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 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

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.

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

$$\begin{split}\vec{\Theta}(\vec{x}_k, \vec{u}_k) = \{y_{k+1} \in Y \mid \exists x_{k+1} \in X_{k+1}(\eta_k, u_k) \text{ and } \exists \psi_{k+1} \in \Psi \\ \text{such that } y_{k+1} = h(x_{k+1}, \psi_{k+1})\}.\end{split} \tag{12.8}$$

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

$$G^*_k(\vec{x}_k) = \min_{\vec{u}_k \in \vec{U}} \Big\{ \vec{l}(\vec{x}_k, \vec{u}_k) + \sum_{\vec{x}_{k+1} \in \vec{X}} G^*_{k+1}(\vec{x}_{k+1}) P(\vec{x}_{k+1} | \vec{x}_k, \vec{u}_k) \Big\}. \tag{12.10}$$

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 [494]. 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.

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,

[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.

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 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

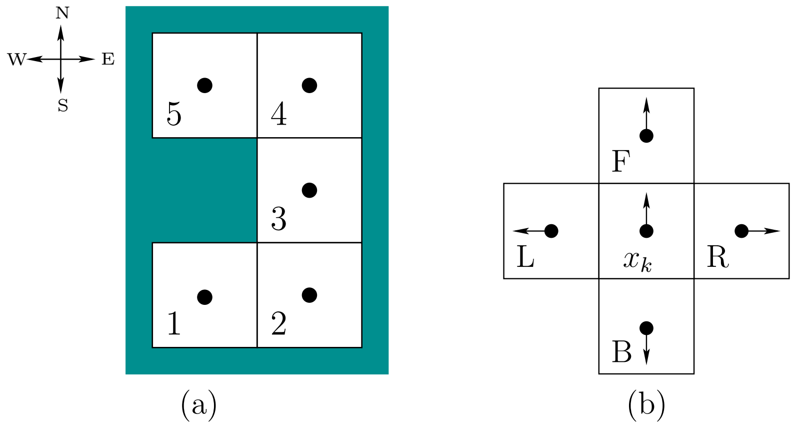


Figure 12.2: (a) This map is given to the robot for localization purposes. (b) The four possible actions each take one step, if possible, and reorient the robot as shown.

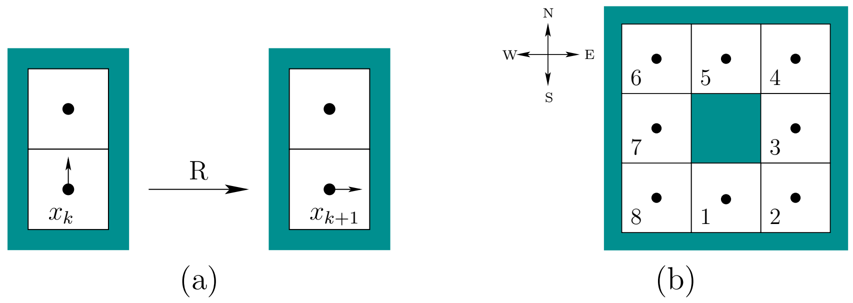


Figure 12.3: (a) If a direction is blocked because of an obstacle, then the orientation changes, but the position remains fixed. In this example, the R action is applied. (b) Another map is given to the robot for localization purposes. In this case, the robot cannot localize itself exactly.

"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, 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

$$X_2(\eta_2) = 1^{\leftarrow} \cup 2^{\rightarrow\downarrow} \cup 3^{\uparrow\downarrow} \cup 4^{\uparrow\rightarrow} \cup 5^{\leftarrow} , \tag{12.15}$$

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 $1^{\rightarrow}$ 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.

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

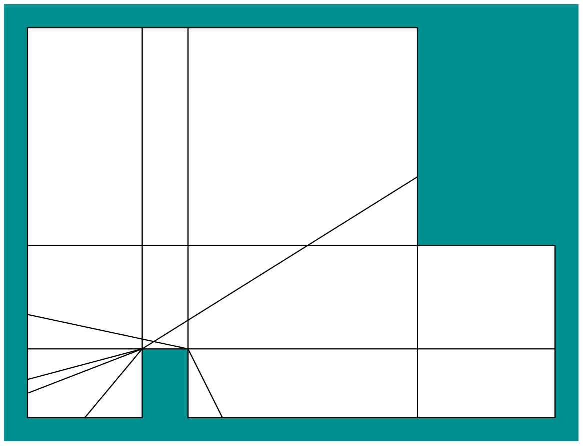


Figure 12.4: An example of the visibility cell decomposition. Inside of each cell, the visibility polygon is composed of the same edges of ∂X.

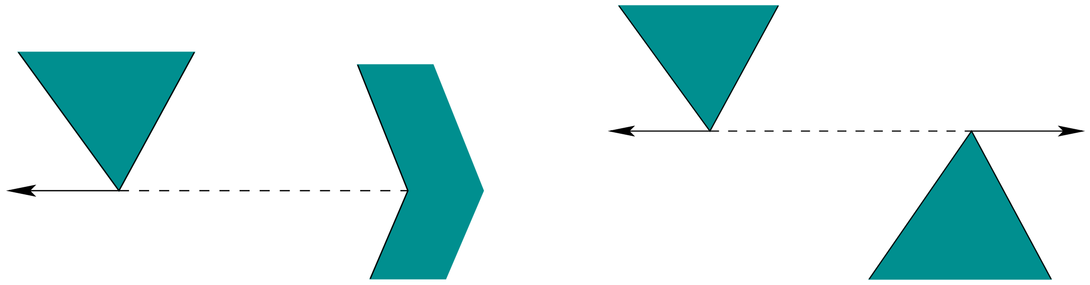


Figure 12.5: Rays are extended outward, whenever possible, from each pair of mutually visible vertices. The case on the right is a bitangent, as shown in Figure 6.10; however, here the edges extend outward instead of inward as required for the visibility graph.

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 $V(x)$ [136, 415]. 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 shown in Figure 12.6, in which rays are extended outward at every reflex vertex

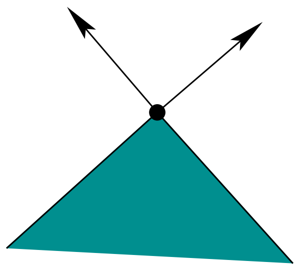


Figure 12.6: A reflex vertex: If the interior angle at a vertex is greater than π, then two outward rays are extended from the incident edges.

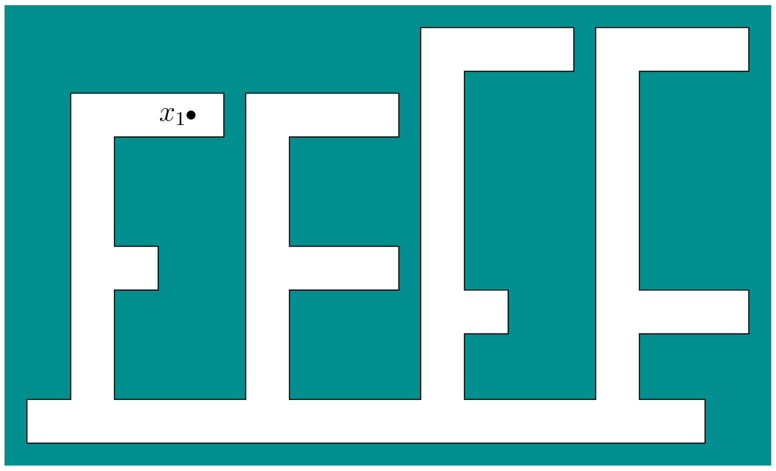


Figure 12.7: Consider this example, in which the initial state is not known [298].

(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 [415], 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 [415] 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.

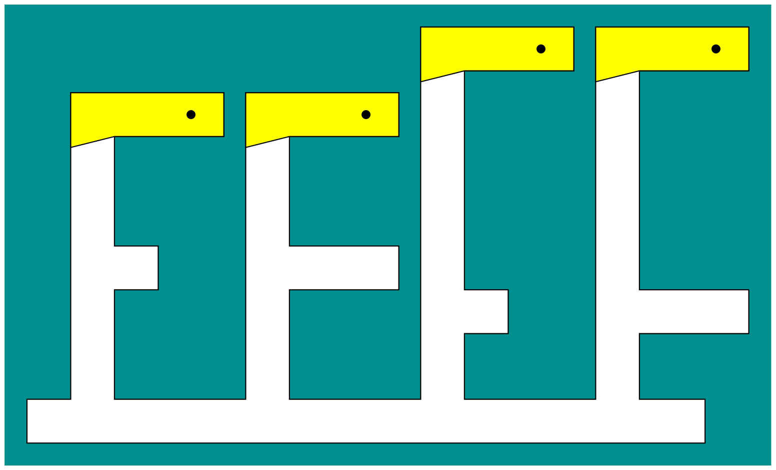


Figure 12.8: The four possible initial positions for the robot in Figure 12.7 based on the visibility sensor.

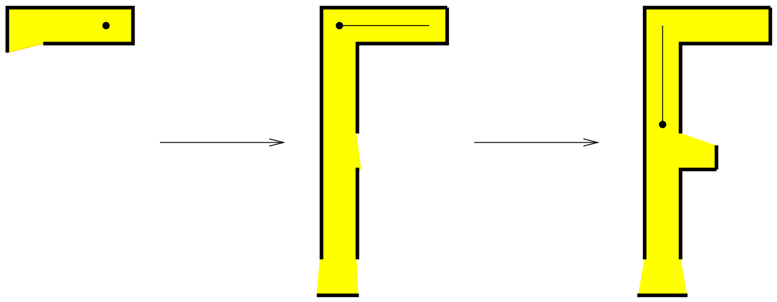


Figure 12.9: These motions completely disambiguate the state.

The following example is taken from [298].

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. ■

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 [415]. 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 $X_1(\eta_1)$

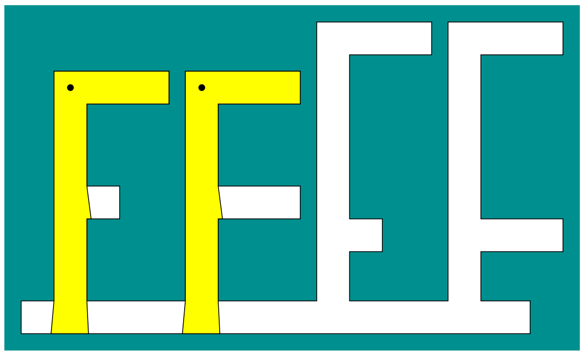


Figure 12.10: There are now only two possible states.

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 [298], 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 [298] 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 [751, 752], 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

$$y_k = h(x_k, x_{k-1}, \psi_k). \tag{12.19}$$

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 [258, 447, 622, 825, 887, 962]. 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 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 [1004].

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

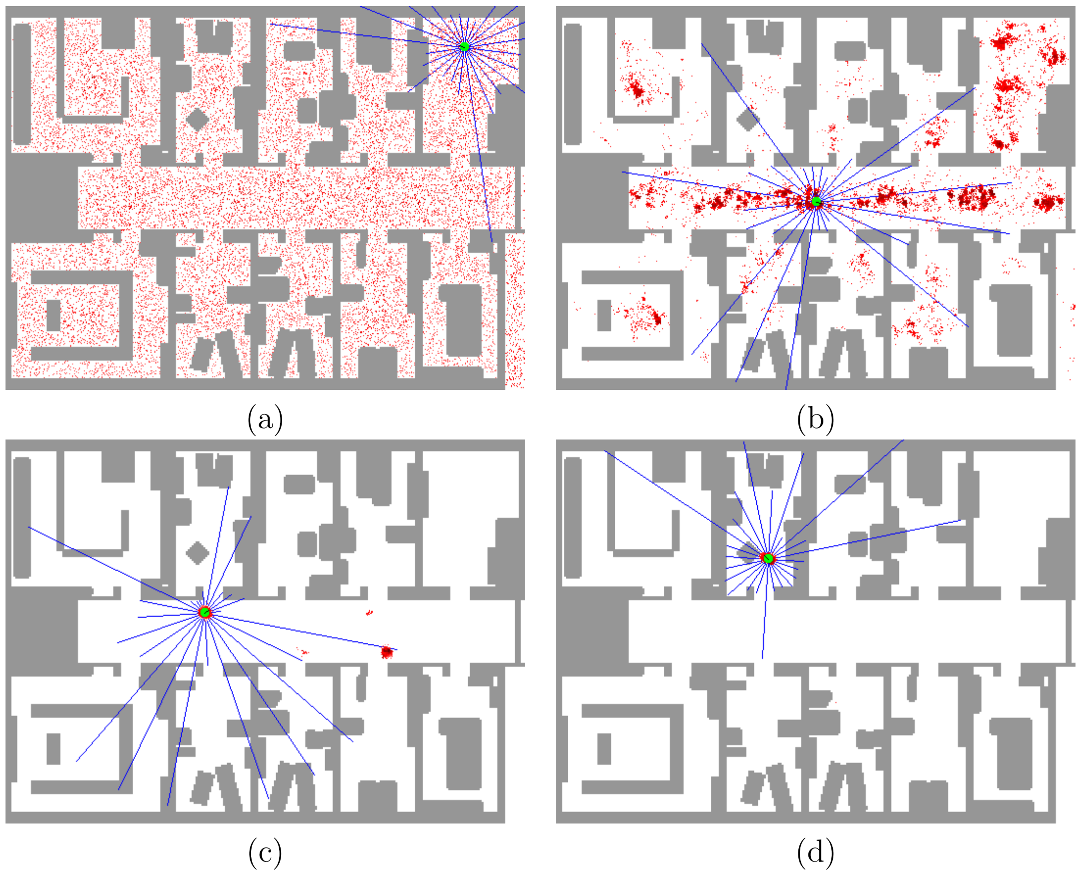


Figure 12.11: Four frames from an animation that performs probabilistic localization of an indoor mobile robot using sonars [350].

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 [483, 970]. 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 consid-

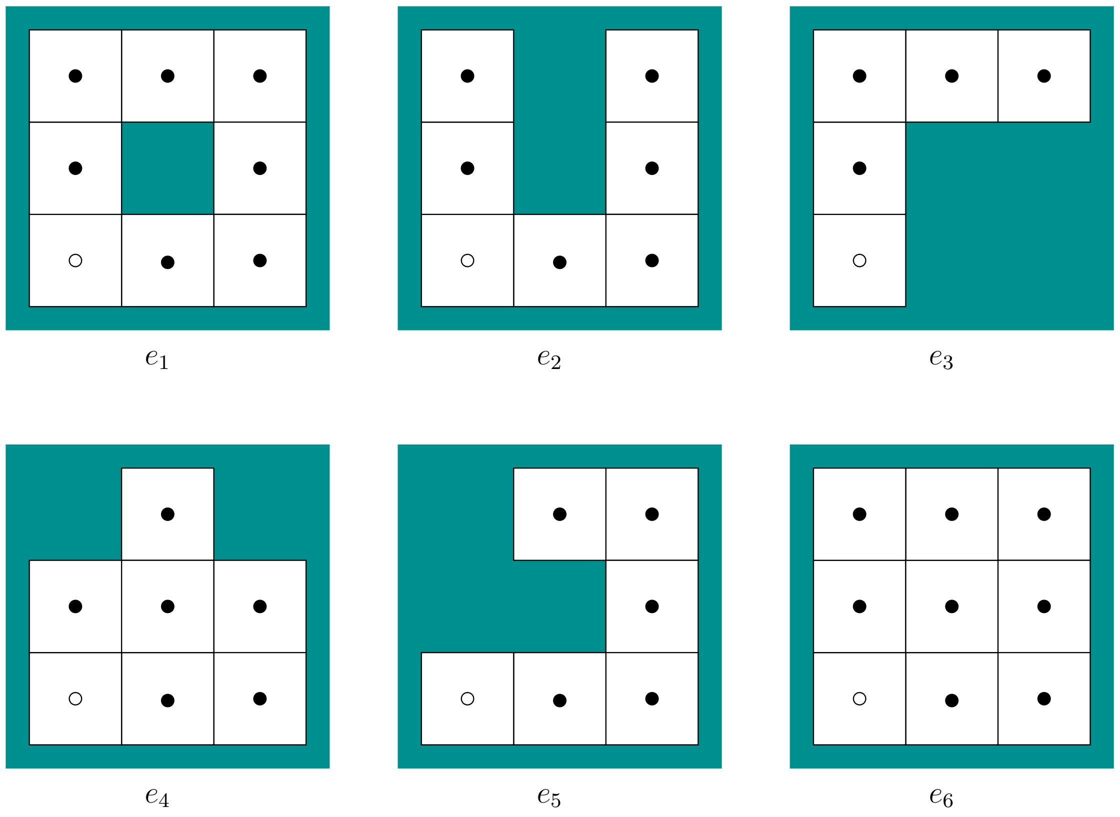


Figure 12.12: A set of possible 2D grid environments. In each case, the "up" direction represents north and the white circle represents the origin, $p = (0, 0)$.

ered 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.

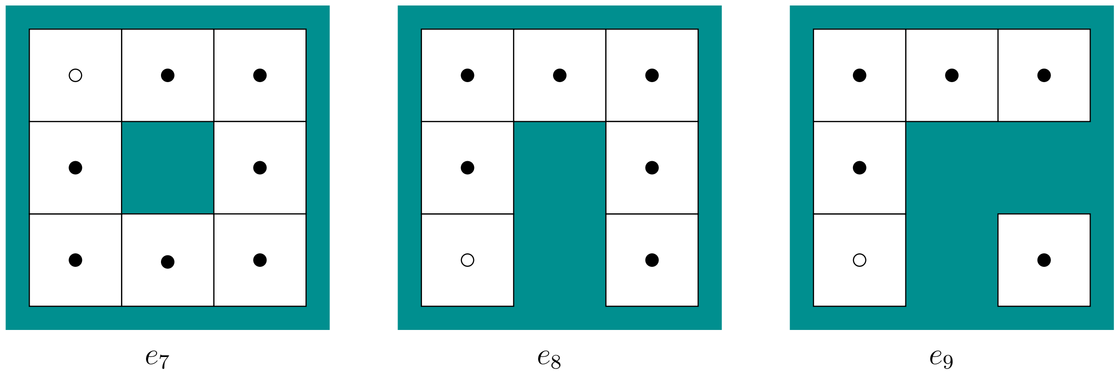


Figure 12.13: Add these environments to the set depicted in Figure 12.12. Each is essentially equivalent to an environment already given and generally does not affect the planning problem.

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 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; 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 [119]. 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 [119] 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 [286]). 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 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 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* [913], which has been used in many

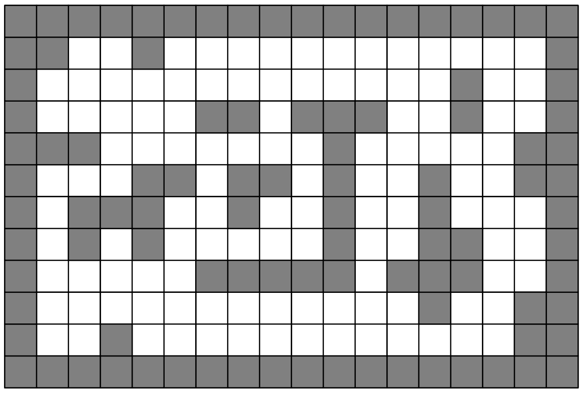


Figure 12.14: An example that has six obstacles.

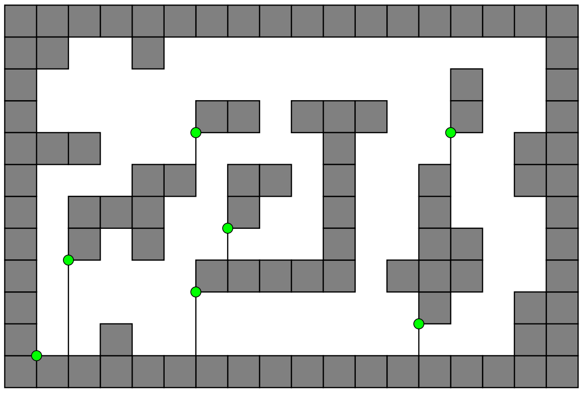


Figure 12.15: The obstacles are connected together by extending a thin obstacle downward from their unique points.

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

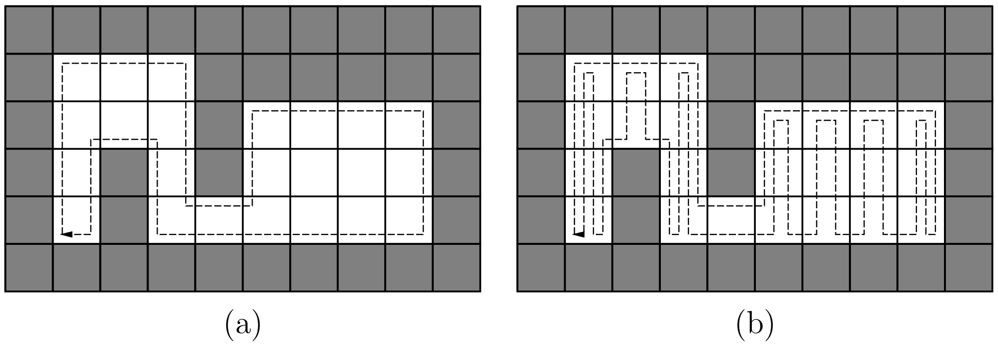


Figure 12.16: (a) A clockwise loop produced by wall following. (b) An alternative loop that visits all of the tiles in the interior.

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).

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 [913].

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.

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

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.

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

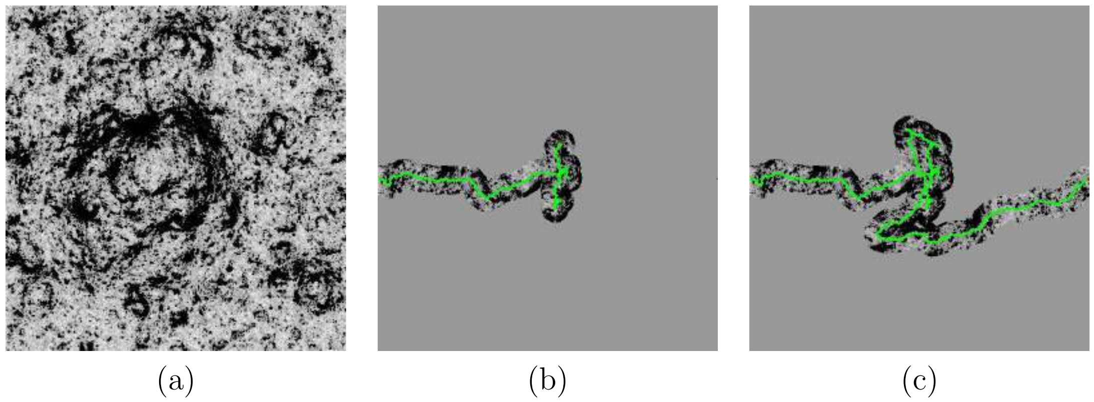


Figure 12.19: An example of executing Stentz's algorithm (courtesy of Tony Stentz).

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* [504, 667, 505] 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

[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.

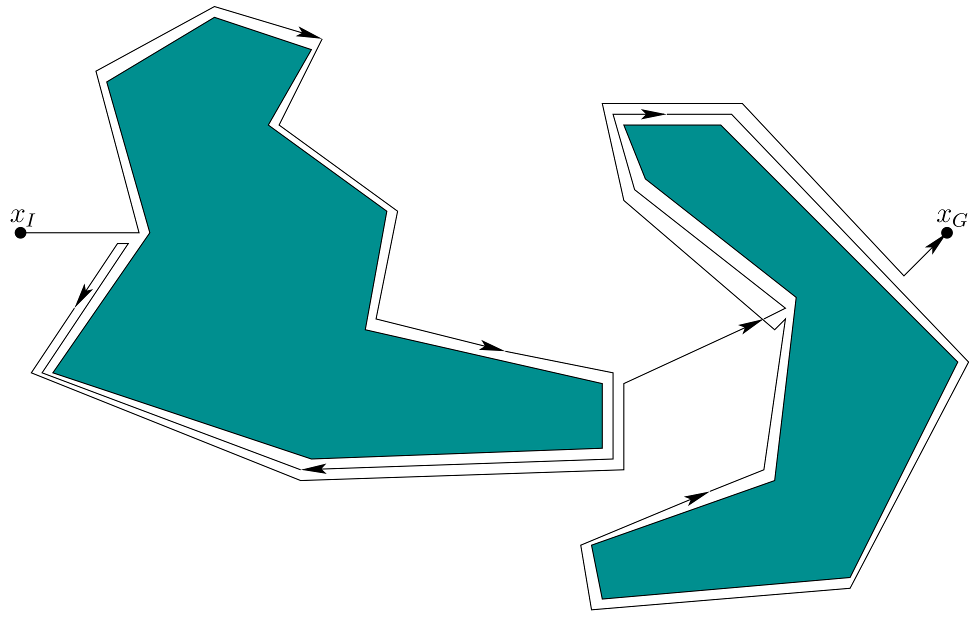


Figure 12.20: An illustration of the Bug1 strategy.

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 [667] 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. 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.

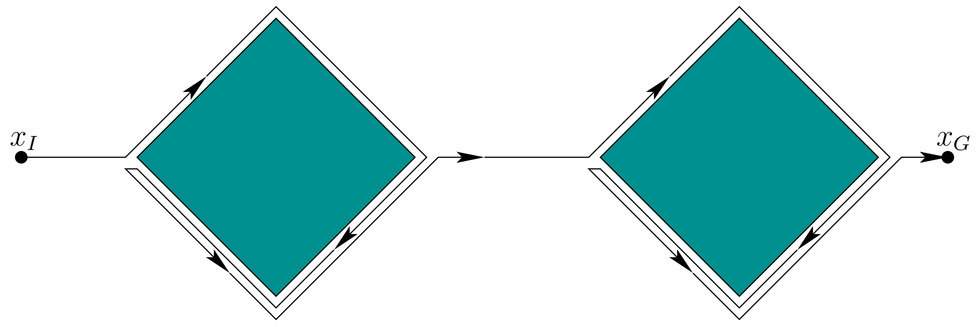


Figure 12.21: A bad example for Bug1. The perimeter of each obstacle is spanned one and a half times.

The worst case is conceptually simple to understand. The total distance traveled by the robot is no greater than

$$d + \frac{3}{2} \sum_{i=1}^{M} p_i, \tag{12.25}$$

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.

For the Bug2 strategy, the total distance traveled is no more than

$$d + \frac{1}{2} \sum_{i=1}^{M} n_i p_i, \tag{12.26}$$

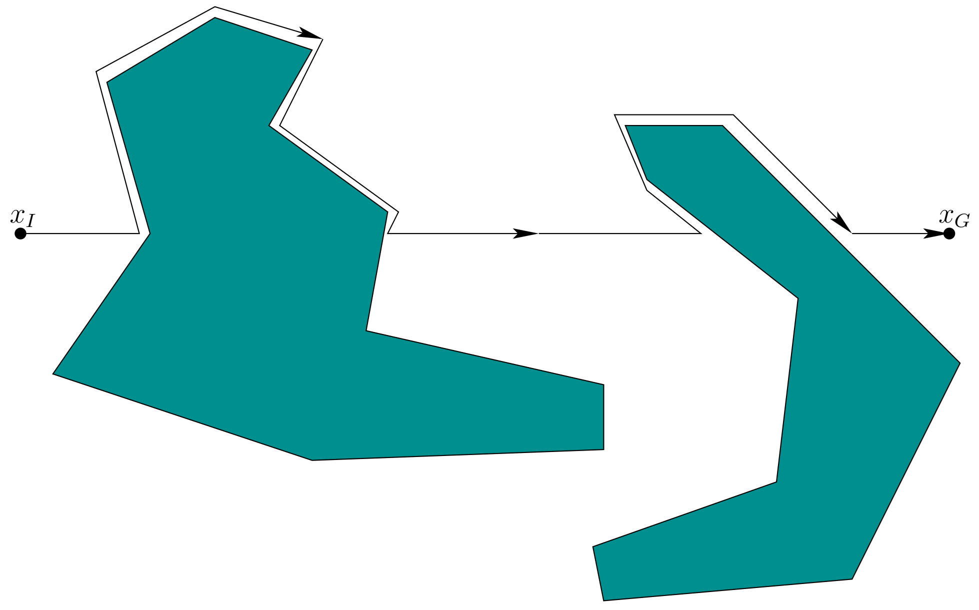


Figure 12.22: An illustration of the Bug2 strategy.

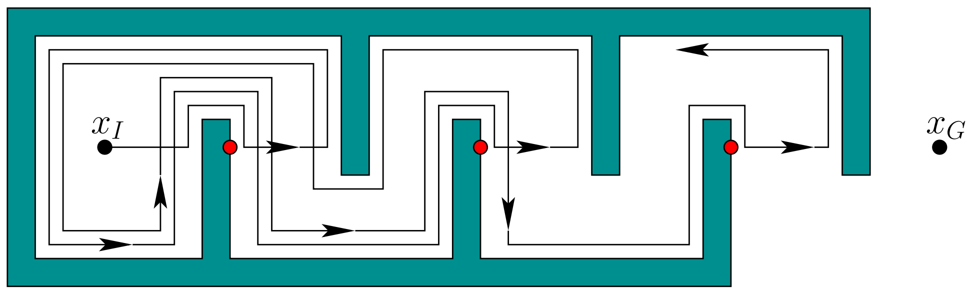


Figure 12.23: A bad case for Bug2. Only part of the resulting path is shown. Points from which the robot can leave the boundary are indicated.

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* [666] and *TangentBug* [505, 592] 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 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

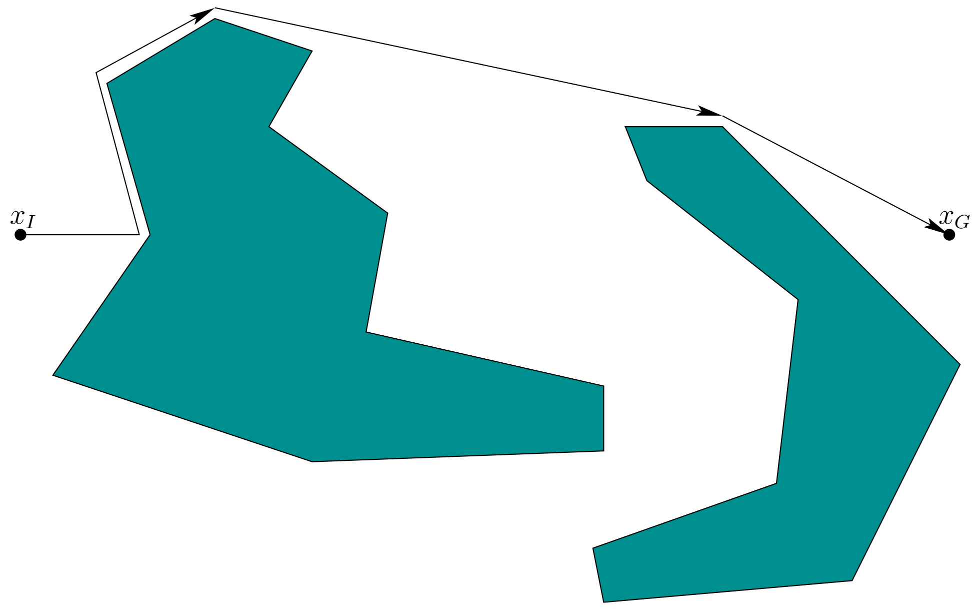


Figure 12.24: An illustration of the VisBug strategy with unlimited radius.

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 [505] 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 [592] for planetary rovers that have a limited field of view.

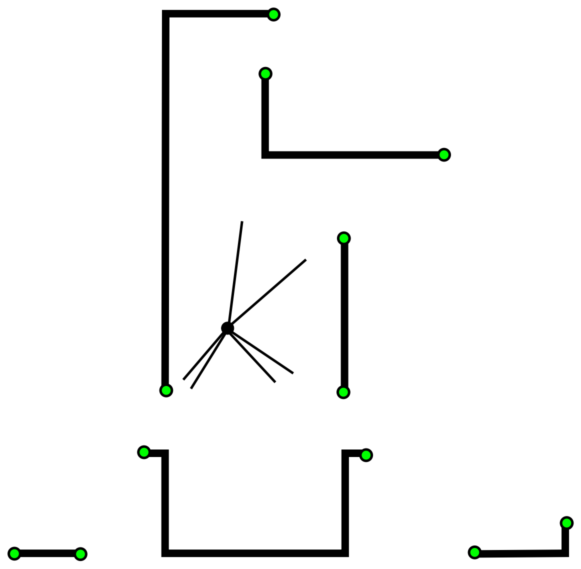


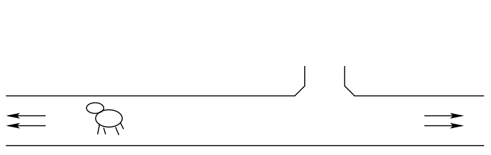
Figure 12.25: The candidate motions with respect to the range sensor are the directions in which there is a discontinuity in the depth map. The distances from the robot to the small circles are used to select the desired motion.

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 [674, 892]. 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 [375].

A nice illustration of competitive ratio analysis and issues is provided by the *lost-cow problem* [60]. As shown in Figure 12.26a, a short-sighted cow is following


(a)

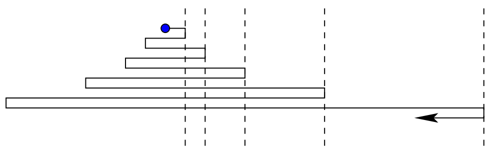

(b)

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.

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) [943, 945], 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 [944]. 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

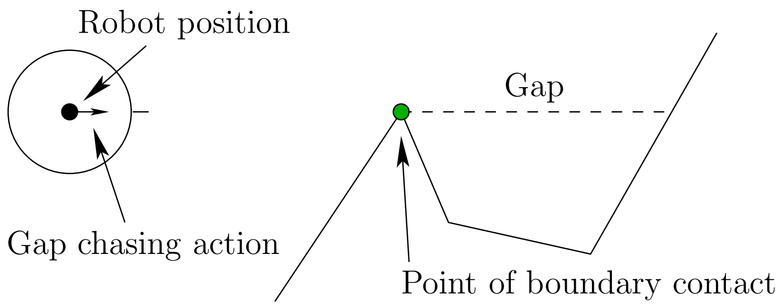


Figure 12.27: A gap-chasing action is applied, which moves the robot straight in the direction of the gap until the boundary is contacted. Once this occurs, a new part of the environment becomes visible.

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.

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

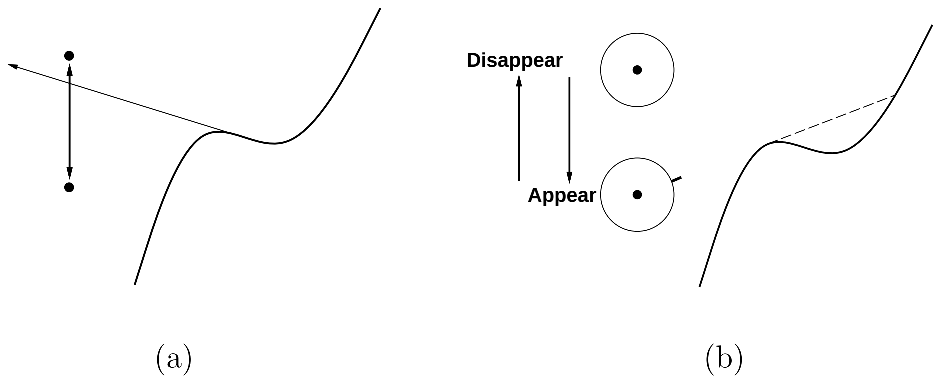


Figure 12.28: (a) The robot crosses a ray that extends from an inflectional tangent. (b) A gap appears or disappears from the gap sensor, depending on the direction.

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.

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.

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

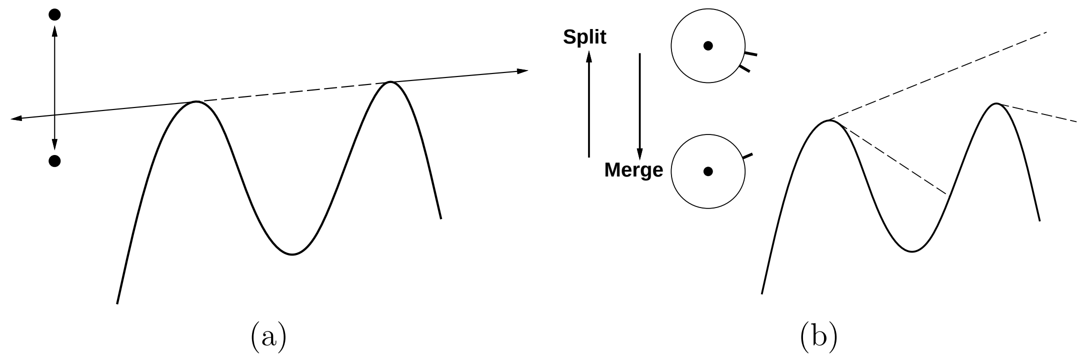


Figure 12.29: (a) The robot crosses a ray that extends from a bitangent. (b) Gaps split or merge, depending on the direction.

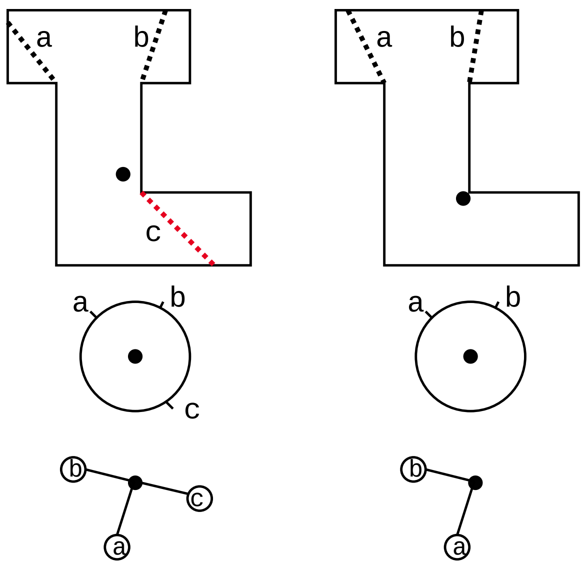


Figure 12.30: If a gap disappears, it is simply removed from the GNT.

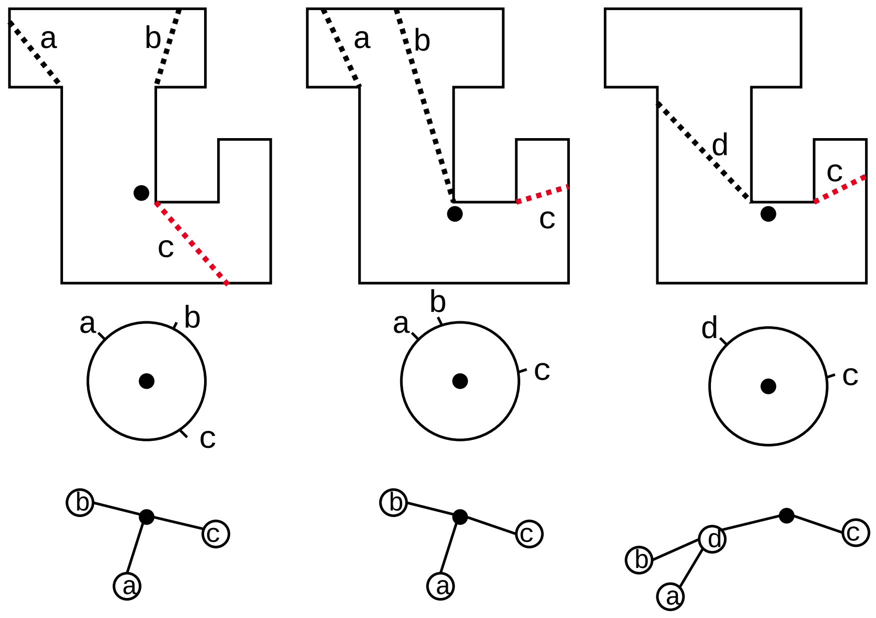


Figure 12.31: If two gaps merge, an intermediate vertex is inserted into the tree.

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 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.

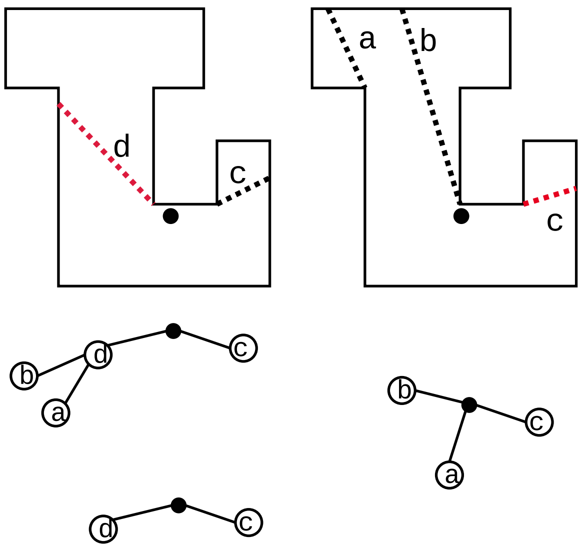


Figure 12.32: If two gaps split, the intermediate vertex is removed.

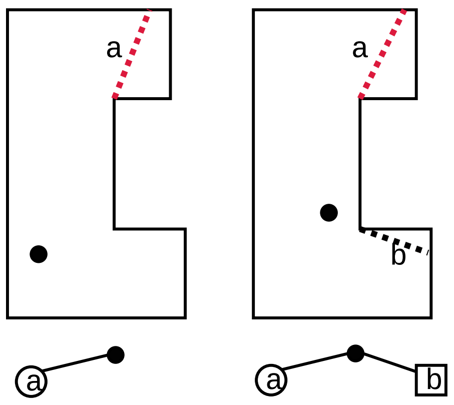


Figure 12.33: The appearance of a gap results in a primitive vertex, which is denoted by a square.

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 [945] that when the robot chases a vertex in the GNT, it precisely follows the paths of the shortest-

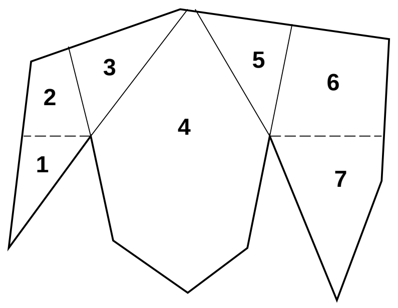


Figure 12.34: A simple environment for illustrating the gap navigation tree.

path roadmap, which was introduced in Section 6.2.4. Each pair of successive gap events corresponds to the traversal of a bitangent edge.

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.

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

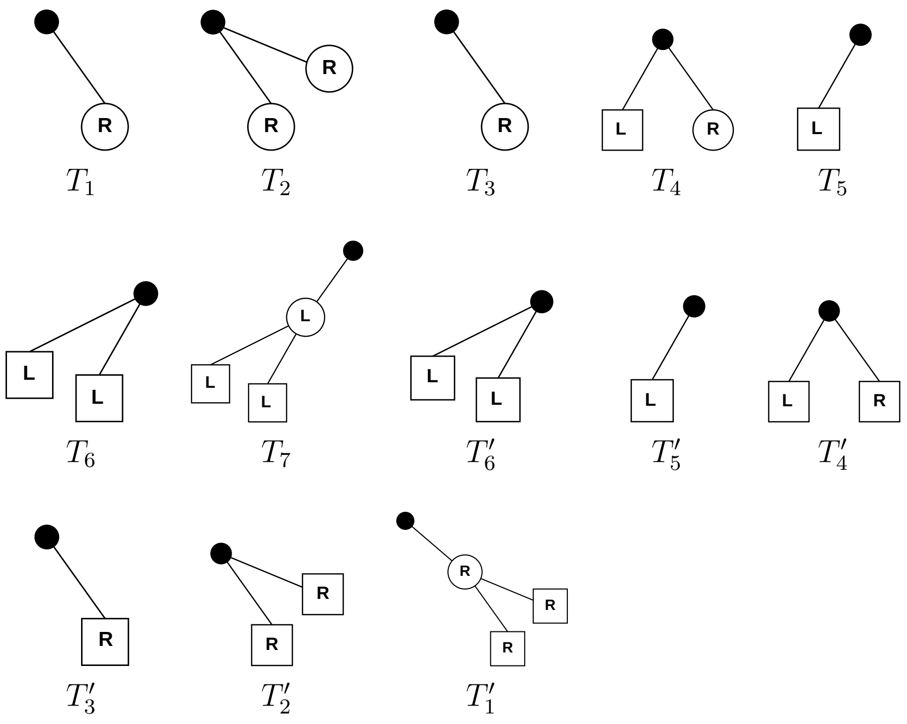


Figure 12.35: Building a representation of the environment in Figure 12.34 using the gap navigation tree. The sequence is followed from left to right. For convenience, the "R" or "L" inside of each vertex indicates whether the shadow component is to the right or left of the gap, respectively. This information is not needed by the algorithm, but it helps in understanding the representation.

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* [66, 238, 793]. 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

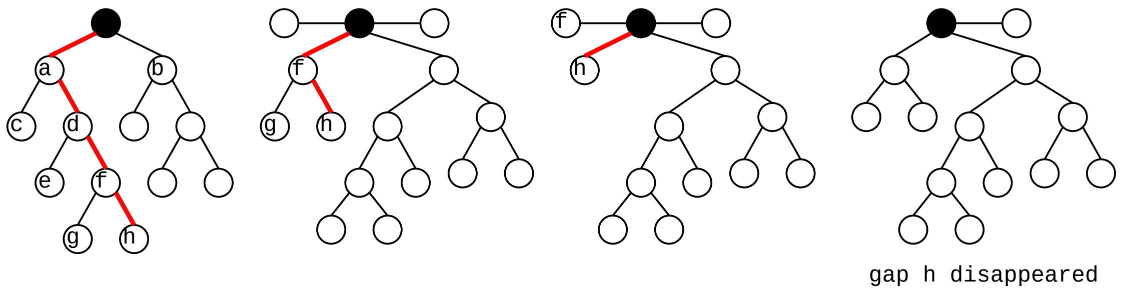


Figure 12.36: Optimal navigation to a specified part of the environment is achieved by "chasing" the desired vertex in the GNT until it disappears. This will make a portion of the environment visible. In the example, the gap labeled "h" is chased.

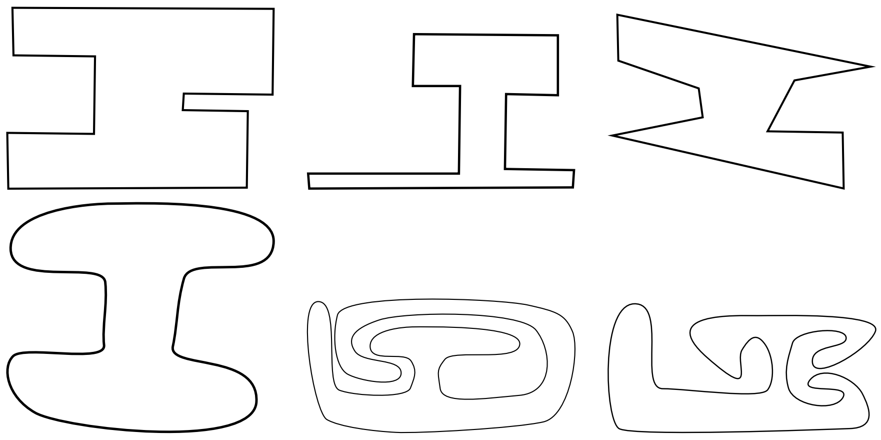


Figure 12.37: These environments yield the same GNTs and are therefore equivalent at the resolution of the derived I-space. The robot cannot measure distances and does not even care whether walls are straight or curved; it is not relevant to the navigation task. Nevertheless, it executes optimal motions in terms of the Euclidean distance traveled.

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 [106]. 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 [269, 977, 978]. In practice, it appears to work well even under cases that are not theoretically guaranteed to converge [940].

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 [941] 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 [941], the landmarks are indicated by a user as the robot travels. Classification and positioning errors can both be modeled probabilistically and

[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.

incorporated into the EM approach. Another idea that dramatically simplifies 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* [307, 685, 850]. 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* [612, 932], 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 by

[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.

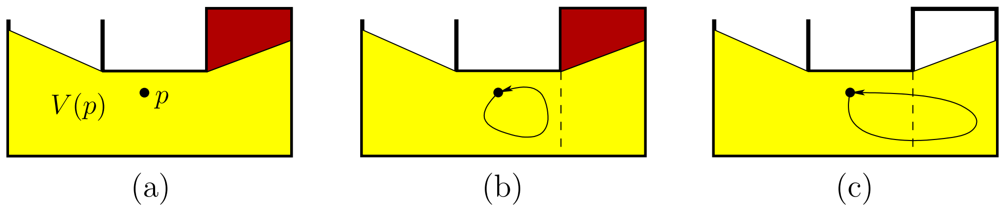


Figure 12.38: (a) Suppose the pursuer comes near the end of a contaminated corridor. (b) If the pursuer moves in a loop path, the nondeterministic I-state gradually changes, but returns to its original value. (c) However, if a critical boundary is crossed, then the nondeterministic I-state fundamentally changes.

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 [414]. 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 [773]. ■

The problem can be made more challenging by considering multiply connected environments (environments with holes). A single pursuer cannot solve any of the these problems. Determining the minimum number of pursuers required to solve

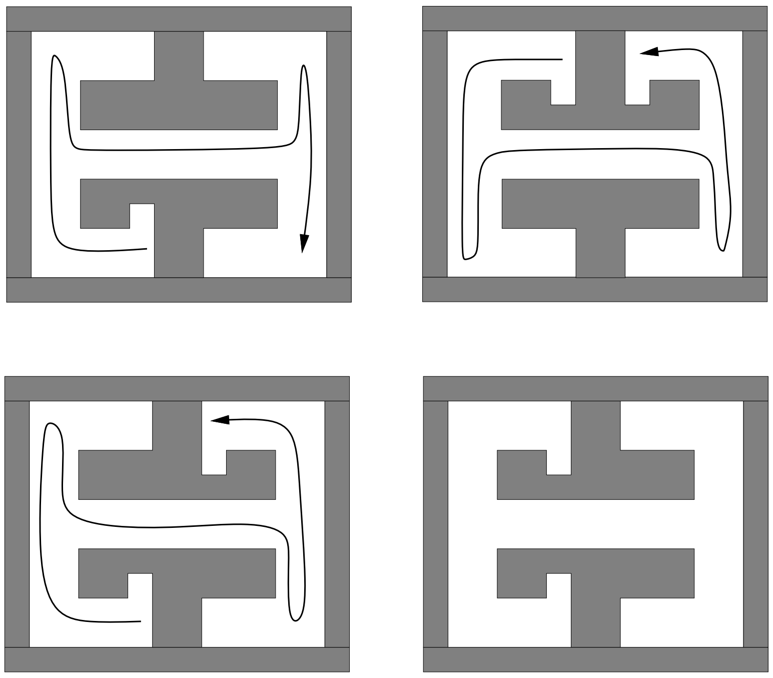


Figure 12.39: Three problems that can be easily solved with one pursuer, and a minor variant for which no solution exists.

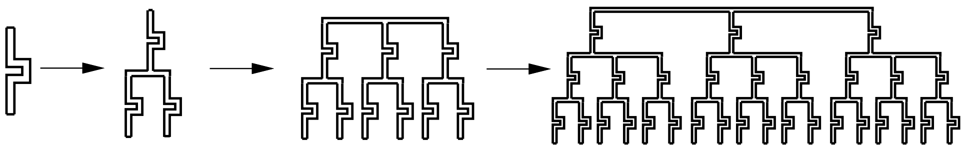


Figure 12.40: Each collection of corridors requires one more pursuer than the one before it because a new pursuer must guard the junction.

such a problem is NP-hard [414].

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.

Recall that the shadow region is the set of all points not visible from some

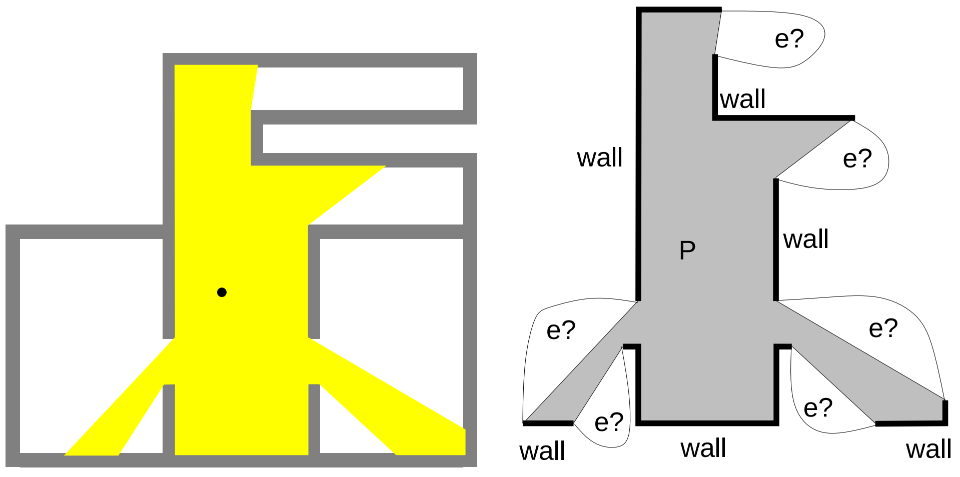


Figure 12.41: Recall Figure 11.15. Beyond each gap is a portion of the environment that may or may not contain the evader.

$p(t)$; 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 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

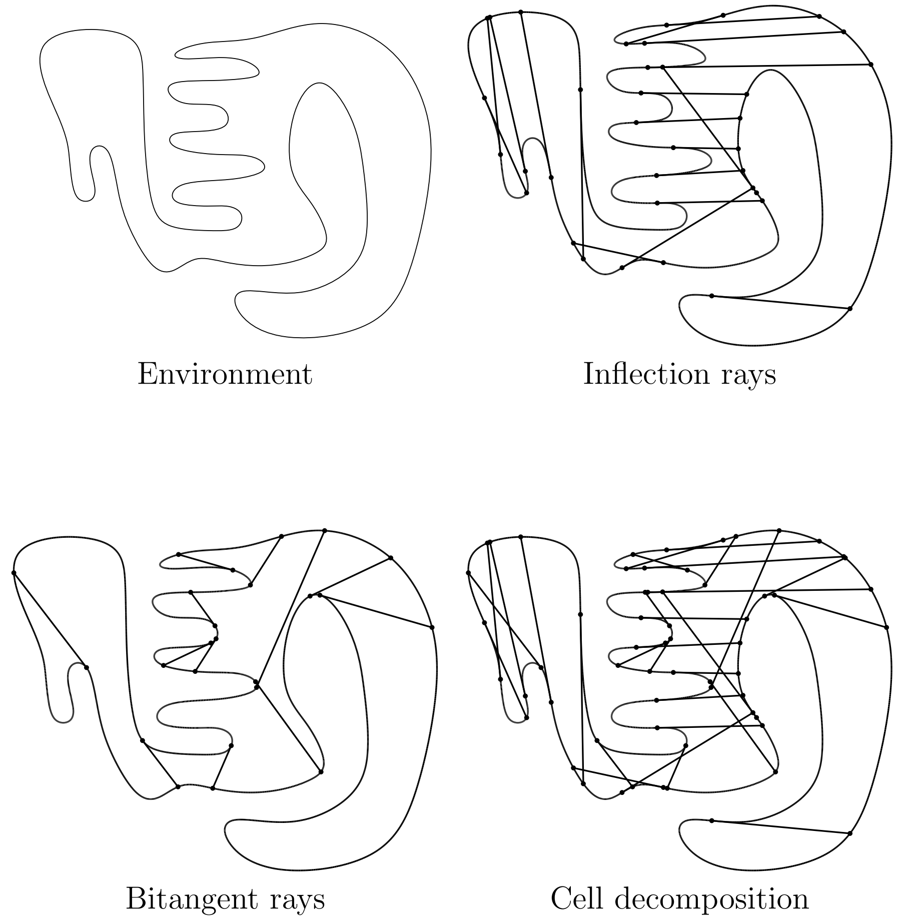


Figure 12.42: The environment is decomposed into cells based on inflections and bitangents, which are the only critical visibility events.

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 $O(n^3)$ [412]. 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 [770]. 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

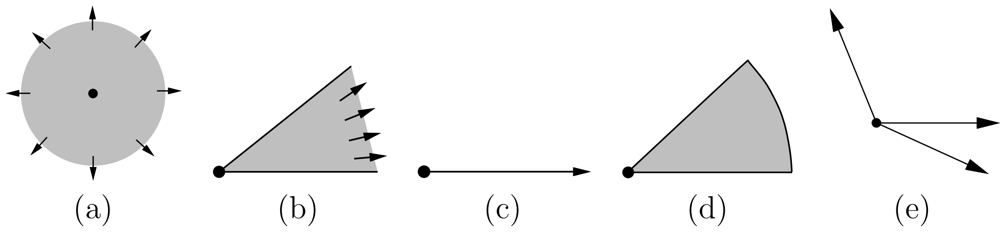


Figure 12.43: Several evader detection models: (a) omnidirectional sensing with unlimited distance; (b) visibility with a limited field of view; (c) a single visibility ray that is capable of rotating; (d) limited distance and a rotating viewing cone, which corresponds closely to a camera model; and (e) three visibility rays that are capable of rotating.

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.

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 [381]; 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 [311, 659], 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 [311, 659]. 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 [681] 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 [966].

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 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

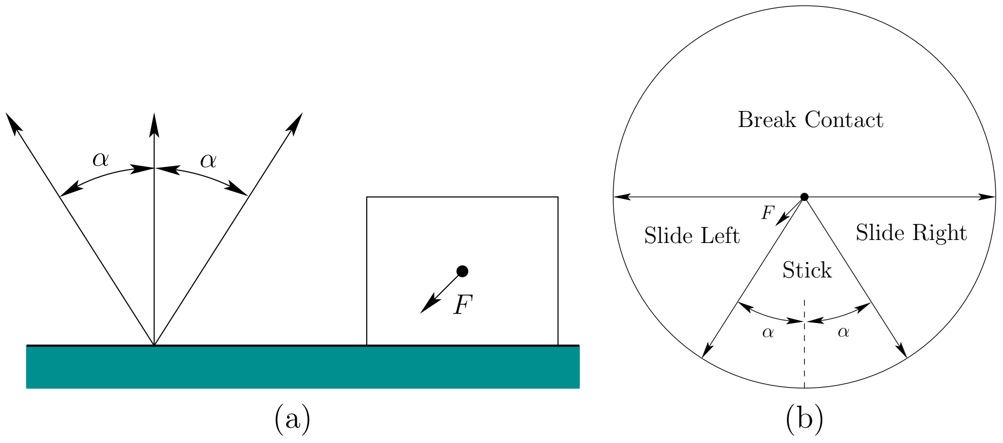


Figure 12.44: The compliant motion model. If a force F is applied by the robot at $q \in \mathcal{C}_{con}$, then it moves along the boundary only if $-F$ points outside of the friction cone.

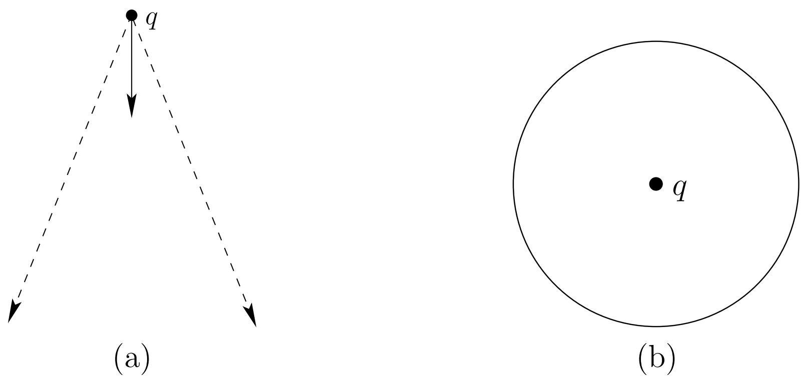


Figure 12.45: Nature interferes with both the configuration transitions and the sensor observations.

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 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 $G \subset \mathrm{cl}(\mathcal{C}_{free})$ 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

[6]Note that this open-loop plan is composed of closed-loop motion commands. This is perfectly acceptable using hierarchical modeling.

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 [281, 311, 588].

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 [311], by a reduction from the Turing machine halting problem [891], that the preimage in general is uncomputable by any algorithm. It was shown in [732] that the 3D version of preimage planning, in which the obstacles are polyhedral, is PSPACE-hard. It was then shown in [172] 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 [311, 312]. 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* [283], $\text{WB}(S)$ and $\text{SB}(S)$, can be defined, which are analogous to (10.25) and (10.26), respectively.

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.

[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.

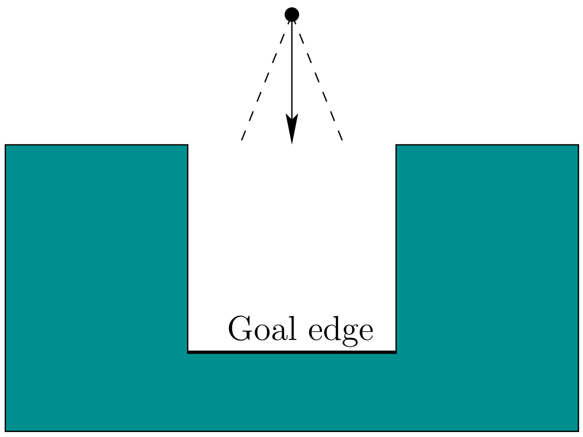


Figure 12.46: A simple example that resembles the peg-in-hole problem.

Using weak backprojections, it is possible to develop an alternative framework of *error detection and recovery* (EDR), which was introduced by Donald in [281].

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 $\mathrm{cl}(\mathcal{C}_{free})$ 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 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 [311, 312]. 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.

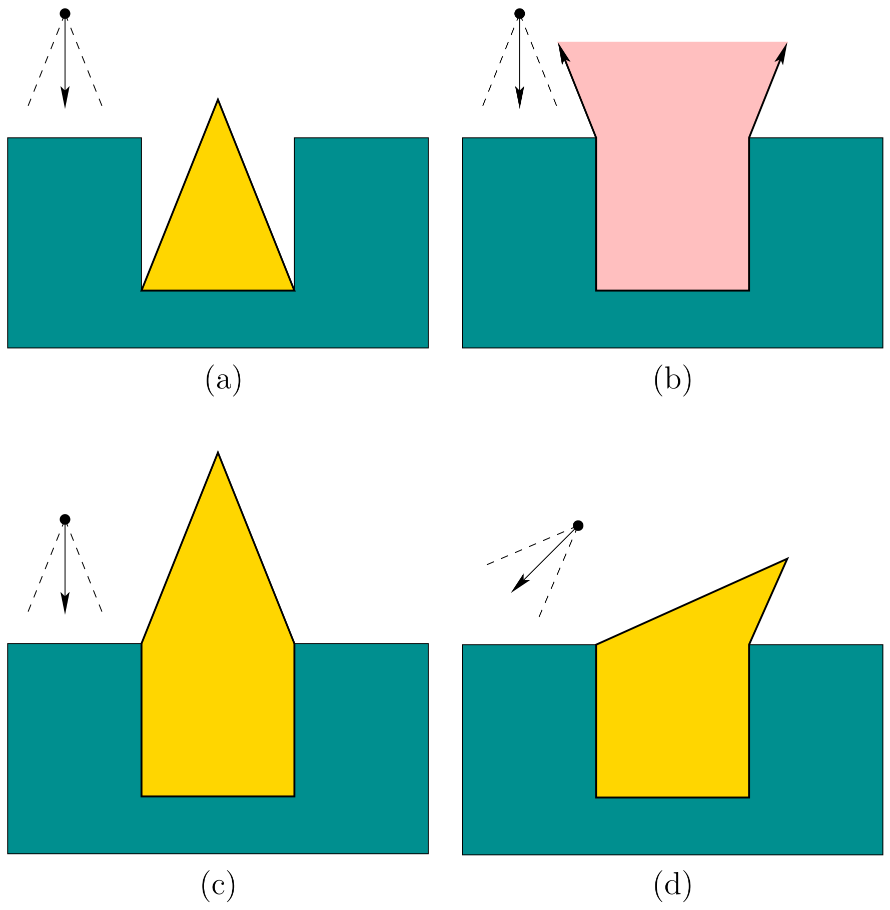


Figure 12.47: Several backprojections are shown for the peg-in-hole problem.

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 ($\alpha = 0$).

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 [283] (some details are also covered in [588]). 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

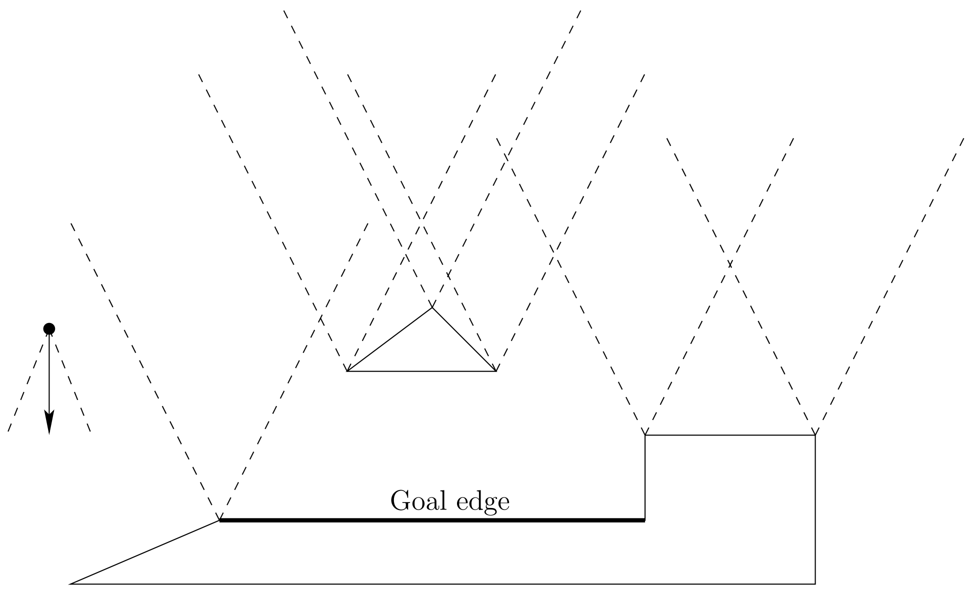


Figure 12.48: Erdmann's backprojection algorithm traces out the boundary after constructing cones based on friction.

nondirectional backprojections are given in [140, 283]. 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 [140].

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 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 [588, 590], $\mathcal{M}$ 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 [174], 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 [147],

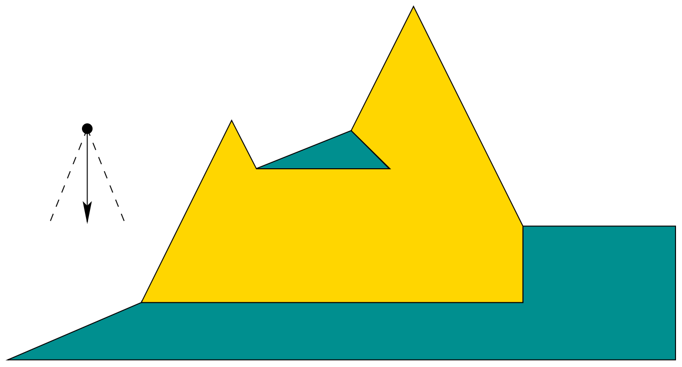


Figure 12.49: The computed backprojection. Sliding is guaranteed from the steeper edge of the triangle; hence, it is included in the backprojection. From the other top edge, sticking is possible.

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 [597, 605]. 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* [175, 321, 681], 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 [321]. 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 [394, 395, 396]; see also [681]. A Java implementation of the algorithm appears in [131]. 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 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 [175]). 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

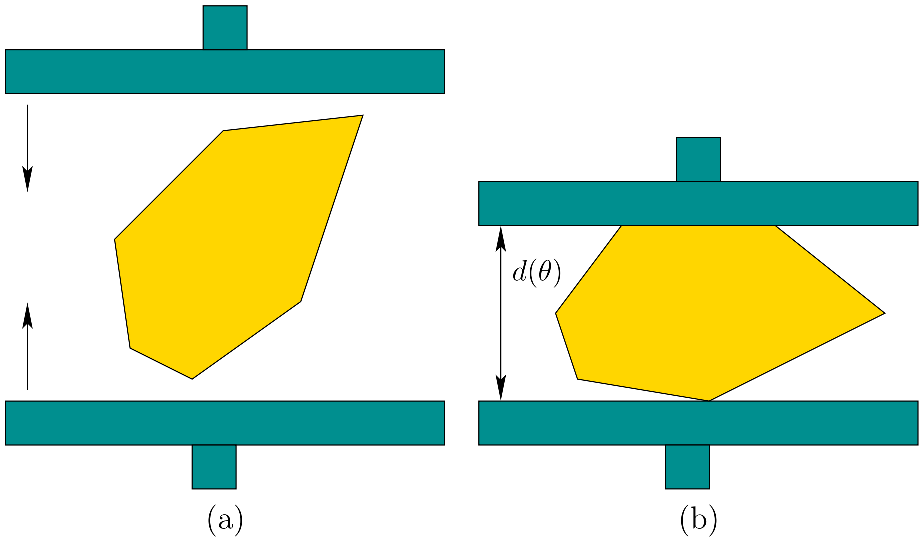


Figure 12.50: A parallel-jaw gripper can orient a part without using sensors.

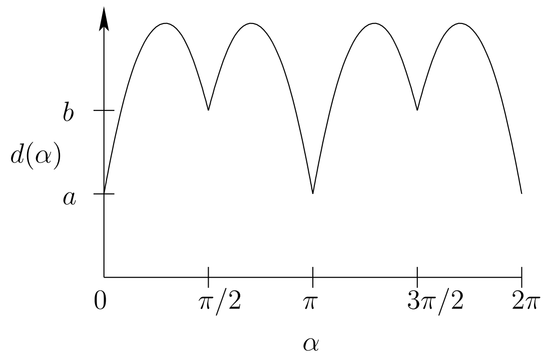
(includes only one possible orientation). If the part has symmetries, then the task is instead to determine a single symmetry class (which includes only a finite number of orientations)

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 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



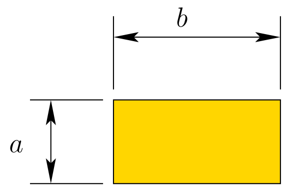


Figure 12.51: The diameter function for a rectangle.

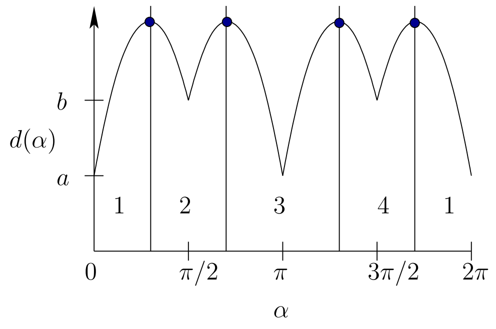


Figure 12.52: There are four regions of attraction, each of which represents an interval of orientations.

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 [395] 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 [395] 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 $u_{-k} = 0$ [395]. 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 [195] 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 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 [564, 655, 714, 899] and more recently in artificial intelligence [494, 647, 648, 737, 772, 791, 803, 805, 835, 1002, 1003]. 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 [297, 342]. The combinatorial localization presentation was based on [298, 415]. Ambiguities due to symmetry also appeared in [78]. Combinatorial localization with very little sensing is presented in [752]. For further reading on probabilistic localization, see [43, 258, 421, 447, 485, 493, 549, 621, 622, 754, 825, 887, 888, 962]. In

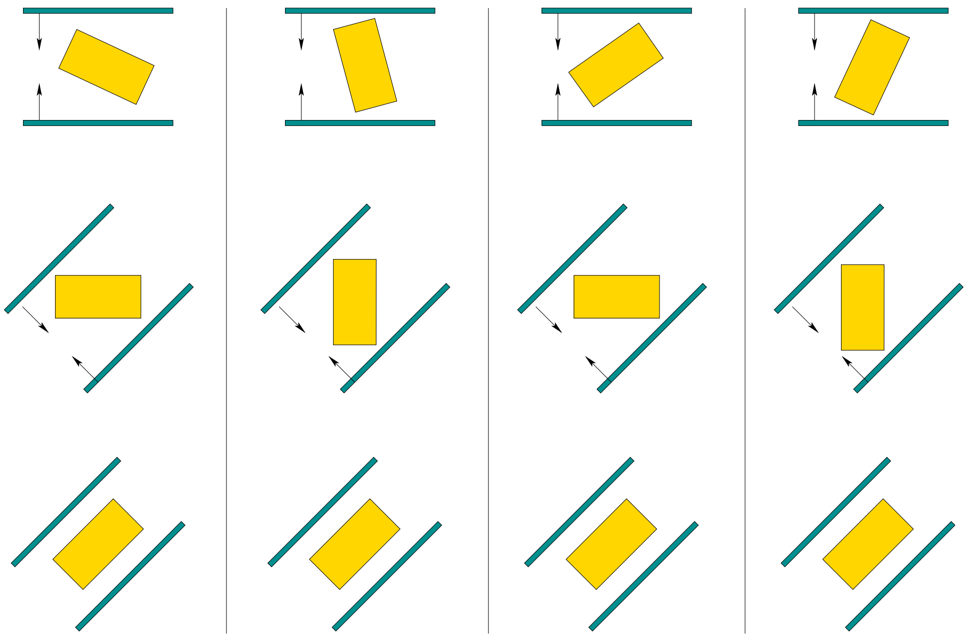


Figure 12.53: A two-step squeeze plan [395].

[935, 936], 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 [119]. The issue of distinguishability and pebbles arises again in [87, 286, 287, 668, 840, 944]. For more on competitive ratios and combinatorial approaches to on-line navigation, see [116, 260, 270, 332, 375, 507, 537, 674, 768, 811].

For more on Stentz's algorithm and related work, see [543, 913]. A multi-resolution approach to terrain exploration appears in [761]. For material on bug algorithms, see [505, 568, 592, 666, 667, 809, 882]. Related sensor-based planning work based on generalized Voronoi diagrams appears in [218, 219]; also related is [828]. Gap navigation trees were introduced in [943, 944, 945]. For other work on minimal mapping, see [484, 824, 873]. Landmark-based navigation is considered in [369, 590, 884].

There is a vast body of literature on probabilistic methods for mapping and localization, much of which is referred to as SLAM [942]; see also [182, 221, 275, 717, 771, 982]. One of the earliest works is [897]. An early application of dynamic programming in this context appears in [584]. A well-known demonstration of SLAM techniques is described in [159]. For an introduction to the EM algorithm, see [106]; its convergence is addressed in [269, 977, 978]. For more on mobile robotics in general, see [134, 296].

The presentation of Section 12.4 was based mainly on [414, 612]. Pursuit-evasion problems in general were first studied in differential game theory [59, 422, 477]. Pursuit-evasion in a graph was introduced in [773], and related theoretical analysis appears in [105, 580, 715]. Visibility-based pursuit-evasion was introduced in [932], and the first complete algorithm appeared in [612]. An algorithm that runs in $O(n^2)$ for a single pursuer in a simple polygon was given in [770]. Variations that consider curved

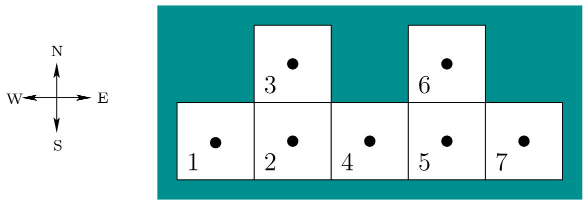


Figure 12.54: An environment for grid-based localization.

environments, beams of light, and other considerations appear in [208, 254, 304, 603, 618, 745, 889, 890, 931, 933, 981]. Pursuit-evasion in three dimensions is discussed in [614]. For versions that involve minimal sensing and no prior given map, see [416, 503, 809, 840, 988]. The problem of visually tracking a moving target both with [81, 401, 402, 602, 723, 728] and without [323, 470, 869] obstacles is closely related to pursuit-evasion. For a survey of combinatorial algorithms for computing visibility information, see [756]. Art gallery and sensor placement problems are also related [141, 755, 874]. The bitangent events also arise in the visibility complex [796] and in aspect graphs [782], which are related visibility-based data structures.

Section 12.5 was inspired mostly by the works in [283, 311, 321, 396, 659, 967]. Many works are surveyed in [681]. A probabilistic version of preimage planning was considered in [148, 149, 605]. Visual preimages are considered in [349]. Careful analysis of manipulation uncertainty appears in [145, 146]. For more on preimage planning, see [588, 590]. 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 [281, 284, 285]. Compliant motions are also considered in [140, 283, 486, 678, 680, 776]. The effects of friction in the C-space are studied in [316]. For more work on orienting parts, see [175, 322, 394, 395, 810, 969]. For more forms of nonprehensile manipulation, see [12, 14, 110, 318, 670, 671, 921]. A humorous paper, which introduces the concept of the "principle of virtual dirt," is [679]; the idea later appears in [839] and in the Roomba autonomous vacuum cleaner from the iRobot Corporation.

Exercises

1. For the environment in Figure 12.1a, give the nondeterministic I-states for the action sequence $(\mathrm{L}, \mathrm{L}, \mathrm{F}, \mathrm{B}, \mathrm{F}, \mathrm{R}, \mathrm{F}, \mathrm{F})$, 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.

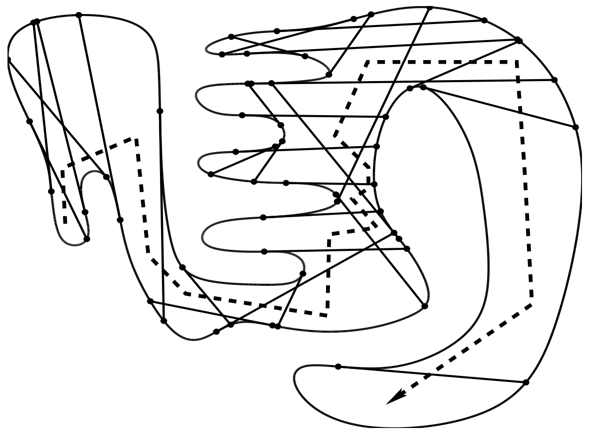


Figure 12.55: A path followed by the robot in an initially unknown environment. The robot finishes in the lower right.

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 $(\mathrm{F}, \mathrm{R}, \mathrm{B}, \mathrm{F}, \mathrm{L}, \mathrm{L}, \mathrm{F})$.

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?

   (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.

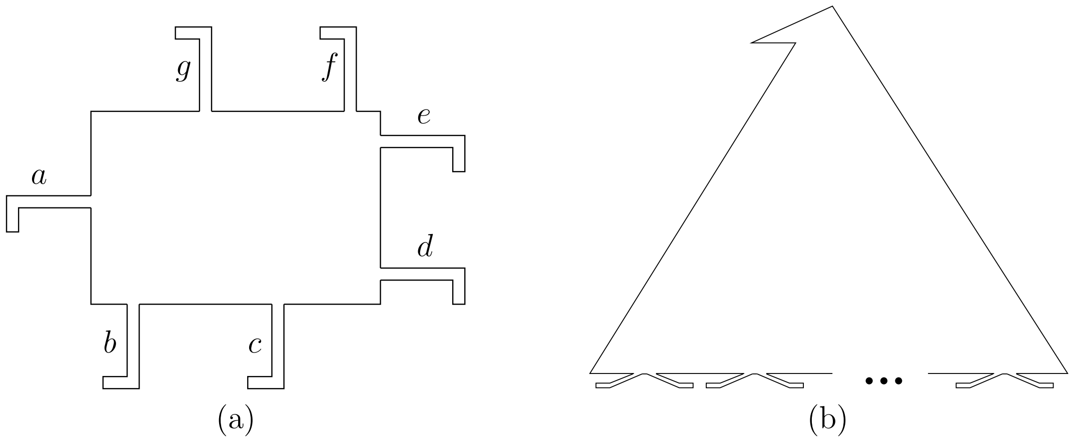


Figure 12.56: Two pursuit-evasion problems that involve recontamination.

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.