions on Automatic Control*, pages 15–22, 1972.

[126] J. E. Hopcroft, J. D. Ullman, and R. Motwani. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 2000.

[127] S. A. Hutchinson, G. D. Hager, and P. I. Corke. A tutorial on visual servo control. *IEEE Transactions on Robotics & Automation*, 12(5):651–670, October 1996.

[128] M. Hutter. *Universal Artificial Intelligence*. Springer-Verlag, Berlin, 2005.

[129] R. Isaacs. *Differential Games*. Wiley, New York, 1965.

[130] A. Isidori. *Nonlinear Control Systems, 2nd Ed.* Springer-Verlag, Berlin, 1989.

[131] P. Jensfelt. *Approaches to Mobile Robot Localization*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2001.

[132] P. Jensfelt and H. I. Christensen. Pose tracking using laser scanning and minimalistic environmental models. *IEEE Transactions on Robotics & Automation*, 17(2):138–147, 2001.

[133] P. Jensfelt and S. Kristensen. Active global localisation for a mobile robot using multiple hypothesis tracking. *IEEE Transactions on Robotics & Automation*, 17(5):748–760, October 2001.

[134] X. Ji and J. Xiao. Planning motion compliant to complex contact states. *International Journal of Robotics Research*, 20(6):446–465, 2001.

[135] L. P. Kaelbling, A. Cassandra, and J. Kurien. Acting under uncertainty: Discrete Bayesian models for mobile robot navigation. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 963–972, 1996.

[136] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence Journal*, 101, 1998.

[137] R. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME, Journal of Basic Engineering*, 82:35–45, 1960.

[138] T. Kameda, M. Yamashita, and I. Suzuki. On-line polygon search by a seven-state boundary 1-searcher. *IEEE Transactions on Robotics*, 2006. To appear.

[139] I. Kamon and E. Rivlin. Sensory-based motion planning with global proofs. *IEEE Transactions on Robotics & Automation*, 13(6):814–822, December 1997.

[140] I. Kamon, E. Rivlin, and E. Rimon. Range-sensor based navigation in three dimensions. In *Proceedings IEEE International Conference on Robotics & Automation*, 1999.

[141] M.-Y. Kao, J. H. Reif, and S. R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 441–447, 1993.

[142] K. H. Kim and F. W. Roush. *Team Theory*. Ellis Horwood Limited, Chichester, U.K., 1987.

[143] G. Kitagawa. Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5(1), 1996.

[144] J. M. Kleinberg. On-line algorithms for robot navigation and server problems. Technical Report MIT/LCS/TR-641, MIT, Cambridge, MA, May 1994.

[145] S. A. Klugman, H. H. Panjer, and G. E. Willmot. *Loss Models: From Data to Decisions, 2nd Ed.* Wiley, New York, 2004.

[146] S. Koenig and M. Likhachev. D^* lite. In *Proceedings AAAI National Conference on Artificial Intelligence*, pages 476–483, 2002.

[147] D. Koller, N. Megiddo, and B. von Stengel. Efficient computation of equilibria for extensive two-person games. *Games and Economic Behavior*, 14:247–259, 1996.

[148] A. N. Kolmogorov and S. V. Fomin. *Introductory Real Analysis*. Dover, New York, 1975.

[149] K. Konolige. Markov localization using correlation. In *Proceedings International Joint Conference on Artificial Intelligence*, 1999.

[150] H. W. Kuhn. Extensive games and the problem of information. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games*, pages 196–216. Princeton University Press, Princeton, NJ, 1953.

[151] P. R. Kumar and P. Varaiya. *Stochastic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[152] H. J. Kushner and D. S. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Systems*. Springer-Verlag, Berlin, 1978.

[153] H. J. Kushner and P. G. Dupuis. *Numerical Methods for Stochastic Control Problems in Continuous Time*. Springer-Verlag, Berlin, 1992.

[154] K. N. Kutulakos, C. R. Dyer, and V. J. Lumelsky. Provable strategies for vision-guided exploration in three dimensions. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1365–1371, 1994.

[155] A. S. Lapaugh. Recontamination does not help to search a graph. *Journal of the ACM*, 40(2):224–245, April 1993.

[156] P.-S. Laplace. *Théorie Analityque des Probabilités*. Courceir, Paris, 1812.

[157] R. E. Larson and W. G. Keckler. Optimum adaptive control in an unknown environment. *IEEE Transactions on Automatic Control*, 13(4):438–439, August 1968.

[158] J.-C. Latombe. *Robot Motion Planning*. Kluwer, Boston, MA, 1991.

[159] J.-C. Latombe, A. Lazanas, and S. Shekhar. Robot motion planning with uncertainty in control and sensing. *Artificial Intelligence Journal*, 52:1–47, 1991.

[160] S. L. Laubach and J. W. Burdick. An autonomous sensor-based path-planning for planetary microrovers. In *Proceedings IEEE International Conference on Robotics & Automation*, 1999.

[161] S. M. LaValle. *A Game-Theoretic Framework for Robot Motion Planning*. PhD thesis, University of Illinois, Urbana, IL, July 1995.

[162] S. M. LaValle. Robot motion planning: A game-theoretic foundation. *Algorithmica*, 26(3):430–465, 2000.

[163] S. M. LaValle, H. H. González-Baños, C. Becker, and J.-C. Latombe. Motion strategies for maintaining visibility of a moving target. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 731–736, 1997.

[164] S. M. LaValle and J. Hinrichsen. Visibility-based pursuit-evasion: The case of curved environments. *IEEE Transactions on Robotics and Automation*, 17(2):196–201, April 2001.

[165] S. M. LaValle and S. A. Hutchinson. An objective-based stochastic framework for manipulation planning. In *Proceedings IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, pages 1772–1779, September 1994.

[166] S. M. LaValle and S. A. Hutchinson. An objective-based framework for motion planning under sensing and control uncertainties. *International Journal of Robotics Research*, 17(1):19–42, January 1998.

[167] S. M. LaValle, D. Lin, L. J. Guibas, J.-C. Latombe, and R. Motwani. Finding an unpredictable target in a workspace with obstacles. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 737–742, 1997.

[168] S. M. LaValle and R. Sharma. On motion planning in changing, partially-predictable environments. *International Journal of Robotics Research*, 16(6):775–805, December 1997.

[169] S. Lazebnik. Visibility-based pursuit evasion in three-dimensional environments. Technical Report CVR TR 2001-01, Beckman Institute, University of Illinois, 2001.

[170] J.-H. Lee, S. Y. Shin, and K.-Y. Chwa. Visibility-based pursuit-evasions in a polygonal room with a door. In *Proceedings ACM Symposium on Computational Geometry*, 1999.

[171] S. Lenser and M. Veloso. Sensor resetting localization for poorly modelled mobile robots. In *Proceedings IEEE International Conference on Robotics & Automation*, 2000.

[172] J. Leonard, H. Durrant-Whyte, and I. Cox. Dynamic map building for an autonomous mobile robot. *International Journal of Robotics Research*, 11(4):89–96, 1992.

[173] M. Littman. The witness algorithm: Solving partially observable Markov decision processes. Technical Report CS-94-40, Brown University, Providence, RI, 1994.

[174] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially-observable environments: Scaling up. In *Proceedings International Conference on Machine Learning*, pages 362–370, 1995.

[175] W. S. Lovejoy. Computationally feasible bounds for partially observed Markov decision processes. *Operations Research*, 39(1):162–175, 1991.

[176] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computing*, C-32(2):108–120, 1983.

[177] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1):3–24, 1984.

[178] D. G. Luenberger. *Introduction to Linear and Nonlinear Programming*. Wiley, New York, 1973.

[179] V. J. Lumelsky and T. Skewis. Incorporating range sensing in the robot navigation function. *IEEE Transactions on Systems, Man, & Cybernetics*, 20(5):1058–1069, 1990.

[180] V. J. Lumelsky and A. A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[181] V. J. Lumelsky and S. Tiwari. An algorithm for maze searching with azimuth input. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 111–116, 1994.

[182] K. M. Lynch and M. T. Mason. Pulling by pushing, slip with infinite friction, and perfectly rough surfaces. *International Journal of Robotics Research*, 14(2):174–183, 1995.

[183] K. M. Lynch and M. T. Mason. Stable pushing: Mechanics, controllability, and planning. *International Journal of Robotics Research*, 15(6):533–556, 1996.

[184] I. M. Makarov, T. M. Vinogradskaya, A. A. Rubchinsky, and V. B. Sokolov. *The Theory of Choice and Decision Making*. Mir Publishers, Moscow, 1987.

[185] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for on-line problems. In *Proceedings ACM Symposium on Theory of Computing*, pages 322–333, 1988.

[186] M. T. Mason. Compliance and force control for computer controlled manipulators. In M. Brady *et al.*, editor, *Robot Motion: Planning and Control*, pages 373–404. MIT Press, Cambridge, MA, 1982.

[187] M. T. Mason. The mechanics of manipulation. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 544–548, 1985.

[188] M. T. Mason. Mechanics and planning of manipulator pushing operations. *International Journal of Robotics Research*, 5(3):53–71, 1986.

[189] M. T. Mason. *Mechanics of Robotic Manipulation*. MIT Press, Cambridge, MA, 2001.

[190] L. Matthies and A. Elfes. Integration of sonar and stereo range data using a grid-based representation. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 727–733, 1988.

[191] R. McKelvey and A. McLennan. Computation of equilibria in finite games. In H. Amman, D. A. Kendrick, and J .Rust, editors, *The Handbook of Computational Economics*, pages 87–142. Elsevier, New York, 1996.

[192] D. A. Miller and S. W. Zucker. Copositive-plus Lemke algorithm solves polymatrix games. *Operations Research Letters*, 10:285–290, 1991.

[193] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[194] G. Monahan. A survey of partially observable Markov decision processes. *Management Science*, 101(1):1–16, 1982.

[195] B. Monien and I. H. Sudborough. Min cut is NP-complete for edge weighted graphs. *Theoretical Computer Science*, 58:209–229, 1988.

[196] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings AAAI National Conference on Artificial Intelligence*, 1999.

[197] R. Munos. Error bounds for approximate value iteration. In *Proceedings AAAI National Conference on Artificial Intelligence*, 2005.

[198] T. Muppirala, R. Murrieta-Cid, and S. Hutchinson. Optimal motion strategies based on critical events to maintain visibility of a moving target. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 3837–3842, 2005.

[199] R. Murrieta-Cid, A. Sarmiento, S. Bhattacharya, and S. Hutchinson. Maintaining visibility of a moving target at a fixed distance: The case of observer bounded speed. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 479–484, 2004.

[200] J. Nash. Noncooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

[201] S. G. Nash and A. Sofer. *Linear and Nonlinear Programming*. McGraw-Hill, New York, 1996.

[202] B. K. Natarajan. The complexity of fine motion planning. *International Journal of Robotics Research*, 7(2):36–42, 1988.

[203] A. Y. Ng and M. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In *Proceedings Conference on Uncertainty in Artificial Intelligence*, 2000.

[204] S. Ntafos. Watchman routes under limited visibility. *Computational Geometry: Theory and Applications*, 1:149–170, 1992.

[205] J. M. O'Kane. Global localization using odometry. In *Proceedings IEEE International Conference on Robotics and Automation*, 2005.

[206] J. M. O'Kane and S. M. LaValle. Almost-sensorless localization. In *Proceedings IEEE International Conference on Robotics and Automation*, 2005.

[207] S. Oore, G. E. Hinton, and G. Dudek. A mobile robot that learns its place. *Neural Computation*, 9:683–699, 1997.

[208] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, 1987.

[209] J. O'Rourke. Visibility. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry, 2nd Ed.*, pages 643–663. Chapman and Hall/CRC Press, New York, 2004.

[210] G. Owen. *Game Theory*. Academic, New York, 1982.

[211] D. K. Pai and L. M. Reissell. Multiresolution rough terrain motion planning. *IEEE Transactions on Robotics & Automation*, 14(5):709–717, 1998.

[212] C. H. Papadimitriou. Games against nature. *Journal of Computer and System Sciences*, 31:288–301, 1985.

[213] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987.

[214] C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

[215] A. Papantonopoulou. *Algebra: Pure and Applied*. Prentice Hall, Englewood Cliffs, NJ, 2002.

[216] S.-M. Park, J.-H. Lee, and K.-Y. Chwa. Visibility-based pursuit-evasion in a polygonal region by a searcher. Technical Report CS/TR-2001-161, Dept. of Computer Science, KAIST, Seoul, South Korea, January 2001.

[217] R. Parr and A. Eliazar. DP-SLAM: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *Proceedings International Joint Conference on Artificial Intelligence*, 2003.

[218] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings International Joint Conference on Artificial Intelligence*, 1995.

[219] T. D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. R. Lick, editors, *Theory and Application of Graphs*, pages 426–441. Springer-Verlag, Berlin, 1976.

[220] T. Parthasarathy and M. Stern. Markov games: A survey. In *Differential Games and Control Theory II*, pages 1–46. Marcel Dekker, New York, 1977.

[221] R. P. Paul and B. Shimano. Compliance and control. In *Proceedings of the Joint American Automatic Control Conference*, pages 1694–1699, 1976.

[222] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, 1988.

[223] B. Peleg and P. Sudlölter. *Introduction to the Theory of Cooperative Games*. Springer-Verlag, Berlin, 2003.

[224] S. Petitjean, D. Kriegman, and J. Ponce. Computing exact aspect graphs of curved objects: algebraic surfaces. *International Journal of Computer Vision*, 9:231–255, Dec 1992.

[225] L. A. Petrosjan. *Differential Games of Pursuit*. World Scientific, Singapore, 1993.

[226] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 1025–1032, 2003.

[227] R. Pito. A sensor based solution to the next best view problem. In *International Conference Pattern Recognition*, 1996.

[228] M. Pocchiola and G. Vegter. The visibility complex. *International Journal Computational Geometry & Applications*, 6(3):279–308, 1996.

[229] J. M. Porta, M. T. J. Spaan, and N. Vlassis. Robot planning in partially observable continuous domains. In *Proceedings Robotics: Science and Systems*, 2005.

[230] P. Poupart and C. Boutilier. Value-directed compression of POMDPs. In *Proceedings Neural Information Processing Systems*, 2003.

[231] S. Rajko and S. M. LaValle. A pursuit-evasion bug algorithm. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 1954–1960, 2001.

[232] A. Rao and K. Goldberg. Manipulating algebraic parts in the plane. *IEEE Transactions on Robotics & Automation*, 11(4):598–602, 1995.

[233] N. Rao, S. Kareti, W. Shi, and S. Iyenagar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410:1–58, Oak Ridge National Laboratory, July 1993.

[234] E. Remolina and B. Kuipers. Towards a general theory of topological maps. *Artificial Intelligence Journal*, 152(1):47–104, 2004.

[235] W. Rencken. Concurrent localisation and map building for mobile robots using ultrasonic sensors. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2192–2197, 1993.

[236] E. Rimon and J. Canny. Construction of C-space roadmaps using local sensory data – What should the sensors look for? In *Proceedings IEEE International Conference on Robotics & Automation*, pages 117–124, 1994.

[237] C. P. Robert. *The Bayesian Choice, 2nd. Ed.* Springer-Verlag, Berlin, 2001.

[238] N. Roy and G. Gordon. Exponential family PCA for belief compression in POMDPs. In *Proceedings Neural Information Processing Systems*, 2003.

[239] H. L. Royden. *Real Analysis*. MacMillan, New York, 1988.

[240] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd Edition*. Prentice-Hall, Englewood Cliffs, NJ, 2003.

[241] S. Sachs, S. Rajko, and S. M. LaValle. Visibility-based pursuit-evasion in an unknown planar environment. *International Journal of Robotics Research*, 23(1):3–26, January 2004.

[242] H. Sagan. *Introduction to the Calculus of Variations*. Dover, New York, 1992.

[243] Y. Sawaragi, H. Nakayama, and T. Tanino. *Theory of Multiobjective Optimization*. Academic, New York, 1985.

[244] B. Schiele and J. Crowley. A comparison of position estimation techniques using occupancy grids. In *Proceedings IEEE International Conference on Robotics & Automation*, 1994.

[245] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: II. General techniques for computing topological properties of algebraic manifolds. *Communications on Pure and Applied Mathematics*, 36:345–398, 1983.

[246] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 1948.

[247] R. Sharma. Locally efficient path planning in an uncertain, dynamic environment using a probabilistic model. *IEEE Transactions on Robotics & Automation*, 8(1):105–110, February 1992.

[248] R. Sharma. A probabilistic framework for dynamic motion planning in partially known environments. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 2459–2464, Nice, France, May 1992.

[249] R. Sharma, J.-Y. Hervé, and P. Cucka. Dynamic robot manipulation using visual tracking. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1844–1849, 1992.

[250] R. Sharma, S. M. LaValle, and S. A. Hutchinson. Optimizing robot motion strategies for assembly with stochastic models of the assembly process. *IEEE Trans. on Robotics and Automation*, 12(2):160–174, April 1996.

[251] R. Sharma, D. M. Mount, and Y. Aloimonos. Probabilistic analysis of some navigation strategies in a dynamic environment. *IEEE Transactions on Systems, Man, & Cybernetics*, 23(5):1465–1474, September 1993.

[252] H. Shatkay and L. P. Kaelbling. Learning topological maps with weak local odometric information. In *Proceedings International Joint Conference on Artificial Intelligence*, 1997.

[253] T. Shermer. Recent results in art galleries. *Proceedings of the IEEE*, 80(9):1384–1399, September 1992.

[254] A. M. Shkel and V. J. Lumelsky. Incorporating body dynamics into sensor-based motion planning: The maximum turn strategy. *IEEE Transactions on Robotics & Automation*, 13(6):873–880, December 1997.

[255] R. Sim and G. Dudek. Learning generative models of scene features. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 920–929, 2001.

[256] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. O'Sullivan. A layered architecture for office delivery robots. In *Proceedings First International Conference on Autonomous Agents*, Marina del Rey, CA, 1997.

[257] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 1080–1087, 1995.

[258] B. Simov, S. M. LaValle, and G. Slutzki. A complete pursuit-evasion algorithm for two pursuers using beam detection. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 618–623, 2002.

[259] B. Simov, G. Slutzki, and S. M. LaValle. Pursuit-evasion using beam detection. In *Proceedings IEEE International Conference on Robotics and Automation*, 2000.

[260] M. Sipser. *Introduction to the Theory of Computation*. PWS, Boston, MA, 1997.

[261] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[262] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings International Conference on Machine Learning*, 2000.

[263] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *International Journal of Robotics Research*, 5(4):56–68, 1986.

[264] E. J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 9(2):149–168, 1978.

[265] W. Stadler. Fundamentals of multicriteria optimization. In W. Stadler, editor, *Multicriteria Optimization in Engineering and in the Sciences*, pages 1–25. Plenum Press, New York, 1988.

[266] R. F. Stengel. *Optimal Control and Estimation*. Dover, New York, 1994.

[267] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 3310–3317, 1994.

[268] P. D. Straffin. *Game Theory and Strategy*. Mathematical Association of America, Washington, DC, 1993.

[269] A. Sudsang, F. Rothganger, and J. Ponce. Motion planning for disc-shaped robots pushing a polygonal object in the plane. *IEEE Transactions on Robotics & Automation*, 18(4):550–562, 2002.

[270] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[271] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[272] I. Suzuki, Y. Tazoe, M. Yamashita, and T. Kameda. Searching a polygonal region from the boundary. *International Journal on Computational Geometry & Applications*, 11(5):529–553, 2001.

[273] I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal on Computing*, 21(5):863–888, October 1992.

[274] I. Suzuki, M. Yamashita, H. Umemoto, and T. Kameda. Bushiness and a tight worst-case upper bound on the search number of a simple polygon. *Information Processing Letters*, 66:49–52, 1998.

[275] H. Takeda, C. Facchinetti, and J.-C. Latombe. Planning the motions of a mobile robot in a sensory uncertainty field. *IEEE Transactions Pattern Analysis Machine Intelligence*, 16(10):1002–1017, October 1994.

[276] H. Takeda and J.-C. Latombe. Sensory uncertainty field for mobile robot navigation. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 2465–2472, Nice, France, May 1992.

[277] S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

[278] S. Thrun, W. Burgard, and D. Fox. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning*, 31(5):1–25, April 1998.

[279] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, MA, 2005.

[280] B. Tovar, L. Guilamo, and S. M. LaValle. Gap navigation trees: A minimal representation for visibility-based tasks. In *Proceedings Workshop on Algorithmic Foundations of Robotics*, pages 11–26, 2004.

[281] B. Tovar, S. M. LaValle, and R. Murrieta. Locally-optimal navigation in multiply-connected environments without geometric maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003.

[282] B. Tovar, S. M. LaValle, and R. Murrieta. Optimal navigation and object finding without geometric maps or localization. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 464–470, 2003.

[283] D. Vanderpooten. Multiobjective programming: Basic concepts and approaches. In R. Slowinski and J. Teghem, editors, *Stochastic vs. Fuzzy Approaches to Multiobjective Mathematical Programming under Uncertainty*, pages 7–22. Kluwer, Boston, MA, 1990.

[284] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

[285] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, 1944.

[286] G. Walker and D. Walker. *The Official Rock Paper Scissors Strategy Guide*. Fireside, 2004.

[287] D. S. Watkins. *Fundamentals of Matrix Computations, 2nd Ed.* Wiley, New York, 2002.

[288] G. Weiss, C. Wetzler, and E. von Puttkamer. Keeping track of position and orientation of moving indoor systems by correlation of range-finder scans. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 595–601, 1994.

[289] D. Whitney. Force feedback control of manipulator fine motions. *Transactions of the ASME, Journal of Dynamical Systems, Measurement, & Control*, 99:91–97, 1977.

[290] D. E. Whitney. Real robots don't need jigs. In *Proceedings IEEE International Conference on Robotics & Automation*, 1986.

[291] J. Wiegley, K. Goldberg, M. Peshkin, and M. Brokowski. A complete algorithm for designing passive fences to orient parts. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1133–1139, 1996.

[292] O. Wijk. *Triangulation-Based Fusion of Sonar Data with Application in Mobile Robot Mapping and Localization*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2001.

[293] C. F. J. Wu. On the convergence properties of the EM algorithm. *The Annals of Statistics*, 11(1):95–103, 1983.

[294] L. Xu and M. I. Jordan. On convergence properties of the EM algorithm for Gaussian mixtures. *Neural Computation*, 8:129–151, 1996.

[295] M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda. Searching for a mobile intruder in a polygonal region by a group of mobile searchers. *Algorithmica*, 31:208–236, 2001.

[296] B. Yamauchi, A. Schultz, and W. Adams. Mobile robot exploration and map-building with continuous localization. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 3715–3720, 2002.

[297] Y. Yavin and M. Pachter. *Pursuit-Evasion Differential Games*. Pergamon, Oxford, U.K., 1987.

[298] A. Yershova, B. Tovar, R. Ghrist, and S. M. LaValle. Bitbots: Simple robots solving complex tasks. In *AAAI National Conference on Artificial Intelligence*, 2005.

[299] N. L. Zhang and W. Zhang. Speeding up the convergence of value iteration in partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 14:29–51, 2001.

[300] R. Zhou and E. A. Hansen. An improved grid-based approximation algorithm for POMDPs. In *Proceedings International Joint Conference on Artificial Intelligence*, 2001.

[301] Y. Zhou and G. S. Chirikjian. Probabilistic models of dead-reckoning error in nonholonomic mobile robots. In *Proceedings IEEE International Conference on Robotics & Automation*, pages 1594–1599, 2003.

[302] S. Zionts. Multiple criteria mathematical programming: An overview and several approaches. In P. Serafini, editor, *Mathematics of Multi-Objective Optimization*, pages 227–273. Springer-Verlag, Berlin, 1985.