TUTORIAL

Motion Planning

Part II: Wild Frontiers

By Steven M. LaValle

Here, we give the Part II of the two-part tutorial. Part I emphasized the basic problem formulation, mathematical concepts, and the most common solutions. The goal of Part II is to help you understand current robotics challenges from a motion-planning perspective.

Limitations of Path Planning

The basic problem of computing a collision-free path for a robot among known obstacles is well understood and reasonably well solved; however, deficiencies in the problem formulation itself and the demand of engineering challenges in the design of autonomous systems raise important questions and topics for future research.

The shortcomings of basic path planning are clearly visible when considering how the computed path is typically used in a robotic system. It has been known for decades that effective autonomous systems must iteratively sense new data and act accordingly; recall the decades-old sense–plan–act (SPA) paradigm. Figure 1 shows how a computed collision-free path $\tau : [0, 1] \rightarrow \mathcal{C}_{\text{free}}$ is usually brought into alignment with this view by producing a feedback control law. Step 1 produces τ using a path-planning algorithm. Step 2 then smoothens τ to produce $\sigma : [0, 1] \rightarrow \mathcal{C}_{\text{free}}$, a path that the robot can actually follow. For example, if the path is piecewise linear, then a carlike mobile robot would not be able to turn sharp corners. Step 3 reparameterizes σ to make a trajectory $\tilde{q} : [0, t_f] \rightarrow \mathcal{C}_{\text{free}}$ that nominally satisfies the robot dynamics (for example, acceleration bounds). In Step 4, a state-feedback control law that tracks $\tilde{q}$ as closely as possible during execution is designed. This results in a policy or plan, $\pi : X \rightarrow U$. The domain X is a state space (or phase space), and U is an action space (or input space). These sets appear in the definition of the control system that models the robot: $\dot{x} = f(x, u)$ in which $x \in X$ and $u \in U$.

One clear problem in this general framework is that a later step might not succeed due to an unfortunate, fixed choice in an earlier step. Even if it does succeed, the produced solution may be horribly inefficient. This motivates planning under differential constraints, which essentially performs

Digital Object Identifier 10.1109/MRA.2011.941635
Date of publication: 14 June 2011

steps 1 and 2 or steps 1–3 in one shot; see the "Differential Constraints" section. The eventual need for feedback in Step 4 motivates the direct computation of a feedback plan, which is covered in the "Feedback Motion Planning" section.

Another issue with the framework in Figure 1, which is perhaps more subtle, is that this fixed decomposition of the overall problem of getting a robot to navigate has artificially inflated the information requirements. The framework requires that powerful sensors, combined with strong prior knowledge, must be providing accurate state estimates at all times, including the robot configuration, velocity components, and obstacle models. This unfortunately overlooks a tremendous opportunity to reduce the overall system complexity by sensing just enough information to complete the task. In this case, a plan is $\pi : \mathcal{I} \rightarrow U$ instead of $\pi : X \rightarrow U$, in which $\mathcal{I}$ is a specific information space that can be derived from sensor measurements and from which a complete reconstruction of the state $x(t) \in X$ is either impossible or undesirable. The "Sensing Uncertainty" section introduces sensing, filtering, and planning from this perspective: The state cannot be fully estimated, but tasks are nevertheless achieved.

Differential Constraints

In this section, it may help to imagine that the C-space $\mathcal{C}$ is $\mathbb{R}^n$ to avoid the manifold technicalities from Part I. In the models and methods of Part I, it was assumed that a path can be easily determined between any two configurations in the absence of obstacles. For example, vertices in the trapezoidal decomposition approach are connected by a straight line segment in the collision-free region, $\mathcal{C}_{\text{free}}$. This section complicates the problem by introducing differential constraints, which restrict the allowable velocities at each point in $\mathcal{C}_{\text{free}}$. These are local constraints in contrast to the global constraints that arise due to obstacles.

Differential constraints naturally arise from the kinematics and dynamics of robots. Rather than treating them as an afterthought, this section discusses how to directly model and incorporate them into the planning process. In this way, a path is produced that already satisfies the constraints.

Modeling the Constraints

For simplicity, suppose $\mathcal{C} = \mathbb{R}^2$. Let $\dot{q} = (\dot{x}, \dot{y})$ denote a velocity vector in which $\dot{x} = dx/dt$ and $\dot{y} = dy/dt$. Starting from any point in $\mathbb{R}^2$, say $(0, 0)$, consider what paths can possibly be produced by integrating the velocity: $\tilde{q}(t) = \int_0^t \dot{q}(s)ds$. Here, $\dot{q}$ is interpreted as a function of time. If no constraints are imposed on $\dot{q}$ (other than requirements for integrability), then the trajectory $\tilde{q}$ is virtually unrestricted. If, however, we require $\dot{x} > 0$, then the only trajectories for which x monotonically increases are allowed. If we further constrain it so that $0 < \dot{x} \leq 1$, then the rate at which x increases is bounded. If time was measured in seconds and $\mathbb{R}^2$ in meters, then $\tilde{q}$ must cause travel in the x direction with a rate of no more than 1 m/s.

More generally, we want to express a set of allowable velocity vectors $\dot{q} = (\dot{x}, \dot{y})$ for every $q = (x, y) \in \mathbb{R}^2$. Rather than write a set-valued function with domain $\mathbb{R}^2$, a more compact, convenient method is to define a function f that yields $\dot{q}$ as a function of q and a new parameter u:

$$\dot{q} = f(q, u). \tag{1}$$

This results in a velocity-valued function called the *configuration-transition equation*, which indicates the required velocity vector, given q and u. The parameter u is called an *action* (or *input*) and is chosen from a predetermined action space U. Since f is a function of two variables, there are two convenient interpretations by holding each variable fixed: 1) if q is held fixed, then each $u \in U$ produces a possible velocity $\dot{q}$ at q; in other words, u parameterizes the set of possible velocities; 2) if u is fixed, then f specifies a velocity at every q; this results in a vector field over $\mathcal{C}$.

For a common example of the configuration-transition equation, Figure 2 shows a carlike robot that has the C-space of a rigid body in the plane: $\mathcal{C} = \mathbb{R}^2 \times S^1$. The configuration vector is $q = (x, y, \theta)$. Imagine that the car drives around slowly (so that dynamics are ignored) in an infinite parking lot. Let ϕ be the steering angle of the front tires, as shown in Figure 2. If driven forward, the car will roll along a circle of radius ρ. Note that it is impossible to move the center of the rear axle laterally because the rear wheels would skid instead of roll. This induces the constraint $\dot{y}/\dot{x} = \tan\theta$. This constraint, along with another due to the steering angle, can be converted into the following form (see [12, section 13.1.2.1]):

$$\begin{aligned} \dot{x} &= u_s \cos\theta \\ \dot{y} &= u_s \sin\theta \\ \dot{\theta} &= \frac{u_s}{L} \tan u_\phi, \end{aligned} \tag{2}$$

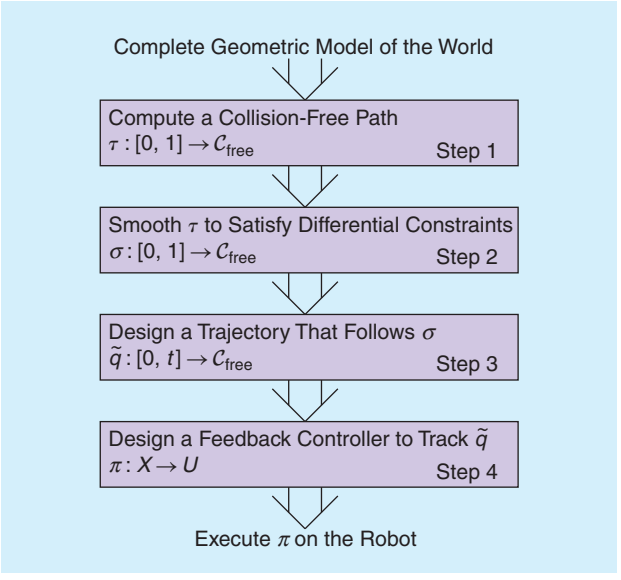

Figure 1. The long road to using a computed collision-free path. Note that complete, perfect knowledge of the robot and obstacles enters in, and sensors are utilized only during the final execution.

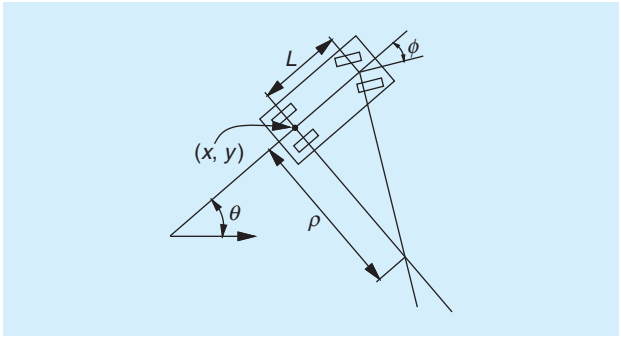

Figure 2. A simple car has three degrees of freedom, but the velocity space at any configuration is only two dimensional.

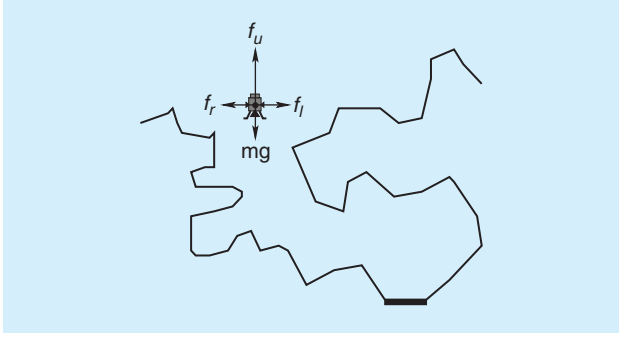

Figure 3. Attempt to land a lunar spacecraft with three orthogonal thrusters that can be switched on or off. The 2-D C-space leads to a four-dimensional state space.

in which $u = (u_s, u_\phi) \in U$ is the action; u_s is the forward speed, and u_ϕ is the steering angle. Now U must be defined. Usually, the steering angle is bounded by some $\phi_{\max} < \pi/2$ so that $|u_\phi| \leq \phi_{\max}$. For the possible speed value u_s, a simple bound is often made. For example $|u_s| \leq 1$ or equivalently, $U = [-1, 1]$, produces a car that can travel no faster than unit speed. A finite set of values is often used for planning problems that are taking into account only the kinematic constraints due to rolling wheels. Setting $U = \{-1, 0, 1\}$ produces what is called the *Reeds-Shepp car*, which can travel forward at unit speed, reverse at unit speed, or stop. By further restricting so that $U = \{0, 1\}$, the Dubins car is obtained, which can only travel forward or stop (this car cannot be parallel parked).

Numerous other models are widely used. Equations similar to (2) arise for common differential drive robots (for example, Roombas). Other examples include a car pulling one or more trailers, three-dimensional (3-D) ball rolling in the plane, and simple aircraft models.

Now consider how the planning problem has changed. The transition equation f becomes the interface through which solution paths must be constructed. We must compute some function $\tilde{u} : [0, t] \rightarrow U$ that indicates how to apply actions, so that upon integration, the resulting trajectory $\tilde{q} : [0, t] \rightarrow \mathcal{C}$ will satisfy: $\tilde{q}(0) = q_I$, $\tilde{q}(t) = q_G$, and $\tilde{q}(t') \in \mathcal{C}_{\text{free}}$ for all $t' \in [0, t]$. Intuitively, we now have to steer the configuration into the goal, thereby losing the freedom of moving in any direction.

Moving to the State Space

The previous section considered what are called *kinematic* differential constraints because they arise from the geometry of rigid body interactions in world. More broadly, we must consider the differential constraints that account for both kinematics and dynamics of the robot. This allows velocity and acceleration constraints to be appropriately modeled, usually resulting in a transition equation of the form $\ddot{q} = h(q, \dot{q}, u)$ in which $\ddot{q} = d\dot{q}/dt$. Differential equations that involve higher-order derivatives are usually more difficult to handle; therefore, we employ a simple trick that converts them into a form involving first derivatives only but at the expense of introducing more variables and equations.

The simplest and most common case is called the *double integrator*. Let $\mathcal{C} = \mathbb{R}$ and let $\ddot{q} = h(q, \dot{q}, u)$ be the special case $\ddot{q} = u$. This corresponds, for example, to a Newtonian point mass accelerating due to an applied force (recall Newton's second law, $F = ma$; here, $\ddot{q} = a$ and $u = F/m$). We now convert h into two first-order equations. Let $X = \mathbb{R}^2$ denote a state space, with coordinates $(x_1, x_2) \in X$. Let $x_1 = q$ and $x_2 = \dot{q}$. Note that $\dot{x}_1 = x_2$ and, using $\ddot{q} = u$, we have $\dot{x}_2 = u$. Using vector notation $\dot{x} = (\dot{x}_1, \dot{x}_2)$ and $x = (x_1, x_2)$, we can interpret $\dot{x}_1 = x_2$ and $\dot{x}_2 = u$ as a state-transition equation of the form

$$\dot{x} = f(x, u), \tag{3}$$

which works the same way as (1) but applies to the new state space X as opposed to $\mathcal{C}$.

To see the structure more clearly, consider the example shown in Figure 3. Here, $\mathcal{C} = \mathbb{R}^2$ to account for the positions of the nonrotatable spacecraft. Three thrusters may be turned on or off, each applying forces f_l, f_r, and f_u. We make three binary action variables u_l, u_f, and u_u; each may take on a value of zero or one to turn off or on the corresponding thruster. Finally, lunar gravity applies a downward force of mg. The following state-transition equation corresponds to independent double integrators in the horizontal and vertical directions:

$$\begin{aligned} \dot{x}_1 &= x_3 \qquad \dot{x}_3 = \frac{f_s}{m}(u_l f_l - u_r f_r), \\ \dot{x}_2 &= x_4 \qquad \dot{x}_4 = \frac{u_u f_u}{m} - g, \end{aligned} \tag{4}$$

which is in the desired form, $\dot{x} = f(x, u)$. Here, we have that $x_1 = q_1$ and $x_2 = q_2$ to account for the position in $\mathcal{C}$. The components x_3 and x_4 are the time derivatives of x_1 and x_2, respectively.

For much more complicated robot systems, the basic structure remains the same. For an n-dimensional C-space, $\mathcal{C}$, the state space X becomes $2n$-dimensional. For a state $x \in X$, the first n components are precisely the configuration parameters and the next n components are their corresponding time derivatives. We can hence imagine that $x = (q, \dot{q})$. Other state-space formulations are possible, including the

ones that can force even higher-order differential equations into first-order form, but these are avoided in this tutorial.

Aside from doubling the dimension, there are conceptually no difficulties with planning in X under differential constraints in comparison to $\mathcal{C}$. Note that obstacles in $\mathcal{C}$ become lifted into X to obtain X_{free}, as shown in Figure 4. If the first n components of x correspond to q and if $q \in \mathcal{C}_{\text{obs}}$ (the obstacle region in C-space), then $x \in X_{\text{obs}}$ regardless of which values are chosen for the remaining components (which correspond to $\dot{q}$).

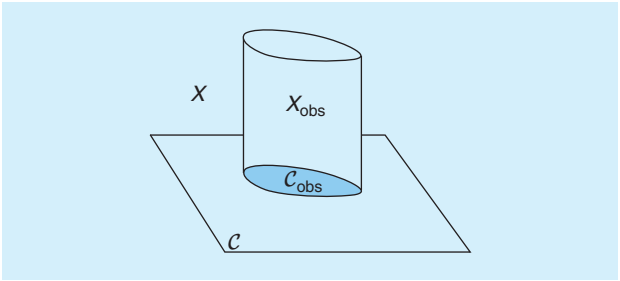

Figure 4. An obstacle region $\mathcal{C}_{\text{obs}} \subset \mathcal{C}$ generates a cylindrical obstacle region $X_{\text{obs}} \subset X$ with respect to the state variables.

Sampling-Based Planning

Now consider the problem of planning under differential constraints. Let X be a state space with a given state-transition equation $\dot{x} = f(x, u)$ and action space U. This model includes the case $X = \mathcal{C}$. Given an initial state $x_I \in X$ and goal region $X_G \subset X$, the task is to compute a function $\tilde{u} : [0, t] \rightarrow U$ that has corresponding trajectory $\tilde{q} : [0, t] \rightarrow X_{\text{free}}$ with $\tilde{q}(0) = x_I$ and $\tilde{q}(t) \in X_G$.

This unifies several problems considered for decades in robotics: 1) nonholonomic planning, which mostly arises from underactuated systems, meaning that the number of action variables is less than the dimension of $\mathcal{C}$; 2) kinodynamic planning, which implies that the original differential constraints on $\mathcal{C}$ are second order, as in the case of Figure 3; these problems include drift, which means that the state may keep changing regardless of the action (for example, you cannot stop a speeding car instantaneously; it must drift); 3) trajectory planning, which has mostly been developed around robot manipulators with dynamics and typically assumes that a collision-free path is given and needs to be transformed into one that satisfies the state-transition equation.

Because of the great difficulty of planning under differential constraints, nearly all planning algorithms are sampling based, as opposed to combinatorial. To develop sampling-based planning algorithms in this context, several discretizations are needed. For ordinary motion planning, only $\mathcal{C}$ needed to be discretized; with differential constraints, the time interval and possibly U require discretization in addition to $\mathcal{C}$ (or X).

One of the simplest ways to discretize the differential constraints is to construct a discrete-time model, which is characterized by three aspects:

1) Time is partitioned into intervals of length Δt. This enables stages to be assigned in which stage k indicates that $(k-1)\Delta t$ time has elapsed.
2) A finite subset U_d of the action space U is chosen. If U is already finite, then this selection may be $U_d = U$.
3) The action $\tilde{u}(t)$ must remain constant over each time interval.

From an initial state, x, a reachability tree can be formed by applying all sequences of discretized actions. Figure 5 shows part of this tree for the Dubins car from the "Modeling the Constraints" section with $U_d = \{-\phi_{\max}, 0, \phi_{\max}\}$. The edges of the tree are circular arcs or line segments. For general

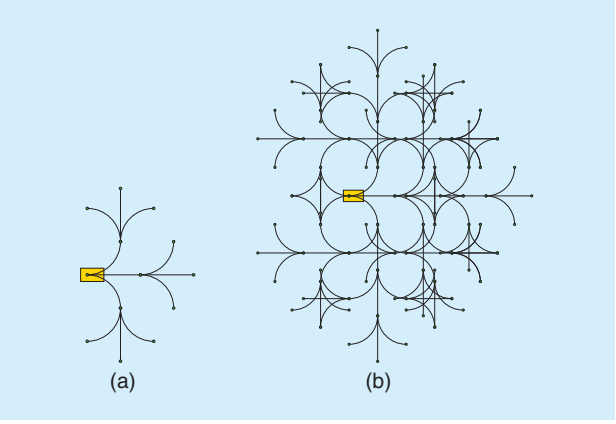

Figure 5. A reachability tree for the Dubins car with three actions. The kth stage produces 3^k new vertices. (a) Two and (b) four stages.

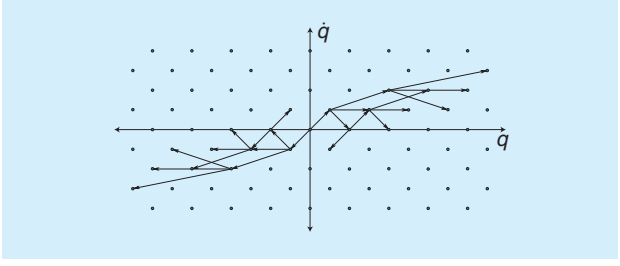

Figure 6. The reachability graph from the origin is shown after three stages (the true state trajectories are actually parabolic arcs when acceleration or deceleration occurs). Note that a lattice is obtained, but the distance traveled in one stage increases as $|\dot{q}|$ increases.

systems, each trajectory segment in the tree is determined by numerical integration of $\dot{x} = f(\tilde{x}(t), \tilde{u}(t))$ for a given $\tilde{u}$. In general, this can be viewed as an incremental simulator that takes an input function $\tilde{u}$ and produces a trajectory segment $\tilde{x}$ that satisfies $\dot{x} = f(\tilde{x}(t), \tilde{u}(t))$ for all times.

Sampling-based planning algorithms proceed by exploring one or more reachability trees that are derived from discretization. In some cases, it is possible to trap the trees onto a regular lattice structure. In this case, planning becomes similar to grid search. Figure 6 shows an example of such a lattice for the double-integrator $\ddot{q} = u$ [6]. For a constant action $u \neq 0$, the trajectory is parabolic and easily obtained by integration. If $u = 0$, then the trajectory is

linear. Consider applying constant actions $u = -a_{\max}$, $u = 0$, and $u = a_{\max}$ for some constant $a_{\max} > 0$ over some fixed interval Δt. The reachability tree becomes a directed acyclic graph, rooted at the origin. Every vertex, except the origin, has an out-degree three, which corresponds to three possible actions. For planning purposes, a solution trajectory can be found by applying standard graph search algorithms to the lattice. If a solution is not found, then the resolution may need to be increased.

Generalizations of this method exist for fully actuated systems. It is also possible to form an approximate lattice, even for underactuated systems, by partitioning the C-space into small cells and ensuring that not more than one reachability tree vertex is expanded per cell [1]; see Figure 7. Each cell is initially marked as being in collision or being collision free but was not yet visited. As cells are visited during the search, they become marked as such. If a potential new vertex lands in a visited cell, it is not saved. This has the effect of pruning the reachability tree.

The planning problem under differential constraints can be solved by incremental sampling and searching, just as the original planning problem in Part I. The discretizations do not necessarily have to be increased as a multiresolution grid. The search trees are constructed by iteratively selecting vertices and applying the incremental simulator to generate trajectory segments. If these are collision free, then they are added to the search trees, and a test for a solution trajectory occurs. One issue commonly confronted is the two-point boundary-value problem (BVP) illustrated in Figure 8. Under differential constraints, it is assumed to be nontrivial to exactly connect a pair of states. Difficult computations may be necessary (a miniplanning problem in itself) to make the connection. Therefore, it is important to minimize the amount of BVP computations if possible.

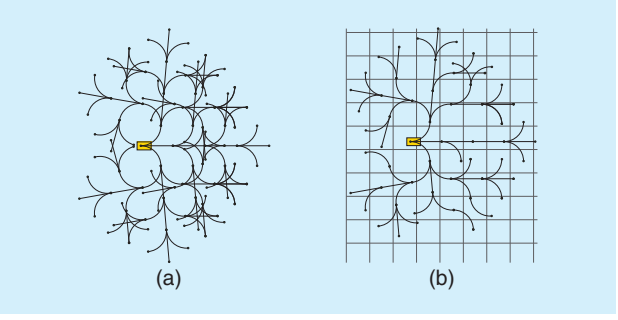

Figure 7. (a) The first four stages of a dense reachability graph for the Dubins car. (b) One possible search graph is obtained by allowing at most one vertex per cell. Many branches are pruned away. In this simple example, there are no cell divisions along the θ axis.

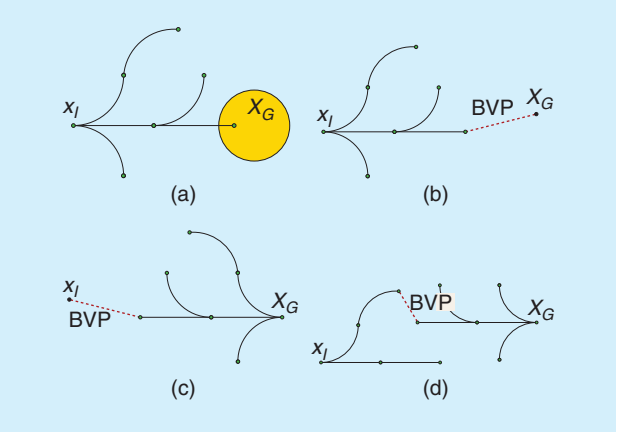

Figure 8. (a) Forward, unidirectional search for which the BVP is avoided. (b) Reaching the goal precisely causes a BVP. (c) Backward, unidirectional search also causes a BVP. (d) For a bidirectional search, the BVP arises when connecting the trees.

Challenges

Although significant progress has been made and many issues are well understood, numerous unresolved issues remain, before planning under differential constraints becomes as well solved as the original planning problem:

- It has been shown in several works (e.g., [9], [10], and [15]) that a wise choice of motion primitives dramatically improves planning performance. Each is an action history $\tilde{u} : [0, \Delta t] \rightarrow U$, and when composed, the state space is efficiently explored. There is no general understanding of how primitives should be designed to optimize planning performance.
- Many sampling-based methods critically depend on the metric over X. Ideally, this metric should be close to the optimal cost-to-go between points; however, calculating these values is as hard as the planning problem itself. What approximations are efficient to compute and useful to planning?
- The region of inevitable collision X_{ric} is the set of all states from which, no matter what action history is applied, entry into X_{obs} is unavoidable. Note that $X_{\text{obs}} \subseteq X_{\text{ric}} \subseteq X$. As a robot moves faster, the portion of the C-space that is essentially forbidden grows due to drift. There has been an interest in calculating estimates of X_{ric} and evidence shows that avoiding it early on in searches improves performance (e.g., [8]); however, more powerful and efficient methods of calculating and incorporating X_{ric} are needed.
- Is it advantageous to trap the system onto a lattice and then perform search, or is it most effective to incrementally explore the reachability tree via special search methods?

Feedback Motion Planning

Recall from Figure 1 that, at the last step, feedback is usually employed to track the path. This becomes necessary because of the imperfections in the transition equation. If the goal is to reach some part of the C-space, then why worry about the artificial problem of tracking a path produced by an imperfect model? This observation calls for a different notion of solution to the planning problem. Rather than computing a path $\tau : [0, 1] \rightarrow \mathcal{C}$ or trajectory $\tilde{q} : [0, t] \rightarrow \mathcal{C}$, we need representations that indicate what

action to apply when the robot is at various places in the C-space. If dynamics are a concern, then we should even know what action to apply from places in the state space X. In these cases, we must feed the current-estimated configuration or state back into the plan to determine which action to apply.

Feasible Feedback Planning

Keep in mind that the issue of differential constraints (the "Differential Constraints" section) is independent of the need for feedback. Both are treated together in control theory and neither is treated in classical path planning; however, the "Differential Constraints" section treated differential constraints without feedback. It is just as sensible to consider feedback without differential constraints as a possible representation on which to build systems.

In the case of having differential constraints, we used the state-transition equation $\dot{x} = f(x, u)$ over the state space X (which includes the case $X = \mathcal{C}$). In the case of no differential constraints, we should directly specify the velocity. In this case, f specializes to $\dot{x} = u$ with $U = \mathbb{R}^n$ (assuming X is n-dimensional). In practice, the speed may be bounded, such as requiring $|u| \leq 1$. This is a very weak differential constraint because it does not constrain the possible directions of motion.

To understand feedback plan representation issues, it is helpful to consider the discrete grid example in Figure 9. A robot moves on a grid, and the possible actions are up ($\uparrow$), down ($\downarrow$), left ($\leftarrow$), right ($\rightarrow$), and terminate (u_T); some directions are not available from some states. In each time step, the robot moves one tile. This corresponds to a discrete-time state-transition equation $x' = f(x, u)$. A solution feedback plan of the form $\pi : X \rightarrow U$ is depicted in Figure 9. From any state, simply follow the arrows to travel to the goal x_G. Each next state is obtained from π and f as $x' = f(x, \pi(x))$. The shown plan is even optimal in the sense that the number of steps to get to x_G is optimal from any starting state.

Another way to represent a feedback plan is through an intermediate potential function $\phi : X \rightarrow [0, \infty]$. Given f and ϕ, a plan π is derived by selecting u according to:

$$u^* = \operatorname*{argmin}_{u \in U(x)} \{\phi(f(x, u))\}, \tag{5}$$

which means that $u^* \in U(x)$ is chosen to reduce ϕ as much as possible (u^* may not be unique).

When is a potential function useful? Let $x' = f(x, u^*)$, which is the state reached after applying the action $u^* \in U(x)$ that was selected by (5). A potential function, ϕ, is called a *navigation function* if

1) $\phi(x) = 0$ for all $x \in X_G$
2) $\phi(x) = \infty$ if and only if no point in X_G is reachable from x
3) for every reachable state, $x \in X \backslash X_G$, applying u^* produces a state x' for which $\phi(x') < \phi(x)$.

In this case, the produced plan is guaranteed to lead to the goal. Figure 10 shows a navigation function for our example. Now consider moving to a continuous C-space. The ideas presented so far nicely extend. The plan $\pi : X \rightarrow U$ applies over whichever space arises, for example, suppose $X = \mathcal{C} \subset \mathbb{R}^2$ and there are polygonal obstacles. Furthermore, there is only a weak differential constraint that $\dot{x} = u$ and $|u| = 1$. A feedback plan must then specify at every point in $\mathcal{C}_{\text{free}}$ a direction to move at unit speed. Figure 11 shows a simple example that converts a triangular decomposition (recall such decompositions from the Part I) into a feedback plan by indicating a

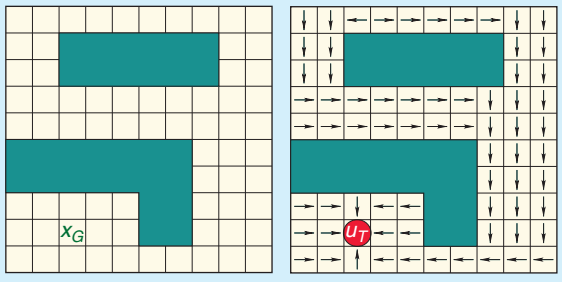

Figure 9. (a) A 2-D grid-planning problem. (b) A solution feedback plan.

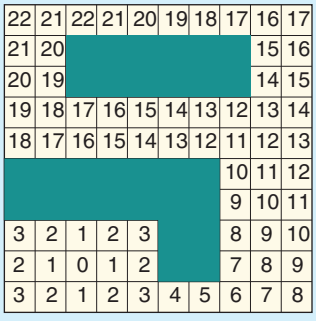

Figure 10. The cost-to-go values serve as a navigation function.

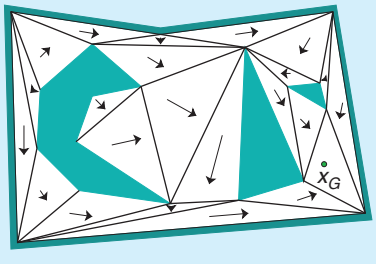

Figure 11. Triangulation is used to define a vector field over X. All solution trajectories lead to the goal.

constant direction inside of each triangle. The task in each triangle is to induce a flow that carries the robot into a triangle that is a step closer to the goal. A navigation function can likewise be constructed on continuous spaces. Figure 12 shows the level sets of a navigation function that sends the robot on the shortest path to the goal.

These examples produce piecewise linear trajectories that are usually inappropriate for execution because the velocity is discontinuous. One weak form of differential constraint is that the resulting plan is smooth along all trajectories to the goal. The method shown in Figure 11 can be adapted to produce smooth vector fields by using bump functions to smoothly blend neighboring field patches [Lin13]. Smooth versions of navigation functions can also be designed for most environments if the obstacles in X are given [16].

Optimal Feedback Planning

In many contexts, we may demand an optimal feedback plan. In the discrete-time case, the goal is to design a plan that optimizes a cost functional,

$$L(x_1, \ldots, x_{K+1}, u_1, \ldots, u_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_{K+1}(x_{K+1}), \tag{6}$$

from every possible start state x_1. Each $l(x_k, u_k) > 0$ is the cost-per-stage and $l_{K+1}(x_{K+1})$ is the final cost, which is zero if $x_{K+1} \in X_G$, or ∞ otherwise. In the special case of $l(x_k, u_k) = 1$ for all x_k and u_k, (6) simply counts the number of steps to reach the goal.

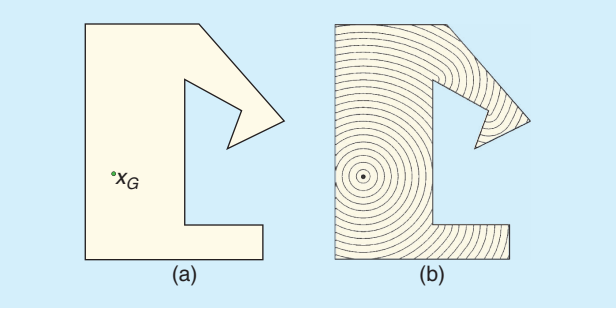

Figure 12. (a) A point goal in a simple polygon. (b) The level sets of the optimal navigation function (Euclidean cost-to-go function).

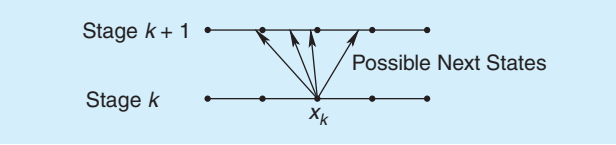

Figure 13. Even though x_k is a sample point, the next state, x_{k+1}, may land between sample points. For each $u_k \in U$, interpolation may be needed for the resulting next state, $x_{k+1} = f(x_k, u_k)$.

The continuous-time counterpart to (6) is

$$L(\tilde{x}, \tilde{u}) = \int_0^{t_F} l(\tilde{x}(t), \tilde{u}(t))dt + l_F(\tilde{x}(t_F)), \tag{7}$$

in which t_F is the termination (or final) time.

Consider a function $G^* : X \to [0, \infty]$ called the optimal cost-to-go, which gives the lowest possible cost $G^*(x)$ to get from any x to X_G. If $x \in X_G$, then $G^*(x) = 0$, and if X_G is not reachable from x, then $G^*(x) = \infty$. Note that G^* is a special form of a navigation function ϕ as defined in the "Feasible Feedback Planning" section. In this case, the optimal plan is executed by applying

$$u^* = \operatorname*{argmin}_{u \in U(x)} \{l(x, u) + G^*(f(x, u))\}. \tag{8}$$

If the term $l(x, u)$ does not depend on the particular u chosen, then (8) actually reduces to (5) with $G^* = \phi$.

The key challenge is to construct the cost-to-go G^*. Fortunately, because of the dynamic programming principle, the cost can be written as (see [12]):

$$G_k^*(x_k) = \min_{u_k \in U(x_k)} \{l(x_k, u_k) + G_{k+1}^*(x_{k+1})\}. \tag{9}$$

The equation expresses the cost-to-go from stage k, G_k^*, in terms of the cost-to-go from stage $k+1$, G_{k+1}^*. The classical method of value iteration [2] can be used to iteratively compute cost-to-go functions until the values stabilize as a stationary G^*. There are also Dijkstra-like [12] and policy iteration methods [2].

When moving to a continuous state space X, the main difficulty is that $G_k^*(x_k)$ cannot be stored for every $x_k \in X$. There are two general approaches. One is to approximate G_k^* using a parametric family of surfaces, such as polynomials or nonlinear basis functions derived from neural networks [3]. The other is to store G_k^* over a finite set of sample points and use interpolation to obtain its value at all other points [11] (see Figure 13). As an example for the one-dimensional case, the value of G_{k+1}^* in (9) at any $x \in [0, 1]$ can be obtained via linear interpolation as

$$G_{k+1}^*(x) \approx \alpha G_{k+1}^*(s_i) + (1 - \alpha) G_{k+1}^*(s_{i+1}), \tag{10}$$

in which the coefficient $\alpha \in [0, 1]$.

Computing such approximate, optimal feedback plans seems to require high-resolution sampling of the state space, which limits their application to lower dimensions (less than five in most applications). Although the planning algorithms are limited to lower-dimensional problems, extensions to otherwise difficult cases are straight forward. For example, consider the stochastic optimal planning problem in which the state-transition equation is expressed as $P(x_{k+1}|x_k, u_k)$. In this case, the expected cost-to-go satisfies

$$G_k^*(x_k) = \min_{u_k \in U(x_k)} \Big\{ l(x_k, u_k) + \sum_{x_{k+1} \in X} G_{k+1}^*(x_{k+1}) P(x_{k+1} | x_k, u_k) \Big\}, \quad (11)$$

which again provides value-iteration methods and in some cases Dijkstra-like algorithms. There are also variations for optimizing worst-case performance, computing game-theoretic equilibria, and reinforcement learning, in which the transition equation must be learned in the process of determining the optimal plan.

Challenges

Feedback motion planning appears to be significantly more challenging than path planning. Some current challenges are

- The curse of dimensionality seems worse. Methods are limited to a few dimensions in practice. Cell decomposition methods do not scale well with dimension and optimal planning methods require high-resolution sampling. Can implicit volumetric representations be constructed and utilized efficiently via sampling?
- Merging with the "Differential Constraints" section leads to both complicated differential constraints and feedback. Hybrid systems models sometimes help by switching controllers over cells during a decomposition [5]. Another possibility is to track space-filling trees, grown backwards from the goal, as opposed to single paths [17]. If optimality is not required, there are great opportunities to improve planning efficiency.
- If a fast-enough path-planning algorithm exists for a problem, then the feedback plan could be a dynamic replanner that recomputes the path as the robot ends up in unexpected states or obstacle change. When is this kind of solution advantageous and how does it relate to explicitly computing $\pi : X \rightarrow U$?
- Perhaps the plan as a mapping $\pi : X \rightarrow U$ is too constraining. Would it be preferable to compute a plan that indicates for every state a set of possible actions that are all guaranteed to make progress toward the goal? This would leave more flexibility during execution to account for unexpected events.

Sensing Uncertainty

Recall from the "Limitations of Path Planning" section that, after following the classical steps in Figure 1, the information requirements are driven artificially high: complete state information, including the models of the obstacles, is needed at all times. On the other hand, we see numerous examples in robotics and nature of simple systems that cannot possibly build complete maps of their environment while nevertheless accomplishing interesting tasks. A simple Roomba vacuum cleaner can obtain a reasonable level of coverage with poor sensors and no prior obstacle knowledge. Ants are able to construct complex living spaces and transport food and materials. Since maintaining the entire state seems futile for most problems, it makes sense to start with the desired task and determine what information is required to solve it. This could lead to a minimalist approach in which a cheap combination of simple sensors, actuators, and computation is sufficient.

The goal in this section is to give you a basic idea of how planning appears from this perspective. There are many open challenges and directions for future research. The presentation here gives representative examples rather than complete modeling alternatives; for more details, see [12] and [13].

Let X be a state space that is typically much larger than $\mathcal{C}$. A state $x \in X$ may contain robot configuration parameters, configuration velocities, and even a complete representation of the obstacles $\mathcal{O} \subset \mathcal{W}$. A change in x could correspond to a moving robot or a change in obstacles. In this case, X is not even assumed to be a manifold (it is just a large set). Suppose $x' = f(x, u)$ for $u \in U$ is a discrete-time state-transition equation that indicates how the entire world changes.

In this section, the state x is hidden from the robot. The only information it receives from the external world is from sensor mappings of the form $h : X \rightarrow Y$, in which Y is a set of sensor outputs, called the *observation space*. Consider h as a many-to-one mapping. A weaker sensor causes more states to produce the same output. In other words, the preimage $h^{-1}(y) = \{x \in X \,|\, y = h(x)\}$ is larger.

At any time during execution, the complete set of information available to the robot consists of all sensor observations and all actions that were applied (and any given initial conditions). This is called the *history information state* (or *I-state*); if an observation and action occur at each stage, then it appears at stage k as

$$\eta_k = (u_1, \ldots, u_{k-1}, y_1, \ldots, y_k). \quad (12)$$

Imagine placing a set of all possible η_k for all $k \geq 1$ in a large set $\mathcal{I}_{\text{hist}}$ called the *history I-space*. Although $\mathcal{I}_{\text{hist}}$ is enormous, the state $\eta_k \in \mathcal{I}_{\text{hist}}$ is at least not hidden from the robot. We can therefore define an information feedback plan $\pi : \mathcal{I}_{\text{hist}} \rightarrow U$.

Of course, $\mathcal{I}_{\text{hist}}$ is so large that it is impractical to work directly with it. Therefore, we design a filter that compresses each η_k to retain only some task-critical pieces of information. The result is an implied information mapping $\kappa : \mathcal{I}_{\text{hist}} \rightarrow \mathcal{I}$ into some new filter I-space $\mathcal{I}$. As new information, u_{k-1} and y_k, becomes available, the filter I-state $\imath_k \in \mathcal{I}$ becomes updated through a filter transition equation

$$\imath_k = \phi(\imath_{k-1}, u_{k-1}, y_k). \quad (13)$$

Now let $\mathcal{I}$ be any I-space. Generally, the planning problem is to choose each u_k so that some predetermined goal is achieved. Let $G \subset \mathcal{I}$ be called a *goal region* in the I-space. Starting from an initial I-state $\imath_0 \in \mathcal{I}$, what sequence of

actions $u_1, u_2, \ldots$, will lead to some future I-state $\imath_k \in G$? Since future observations are usually unpredictable, it may be impossible to specify the appropriate action sequence in advance. Therefore, a better way to define the action selections is to define a plan $\pi : I \rightarrow U$, which specifies an action $\pi(\imath)$ from every filter I-state $\imath \in \mathcal{I}$.

During execution of the plan, the filter (13) is executed, filter I-states $\imath \in \mathcal{I}$ are generated, and actions get automatically applied using $u = \pi(\imath)$. The state-transition equation $x' = f(x, u)$ produces the next state, which remains hidden.

Using a filter ϕ, the execution of a plan can be expressed as

$$\imath_k = \phi(\imath_{k-1}, y_k, \pi(\imath_{k-1})), \tag{14}$$

which makes the filter no longer appear to depend on actions. The filter runs autonomously as the observations appear.

Generic Examples

To help understand the concepts so far, we describe some well-known approaches in terms of filters over I-spaces $\mathcal{I}$ and information feedback plans $\pi : \mathcal{I} \rightarrow U$.

Most tasks require some memory of the sensing and action histories.

State Feedback

Suppose we have a filter that produces a reliable estimate of x_k using η_k and fits the incremental form (13), in which the I-space is $\mathcal{I} = X$ and $\imath_k$ is the estimate of x_k. In this case, a plan as expressed in (14) becomes $\pi : X \rightarrow U$. This method was implicitly used throughout the "Feedback Motion Planning" section.

This choice of the filter is convenient because there is no need to worry in the planning and execution stages about its uncertainty with regard to the current state. All sensing uncertainty is the problem of the filter. This is a standard approach throughout control theory and robotics; however, as mentioned in the "Limitations of Path Planning" section, the information requirement may be artificially high.

Open Loop

This example uses (13) to count the number of stages by incrementing a counter in each step. The I-space is $\mathcal{I} = \mathbb{N}$.

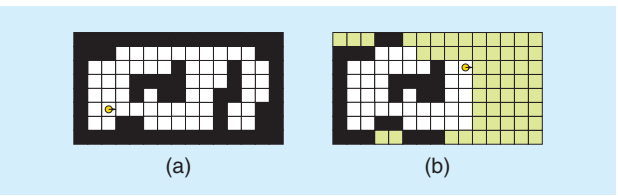

Figure 14. (a) A discrete grid problem is made in which a robot is placed into a bounded, unknown environment. (b) An encoding of a partial map obtained after some exploration. The hatched lines represent unknown tiles (neither white nor black).

A plan is expressed as $\pi : \mathbb{N} \rightarrow U$. This can be interpreted as specifying a sequence of actions:

$$\pi = (u_1, u_2, u_3, \ldots). \tag{15}$$

The result is just a sequence of actions to apply. Such plans are often called *open loop* because no significant sensor observations are being utilized during execution. However, it is important to be careful, because implicit time information is being used. It is known that u_3 is being applied later than u_2 for example.

Sensor Feedback

At one extreme, we can make the system memoryless or reactive, causing actions to depend only on the current observation y_k. In this case, $\mathcal{I} = Y$ and (13) returns y_k in each iteration. A plan becomes $\pi : Y \rightarrow X$. If a useful task can be solved in this way, then it is almost always advantageous to do so. Most tasks, however, require some memory of the sensing and action histories.

Full History Feedback

Sensor feedback was at one end of the spectrum by discarding all history. At the other end, we can retain all history. The filter (13) simply concatenates u_{k-1} and y_k onto the history. The filter I-space is just $\mathcal{I} = \mathcal{I}_{\text{hist}}$. As mentioned before, however, this becomes unmanageable at the planning stage.

Designing Task-Specific I-Spaces

It is best to design the I-space around the task. A discrete exploration task is presented first. A robot is placed into a discrete environment in which coordinates are described by a pair (i, j) of integers, and there are only four possible orientations (such as north, east, west, south). The state space is

$$X = \mathbb{Z} \times \mathbb{Z} \times D \times \mathcal{E}, \tag{16}$$

in which $\mathbb{Z} \times \mathbb{Z}$ is the set of all (i, j) positions, D is the set of four possible directions, and $\mathcal{E}$ is a set of environments. Every $E \in \mathcal{E}$ is a connected, bounded set of "white" tiles, and all such possibilities are included in $\mathcal{E}$; an example appears in Figure 14(a). All other tiles are "black." Note that $\mathbb{Z} \times \mathbb{Z} \times D$ can be imagined as a discrete version of $\mathbb{R}^2 \times \mathcal{S}^1$.

The robot is initially placed on a white tile in an unknown environment and unknown orientation. The task is to move the robot so that every tile in E is visited. This strategy could be used to find a lost treasure that has been placed on an unknown tile. Only two actions are needed: 1) move forward in the direction the robot is facing and 2) rotate the robot 90° counterclockwise. If the robot is facing a black tile and forward is applied, then a sensor reports that it is blocked and the robot does not move.

Consider what kind of filters can be used for solving this task. The most straightforward one is for the robot to

construct a partial map of E and maintain its position and orientation with respect to its map. A naive way to attempt this is to enumerate all possible $E \in \mathcal{E}$ that are consistent with the history I-state, and for each one, enumerate all possible $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ and orientations in D. Such a filter would live in an I-space $I = \text{pow}(\mathbb{Z} \times \mathbb{Z} \times D \times E)$, with each I-state being a subset of $\mathcal{I}$. An immediate problem is that every I-state describes a complicated, infinite set of possibilities.

A slightly more clever way to handle this is to compress the information into a single map, as shown in Figure 14(b). Rather than be forced to label every $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ as "black" or "white," we can assign a third label, "unknown." Initially, the tile that contains the robot is "white" and all others are "unknown." As the robot is blocked by walls, some tiles become labeled as "black." The result is a partial map that has a finite number of "white" and "black" tiles, with all other tiles being labeled "unknown." An I-state can be described as two finite sets W (white tiles) and B (black tiles), which are disjoint subsets of $\mathbb{Z} \times \mathbb{Z}$. Any tile not included in W or B is assumed to be "unknown."

Now consider a successful search plan that uses this filter. For any "unknown" tile that is adjacent to a "white" tile, we attempt to move the robot onto it to determine how to label it. This process repeats until no more "unknown" tiles are reachable, which implies that the environment has been completely explored.

A far more interesting filter and plan are given in [4]. Their filter maintains I-states that use only logarithmic memory in terms of the number of tiles, whereas recording the entire map would use linear memory. They show that with very little space, not nearly enough to build a map, the environment can nevertheless be systematically searched. For this case, the I-state keeps track of only one coordinate (for example, in the north–south direction) and the orientation, expressed with 2 b. A plan is defined in [4] that is guaranteed to visit all white tiles using only this information.

Moving to continuous spaces leads to the familiar simultaneous robot localization and mapping (SLAM) problem [7], [18]. For the localization problem alone, a Kalman filter is used. In this case, the filter I-state is $\iota = (\mu, \Sigma)$ in which μ is the robot configuration estimate and Σ is the covariance. The Kalman filter computes transitions that follow the form (13). When mapping is combined, each filter I-state encodes a probability distribution over possible maps and configurations. The I-space $\mathcal{I}$ becomes so large that sampling-based particle filters are developed to approximately compute (13).

A full geometric map is useful for many tasks; however, the I-space can be dramatically reduced by focusing on a particular task. An example from [19] is briefly described here. Consider a simple gap sensor placed on a mobile robot in a polygonal environment, as shown in Figure 15. Suppose the task is to optimally navigate the robot in terms of the shortest possible Euclidean distance. The robot is not given a map of the environment. Instead, it uses gap observations and records an association between gaps when two gaps merge into one. It is shown in [19] that this precisely corresponds to the discovery of a bitangent edge, which is a key part of the shortest path graph (alternatively called *reduced visibility graph*), a data structure that encodes the common edges of optimal paths from all initial-goal pairs of positions. The filter I-state records a tree, shown in Figure 16, that indicates how the gaps merged. The tree itself is combinatorial (no geometric data) and precisely encodes the structure needed for optimal robot navigation from the robot's current location. The robot is equipped with an action that allows it to chase any gap until that gap disappears or splits into other gaps. Using the tree, it can optimally navigate to any place that it has previously seen. The set of all trees forms the filter I-space $\mathcal{I}$ from which distance-optimal navigation can be entirely solved in an unknown environment without measuring distances.

Challenges

Because of the wide variety of tasks and possible combinations of sensors and control models, many challenges remain to design planning algorithms by reducing the

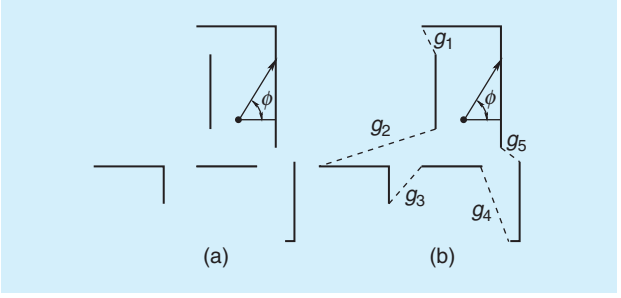

Figure 15. Consider a robot placed in a simple polygon. (a) A strong sensor could omnidirectionally seem to provide a distance measurement along every direction from 0 to 2π. (b) A *gap sensor* can only indicate that there are discontinuities in depth. A cyclic list of gaps $\{g_1, g_2, g_3, g_4, g_5\}$ is obtained, with no angle or distance measurements.

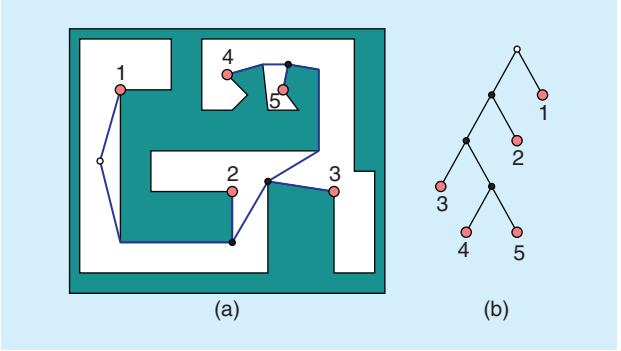

Figure 16. (a) The gap navigation tree captures the structure of the shortest paths to the current robot location (the white circle on the left). (b) The tree precisely characterizes how the shortest paths to the robot location are structured.

complexity of the I-space. The overall framework involves the following steps:

1) formulate the task and the type of system, which includes the environment obstacles, moving bodies, and possible sensors
2) define the models, which provide the state space X, sensor mappings h, and the state-transition function f
3) determine an I-space $\mathcal{I}$ for which a filter ϕ can be practically computed
4) take the desired goal, expressed over X, and convert it into an expression over $\mathcal{I}$
5) compute a plan π over $\mathcal{I}$ that achieves the goal in terms of $\mathcal{I}$.

Ideally, all these steps should be taken into account together; otherwise, a poor choice in an earlier step could lead to an artificially high complexity in later steps. Worse yet, a feasible solution might not even exist. Consider how steps 4 and 5 may fail. Suppose that in Step 3, a simple I-space is designed so that each I-state is straightforward and efficient to compute. If we are not careful, then Step 4 could fail because it might be impossible to determine whether particular I-states achieve the goal. For example, the open-loop filter from the "Generic Examples" section simply keeps track of the current stage number. In most settings, this provides no relevant information about what has been achieved in the state space. Suppose that Step 4 is successful, consider what could happen in Step 5. A nice filter could be designed with an easily expressed goal in $\mathcal{I}$; however, there might not exist plans that could achieve it. In the light of these difficulties, one open challenge may be to design a decomposition, better than the one in Figure 1, of the overall problem so that information requirements are reduced along the way.

Conclusions

Note the sharp contrast between Parts I and II of this tutorial. From the perspective of Part I, it is tempting to think that motion planning is dead as a research field. Most of the issues have been well studied for decades, and powerful methods have been developed that are in widespread use throughout various industries. However, differential constraints, feedback, optimality, sensing uncertainty, and numerous other issues continue to bring exciting new challenges. In some sense, combining the components in Figure 1 leads to merging planning and control theory. Thus, the subject of planning at this level might just as well be considered as algorithmic control theory in which control approaches are enhanced to take advantage of geometric data structures, sampling-based searching methods, collision-detection algorithms, and other tools familiar to motion planning. The wild frontiers are open, and there are plenty of interesting places to explore.

Acknowledgments

The author is grateful for the following support: NSF grant 0904501 (IIS Robotics), NSF grant 1035345 (CNS Cyber-physical Systems), DARPA SToMP grant HR0011-05-1-0008, and MURI/ONR grant N00014-09-1-1052.

References

[1] J. Barraquand and J.-C. Latombe, "Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles," *Algorithmica*, vol. 10, pp. 121–155, 1993.

[2] R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming*, Princeton, NJ: Princeton Univ. Press, 1962.

[3] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Belmont, MA: Athena Scientific1996.

[4] M. Blum and D. Kozen, "On the power of the compass (or, why mazes are easier to search than graphs)," in *Proc. Annu. Symp. Foundations of Computer Science*, 1978, pp. 132–142.

[5] M. S. Branicky, V. S. Borkar, and S. K. Mitter, "A unified framework for hybrid control: Model and optimal control theory," *IEEE Trans. Automat. Contr.*, vol. 43, no. 1, pp. 31–45, 1998.

[6] B. R. Donald, P. G. Xavier, J. Canny, and J. Reif, "Kinodynamic planning," *J. ACM*, vol. 40, pp. 1048–1066, Nov. 1993.

[7] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: Part I," *IEEE Robot. Automat. Mag.*, vol. 13, no. 2, pp. 99–110, 2006.

[8] Th. Fraichard and H. Asama, "Inevitable collision states—A step towards safer robots?" *Adv. Robot.*, pp. 1001–1024, 2004.

[9] E. Frazzoli, M. A. Dahleh, and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Trans. Robot.*, vol. 21, no. 6, pp. 1077–1091, Dec. 2005.

[10] J. Go, T. Vu, and J. J. Kuffner, "Autonomous behaviors for interactive vehicle animations," *Proc. SIGGRAPH Symp. Computer Animation*, 2004.

[11] R. E. Larson and J. L. Casti, *Principles of Dynamic Programming*, part II. New York: Dekker, 1982.

[12] S. M. LaValle. (2006). *Planning Algorithms*, Cambridge, U.K., Cambridge Univ. Press [Online]. Available: http://planning.cs.uiuc.edu/

[13] S. M. LaValle. (2009, Oct.). Filtering and planning in information spaces. Dept. Comput. Sci., Univ. Illinois, Tech. Rep. [Online]. Available: http://msl.cs.uiuc.edu/~lavalle/iros09/paper.pdf

[14] S. R. Lindemann and S. M. LaValle, "Simple and efficient algorithms for computing smooth, collision-free feedback laws over given cell decompositions," *Int. J. Robot. Res.*, vol. 28, no. 5, pp. 600–621, 2009.

[15] M. Pivtoraiko and A. Kelly, "Generating near minimal spanning control sets for constrained motion planning in discrete state spaces," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2005.

[16] E. Rimon and D. E. Koditschek, "Exact robot navigation using artificial potential fields," *IEEE Trans. Robot. Automat.*, vol. 8, no. 5, pp. 501–518, Oct. 1992.

[17] R. Tedrake, I. R. Manchester, M. M. Tobenkin, and J. W. Roberts, "Time optimal trajectories for bounded velocity differential drive vehicles," *Int. J. Robot. Res.*, vol. 29, pp. 1038–1052, July 2010.

[18] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, Cambridge, MA, MIT Press, 2005.

[19] B. Tovar, R Murrieta, and S. M. LaValle, "Distance-optimal navigation in an unknown environment without sensing distances," *IEEE Trans. Robot.*, vol. 23, no. 3, pp. 506–518, June 2007.

Steven M. LaValle, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. E-mail: lavalle@uiuc.edu.