Chapter 10

Motion Planning

Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles in the environment and satisfies other constraints, such as joint limits or torque limits. Motion planning is one of the most active subfields of robotics, and it is the subject of entire books. The purpose of this chapter is to provide a practical overview of a few common techniques, using robot arms and mobile robots as the primary example systems (Figure 10.1).

The chapter begins with a brief overview of motion planning. This is followed by foundational material including configuration space obstacles and graph search. We conclude with summaries of several different planning methods.

10.1 Overview of Motion Planning

A key concept in motion planning is configuration space, or **C-space** for short. Every point in the C-space $\mathcal{C}$ corresponds to a unique configuration q of the robot, and every configuration of the robot can be represented as a point in C-space. For example, the configuration of a robot arm with n joints can be represented as a list of n joint positions, $q = (\theta_1, \ldots, \theta_n)$. The **free C-space** $\mathcal{C}_{\text{free}}$ consists of the configurations where the robot neither penetrates an obstacle nor violates a joint limit.

In this chapter, unless otherwise stated, we assume that q is an n-vector and that $\mathcal{C} \subset \mathbb{R}^n$. With some generalization, the concepts of this chapter apply to non-Euclidean C-spaces such as $\mathcal{C} = SE(3)$.

The control inputs available to drive the robot are written as an m-vector $u \in \mathcal{U} \subset \mathbb{R}^m$, where $m = n$ for a typical robot arm. If the robot has second-

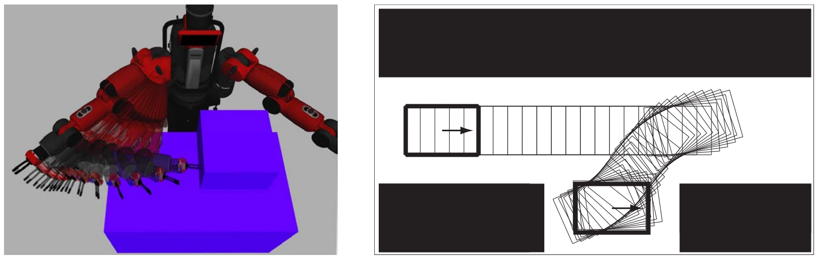


Figure 10.1: (Left) A robot arm executing an obstacle-avoiding motion plan. The motion plan was generated using MoveIt! [180] and visualized using rviz in ROS (the Robot Operating System). (Right) A car-like mobile robot executing parallel parking.

order dynamics, such as that for a robot arm, and the control inputs are forces (equivalently, accelerations), the *state* of the robot is defined by its configuration and velocity, $x = (q, v) \in \mathcal{X}$. For $q \in \mathbb{R}^n$, typically we write $v = \dot{q}$. If we can treat the control inputs as velocities, the state x is simply the configuration q. The notation $q(x)$ indicates the configuration q corresponding to the state x, and $\mathcal{X}_{\text{free}} = \{x \mid q(x) \in \mathcal{C}_{\text{free}}\}$.

The equations of motion of the robot are written

$$\dot{x} = f(x, u) \tag{10.1}$$

or, in integral form,

$$x(T) = x(0) + \int_0^T f(x(t), u(t))dt. \tag{10.2}$$

10.1.1 Types of Motion Planning Problems

With the definitions above, a fairly broad specification of the motion planning problem is the following:

Given an initial state $x(0) = x_{\text{start}}$ and a desired final state x_{goal}, find a time T and a set of controls $u : [0, T] \to \mathcal{U}$ such that the motion (10.2) satisfies $x(T) = x_{\text{goal}}$ and $q(x(t)) \in \mathcal{C}_{\text{free}}$ for all $t \in [0, T]$.

It is assumed that a feedback controller (Chapter 11) is available to ensure that the planned motion $x(t)$, $t \in [0, T]$, is followed closely. It is also assumed that an accurate geometric model of the robot and environment is available to evaluate $\mathcal{C}_{\text{free}}$ during motion planning.

There are many variations of the basic problem; some are discussed below.

Path planning versus motion planning. The path planning problem is a subproblem of the general motion planning problem. Path planning is the purely geometric problem of finding a collision-free path $q(s), s \in [0,1]$, from a start configuration $q(0) = q_{\text{start}}$ to a goal configuration $q(1) = q_{\text{goal}}$, without concern for the dynamics, the duration of motion, or constraints on the motion or on the control inputs. It is assumed that the path returned by the path planner can be time scaled to create a feasible trajectory (Chapter 9). This problem is sometimes called the **piano mover's problem**, emphasizing the focus on the geometry of cluttered spaces.

Control inputs: $m = n$ versus $m < n$. If there are fewer control inputs m than degrees of freedom n, then the robot is incapable of following many paths, even if they are collision-free. For example, a car has $n = 3$ (the position and orientation of the chassis in the plane) but $m = 2$ (forward–backward motion and steering); it cannot slide directly sideways into a parking space.

Online versus offline. A motion planning problem requiring an immediate result, perhaps because obstacles appear, disappear, or move unpredictably, calls for a fast, online, planner. If the environment is static then a slower offline planner may suffice.

Optimal versus satisficing. In addition to reaching the goal state, we might want the motion plan to minimize (or approximately minimize) a cost J, e.g.,

$$J = \int_0^T L(x(t), u(t))dt.$$

For example, minimizing with $L = 1$ yields a time-optimal motion while minimizing with $L = u^{\mathrm{T}}(t)u(t)$ yields a "minimum-effort" motion.

Exact versus approximate. We may be satisfied with a final state $x(T)$ that is sufficiently close to x_{goal}, e.g., $\|x(T) - x_{\text{goal}}\| < \epsilon$.

With or without obstacles. The motion planning problem can be challenging even in the absence of obstacles, particularly if $m < n$ or optimality is desired.

10.1.2 Properties of Motion Planners

Planners must conform to the properties of the motion planning problem as outlined above. In addition, planners can be distinguished by the following properties.

Multiple-query versus single-query planning. If the robot is being asked to solve a number of motion planning problems in an unchanging environment, it may be worth spending the time building a data structure that accurately represents $\mathcal{C}_{\text{free}}$. This data structure can then be searched to solve multiple planning queries efficiently. Single-query planners solve each new problem from scratch.

"Anytime" planning. An anytime planner is one that continues to look for better solutions after a first solution is found. The planner can be stopped at any time, for example when a specified time limit has passed, and the best solution returned.

Completeness. A motion planner is said to be **complete** if it is guaranteed to find a solution in finite time if one exists, and to report failure if there is no feasible motion plan. A weaker concept is **resolution completeness**. A planner is resolution complete if it is guaranteed to find a solution if one exists at the resolution of a discretized representation of the problem, such as the resolution of a grid representation of $\mathcal{C}_{\text{free}}$. Finally, a planner is **probabilistically complete** if the probability of finding a solution, if one exists, tends to 1 as the planning time goes to infinity.

Computational complexity. The computational complexity refers to characterizations of the amount of time the planner takes to run or the amount of memory it requires. These are measured in terms of the description of the planning problem, such as the dimension of the C-space or the number of vertices in the representation of the robot and obstacles. For example, the time for a planner to run may be exponential in n, the dimension of the C-space. The computational complexity may be expressed in terms of the average case or the worst case. Some planning algorithms lend themselves easily to computational complexity analysis, while others do not.

10.1.3 Motion Planning Methods

There is no single planner applicable to all motion planning problems. Below is a broad overview of some of the many motion planners available. Details are left to the sections indicated.

Complete methods (Section 10.3). These methods focus on exact representations of the geometry or topology of $\mathcal{C}_{\text{free}}$, ensuring completeness. For all but simple or low-degree-of-freedom problems, these representations are mathematically or computationally prohibitive to derive.

Grid methods (Section 10.4). These methods discretize $\mathcal{C}_{\text{free}}$ into a grid and search the grid for a motion from q_{start} to a grid point in the goal region. Modifications of the approach may discretize the state space or control space or they may use multi-scale grids to refine the representation of $\mathcal{C}_{\text{free}}$ near obstacles. These methods are relatively easy to implement and can return optimal solutions but, for a fixed resolution, the memory required to store the grid, and the time to search it, grow exponentially with the number of dimensions of the space. This limits the approach to low-dimensional problems.

Sampling methods (Section 10.5). A generic sampling method relies on a random or deterministic function to choose a sample from the C-space or state space; a function to evaluate whether the sample is in $\mathcal{X}_{\text{free}}$; a function to determine the "closest" previous free-space sample; and a local planner to try to connect to, or move toward, the new sample from the previous sample. This process builds up a graph or tree representing feasible motions of the robot. Sampling methods are easy to implement, tend to be probabilistically complete, and can even solve high-degree-of-freedom motion planning problems. The solutions tend to be satisficing, not optimal, and it can be difficult to characterize the computational complexity.

Virtual potential fields (Section 10.6). Virtual potential fields create forces on the robot that pull it toward the goal and push it away from obstacles. The approach is relatively easy to implement, even for high-degree-of-freedom systems, and fast to evaluate, often allowing online implementation. The drawback is local minima in the potential function: the robot may get stuck in configurations where the attractive and repulsive forces cancel but the robot is not at the goal state.

Nonlinear optimization (Section 10.7). The motion planning problem can be converted to a nonlinear optimization problem by representing the path or controls by a finite number of design parameters, such as the coefficients of a polynomial or a Fourier series. The problem is to solve for the design parameters that minimize a cost function while satisfying constraints on the controls, obstacles, and goal. While these methods can produce near-optimal solutions, they require an initial guess at the solution. Because the objective function and feasible solution space are generally not convex, the optimization process can get stuck far away from a feasible solution, let alone an optimal solution.

Smoothing (Section 10.8). Often the motions found by a planner are jerky.

A smoothing algorithm can be run on the result of the motion planner to improve the smoothness.

A major trend in recent years has been toward sampling methods, which are easy to implement and can handle high-dimensional problems.

10.2 Foundations

Before discussing motion planning algorithms, we establish concepts used in many of them: configuration space obstacles, collision detection, graphs, and graph search.

10.2.1 Configuration Space Obstacles

Determining whether a robot at a configuration q is in collision with a known environment generally requires a complex operation involving a CAD model of the environment and robot. There are a number of free and commercial software packages that can perform this operation, and we will not delve into them here. For our purposes, it is enough to know that the workspace obstacles partition the configuration space $\mathcal{C}$ into two sets, the **free space** $\mathcal{C}_{\text{free}}$ and the **obstacle space** $\mathcal{C}_{\text{obs}}$, where $\mathcal{C} = \mathcal{C}_{\text{free}} \cup \mathcal{C}_{\text{obs}}$. Joint limits are treated as obstacles in the configuration space.

With the concepts of $\mathcal{C}_{\text{free}}$ and $\mathcal{C}_{\text{obs}}$, the path planning problem reduces to the problem of finding a path for a point robot among the obstacles $\mathcal{C}_{\text{obs}}$. If the obstacles break $\mathcal{C}_{\text{free}}$ into separate **connected components**, and q_{start} and q_{goal} do not lie in the same connected component, then there is no collision-free path.

The explicit mathematical representation of a C-obstacle can be exceedingly complex, and for that reason C-obstacles are rarely represented exactly. Despite this, the concept of C-obstacles is very important for understanding motion planning algorithms. The ideas are best illustrated by examples.

10.2.1.1 A 2R Planar Arm

Figure 10.2 shows a 2R planar robot arm, with configuration $q = (\theta_1, \theta_2)$, among obstacles A, B, and C in the workspace. The C-space of the robot is represented by a portion of the plane with $0 \leq \theta_1 < 2\pi$, $0 \leq \theta_2 < 2\pi$. Remember from Chapter 2, however, that the topology of the C-space is a torus (or doughnut) since the edge of the square at $\theta_1 = 2\pi$ is connected to the edge $\theta_1 = 0$; similarly, $\theta_2 = 2\pi$ is connected to $\theta_2 = 0$. The square region of $\mathbb{R}^2$ is obtained by slicing

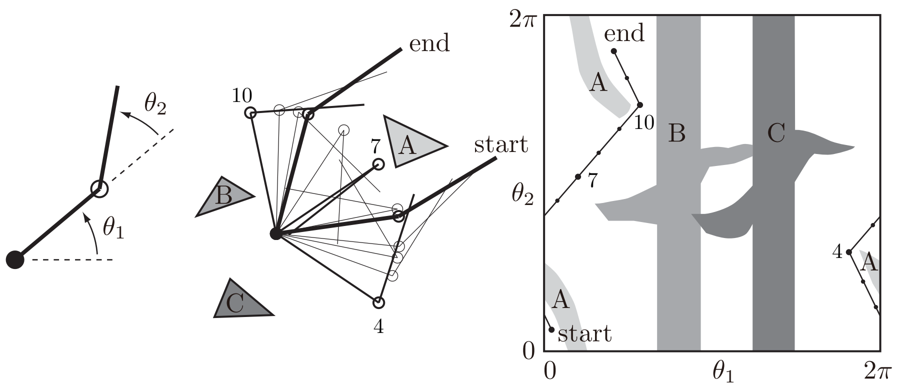


Figure 10.2: (Left) The joint angles of a 2R robot arm. (Middle) The arm navigating among obstacles A, B, and C. (Right) The same motion in C-space. Three intermediate points, 4, 7, and 10, along the path are labeled.

the surface of the doughnut twice, at $\theta_1 = 0$ and $\theta_2 = 0$, and laying it flat on the plane.

The C-space on the right in Figure 10.2 shows the workspace obstacles A, B, and C represented as C-obstacles. Any configuration lying inside a C-obstacle corresponds to penetration of the obstacle by the robot arm in the workspace. A free path for the robot arm from one configuration to another is shown in both the workspace and C-space. The path and obstacles illustrate the topology of the C-space. Note that the obstacles break $\mathcal{C}_{\text{free}}$ into three connected components.

10.2.1.2 A Circular Planar Mobile Robot

Figure 10.3 shows a top view of a circular mobile robot whose configuration is given by the location of its center, $(x, y) \in \mathbb{R}^2$. The robot translates (moves without rotating) in a plane with a single obstacle. The corresponding C-obstacle is obtained by "growing" (enlarging) the workspace obstacle by the radius of the mobile robot. Any point outside this C-obstacle represents a free configuration of the robot. Figure 10.4 shows the workspace and C-space for two obstacles, indicating that in this case the mobile robot cannot pass between the two obstacles.

10.2.1.3 A Polygonal Planar Mobile Robot That Translates

Figure 10.5 shows the C-obstacle for a polygonal mobile robot translating in the presence of a polygonal obstacle. The C-obstacle is obtained by sliding the

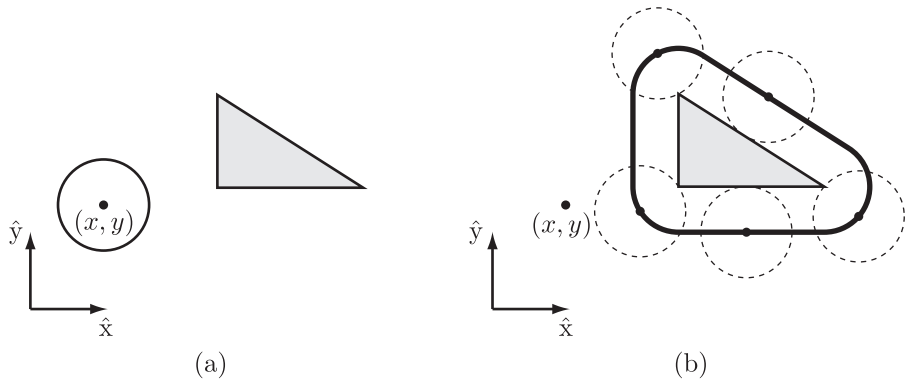


Figure 10.3: (a) A circular mobile robot (open circle) and a workspace obstacle (gray triangle). The configuration of the robot is represented by (x, y), the center of the robot. (b) In the C-space, the obstacle is "grown" by the radius of the robot and the robot is treated as a point. Any (x, y) configuration outside the bold line is collision-free.

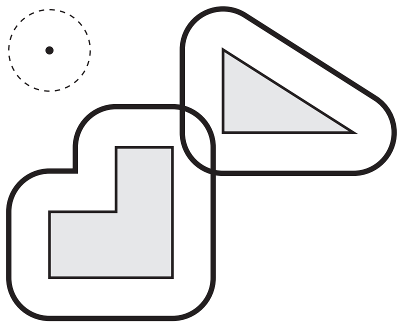


Figure 10.4: The "grown" C-space obstacles corresponding to two workspace obstacles and a circular mobile robot. The overlapping boundaries mean that the robot cannot move between the two obstacles.

robot along the boundary of the obstacle and tracing the position of the robot's reference point.

10.2.1.4 A Polygonal Planar Mobile Robot That Translates and Rotates

Figure 10.6 illustrates the C-obstacle for the workspace obstacle and triangular mobile robot of Figure 10.5 if the robot is now allowed to rotate. The C-space

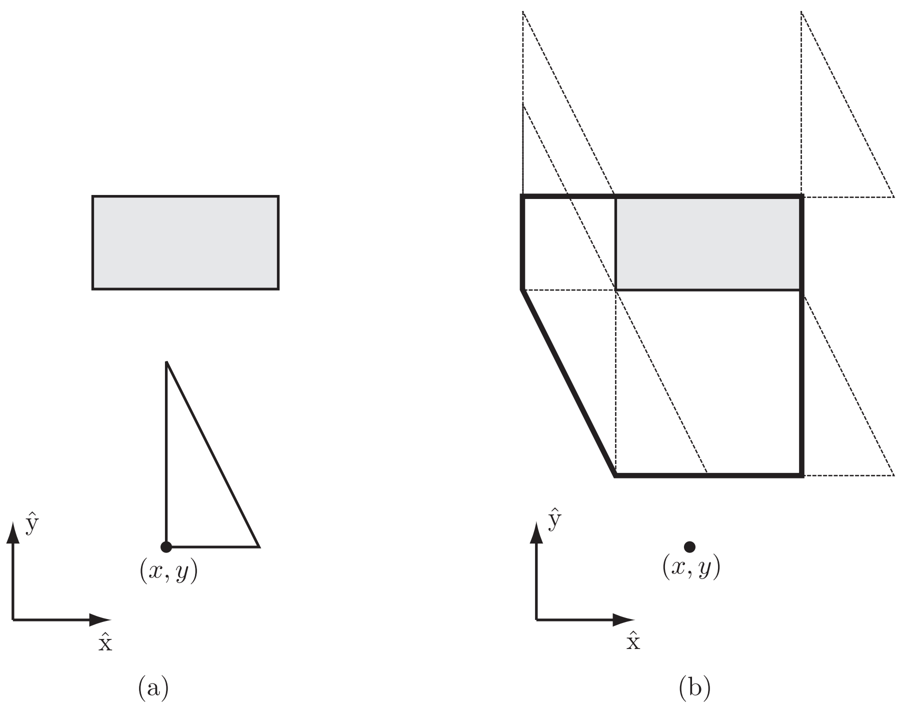
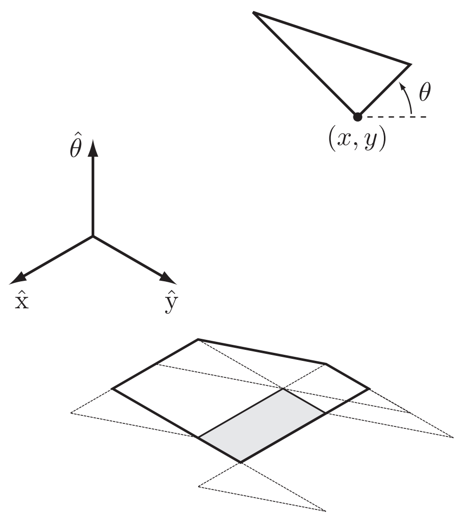


Figure 10.5: (a) The configuration of a triangular mobile robot, which can translate but not rotate, is represented by the (x, y) location of a reference point. Also shown is a workspace obstacle in gray. (b) The corresponding C-space obstacle (bold outline) is obtained by sliding the robot around the boundary of the obstacle and tracing the position of the reference point.

is now three dimensional, given by $(x, y, \theta) \in \mathbb{R}^2 \times S^1$. The three-dimensional C-obstacle is the union of two-dimensional C-obstacle slices at angles $\theta \in [0, 2\pi)$. Even for this relatively low-dimensional C-space, an exact representation of the C-obstacle is quite complex. For this reason, C-obstacles are rarely described exactly.



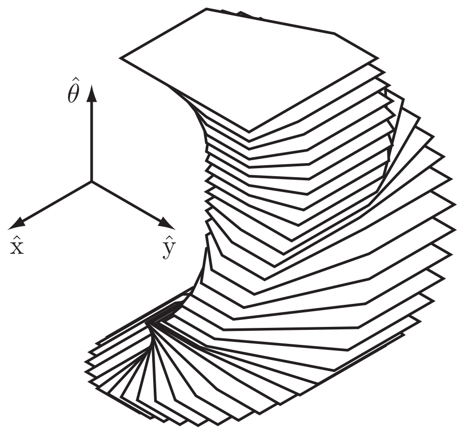


Figure 10.6: (Top) A triangular mobile robot that can both rotate and translate, represented by the configuration (x, y, θ). (Left) The C-space obstacle from Figure 10.5(b) when the robot is restricted to $\theta = 0$. (Right) The full three-dimensional C-space obstacle shown in slices at 10° increments.

10.2.2 Distance to Obstacles and Collision Detection

Given a C-obstacle $\mathcal{B}$ and a configuration q, let $d(q, \mathcal{B})$ be the distance between the robot and the obstacle, where

$$\begin{aligned} d(q, \mathcal{B}) &> 0 \qquad \text{(no contact with the obstacle)},\\ d(q, \mathcal{B}) &= 0 \qquad \text{(contact)},\\ d(q, \mathcal{B}) &< 0 \qquad \text{(penetration)}. \end{aligned}$$

The distance could be defined as the Euclidean distance between the two closest points of the robot and the obstacle, respectively.

A **distance-measurement algorithm** is one that determines $d(q, \mathcal{B})$. A **collision–detection routine** determines whether $d(q, \mathcal{B}_i) \leq 0$ for any C-obstacle $\mathcal{B}_i$. A collision-detection routine returns a binary result and may or may not utilize a distance-measurement algorithm at its core.

One popular distance-measurement algorithm is the Gilbert–Johnson–Keerthi (GJK) algorithm, which efficiently computes the distance between two convex bodies, possibly represented by triangular meshes. Any robot or obstacle can be treated as the union of multiple convex bodies. Extensions of this algorithm are

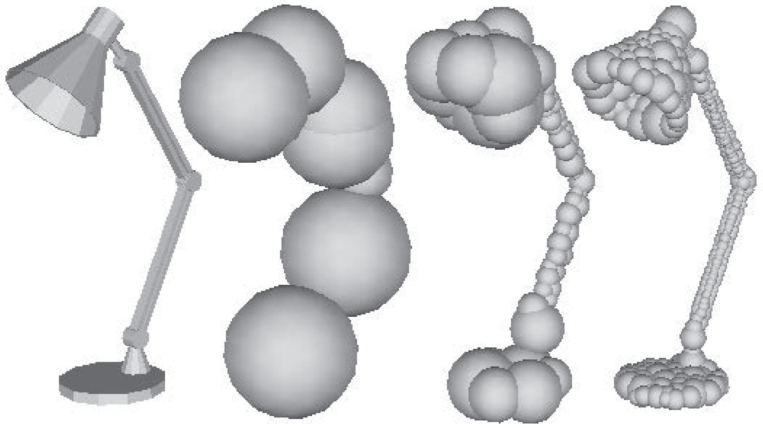


Figure 10.7: A lamp represented by spheres. The approximation improves as the number of spheres used to represent the lamp increases. Figure from [61] used with permission.

used in many distance-measurement algorithms and collision-detection routines for robotics, graphics, and game-physics engines.

A simpler approach is to approximate the robot and obstacles as unions of overlapping spheres. Approximations must always be **conservative** – the approximation must cover all points of the object – so that if a collision-detection routine indicates a free configuration q, then we are guaranteed that the actual geometry is collision-free. As the number of spheres in the representation of the robot and obstacles increases, the closer the approximations come to the actual geometry. An example is shown in Figure 10.7.

Given a robot at q represented by k spheres of radius R_i centered at $r_i(q)$, $i = 1, \ldots, k$, and an obstacle $\mathcal{B}$ represented by ℓ spheres of radius B_j centered at b_j, $j = 1, \ldots, \ell$, the distance between the robot and the obstacle can be calculated as

$$d(q, \mathcal{B}) = \min_{i,j} \|r_i(q) - b_j\| - R_i - B_j.$$

Apart from determining whether a particular configuration of the robot is in collision, another useful operation is determining whether the robot collides during a particular motion segment. While exact solutions have been developed for particular object geometries and motion types, the general approach is to sample the path at finely spaced points and to "grow" the robot to ensure that if two consecutive configurations are collision-free for the grown robot then the

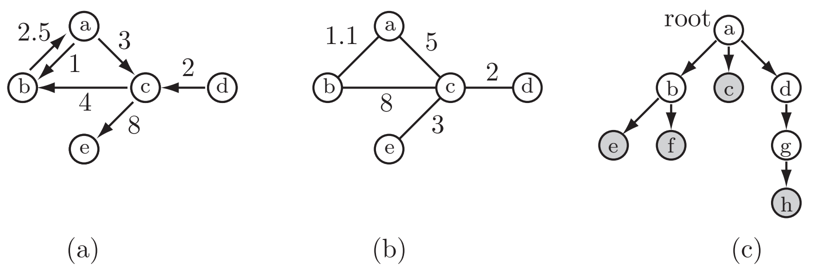


Figure 10.8: (a) A weighted digraph. (b) A weighted undirected graph. (c) A tree. The leaves are shaded gray.

volume swept out by the actual robot between the two configurations is also collision-free.

10.2.3 Graphs and Trees

Many motion planners explicitly or implicitly represent the C-space or state space as a **graph**. A graph consists of a collection of nodes $\mathcal{N}$ and a collection of edges $\mathcal{E}$, where each edge e connects two nodes. In motion planning, a node typically represents a configuration or state while an edge between nodes n_1 and n_2 indicates the ability to move from n_1 to n_2 without penetrating an obstacle or violating other constraints.

A graph can be either **directed** or **undirected**. In an undirected graph, each edge is bidirectional: if the robot can travel from n_1 to n_2 then it can also travel from n_2 to n_1. In a directed graph, or **digraph** for short, each edge allows travel in only one direction. The same two nodes can have two edges between them, allowing travel in opposite directions.

Graphs can also be **weighted** or **unweighted**. In a weighted graph, each edge has a positive cost associated with traversing it. In an unweighted graph each edge has the same cost (e.g., 1). Thus the most general type of graph we consider is a weighted digraph.

A **tree** is a digraph in which (1) there are no cycles and (2) each node has at most one **parent** node (i.e., at most one edge leading to the node). A tree has one **root** node with no parents and a number of **leaf** nodes with no **child**.

A digraph, undirected graph, and tree are illustrated in Figure 10.8.

Given N nodes, any graph can be represented by a matrix $A \in \mathbb{R}^{N \times N}$, where element a_{ij} of the matrix represents the cost of the edge from node i to node j; a zero or negative value indicates no edge between the nodes. Graphs and trees can be represented more compactly as a list of nodes, each with links to its neighbors.

10.2.4 Graph Search

Once the free space is represented as a graph, a motion plan can be found by searching the graph for a path from the start to the goal. One of the most powerful and popular graph search algorithms is A^* (pronounced "A star") search.

10.2.4.1 A^* Search

The A^* search algorithm efficiently finds a minimum-cost path on a graph when the cost of the path is simply the sum of the positive edge costs along the path.

Given a graph described by a set of nodes $\mathcal{N} = \{1, \ldots, N\}$, where node 1 is the start node, and a set of edges $\mathcal{E}$, the A^* algorithm makes use of the following data structures:

- a sorted list `OPEN` of the nodes from which exploration is still to be done, and a list `CLOSED` of nodes for which exploration has already taken place;
- a matrix `cost[node1,node2]` encoding the set of edges, where a positive value corresponds to the cost of moving from `node1` to `node2` (a negative value indicates that no edge exists);
- an array `past_cost[node]` of the minimum cost found so far to reach node `node` from the start node; and
- a search tree defined by an array `parent[node]`, which contains for each `node` a link to the node preceding it in the shortest path found so far from the start node to that `node`.

To initialize the search, the matrix `cost` is constructed to encode the edges, the list `OPEN` is initialized to the start node 1, the cost to reach the start node (`past_cost[1]`) is initialized to 0, and `past_cost[node]` for `node` $\in \{2, \ldots, N\}$ is initialized to infinity (or a large number), indicating that currently we have no idea of the cost of reaching those nodes.

At each step of the algorithm, the first node in `OPEN` is removed from `OPEN` and called `current`. The node `current` is also added to `CLOSED`. The first node in `OPEN` is one that minimizes the total estimated cost of the best path to the goal that passes through that node. The estimated cost is calculated as

```
est_total_cost[node] = past_cost[node]
                     + heuristic_cost_to_go(node)
```

where `heuristic_cost_to_go(node)` ≥ 0 is an optimistic (underestimating) estimate of the actual cost-to-go to the goal from `node`. For many path planning problems, an appropriate choice for the heuristic is the straight-line distance to the goal, ignoring any obstacles.

Because `OPEN` is a list sorted according to the estimated total cost, inserting a new node at the correct location in `OPEN` entails a small computational price.

If the node `current` is in the goal set then the search is finished and the path is reconstructed from the `parent` links. If not, for each neighbor `nbr` of `current` in the graph which is not also in `CLOSED`, the `tentative_past_cost` for `nbr` is calculated as `past_cost[current] + cost[current,nbr]`. If

```
tentative_past_cost < past_cost[nbr],
```

then `nbr` can be reached with less cost than previously known, so `past_cost[nbr]` is set to `tentative_past_cost` and `parent[nbr]` is set to `current`. The node `nbr` is then added (or moved) in `OPEN` according to its estimated total cost.

The algorithm then returns to the beginning of the main loop, removing the first node from `OPEN` and calling it `current`. If `OPEN` is empty then there is no solution.

The A^* algorithm is guaranteed to return a minimum-cost path, as nodes are only checked for inclusion in the goal set when they have the minimum total estimated cost of all nodes. If the node `current` is in the goal set then `heuristic_cost_to_go(current)` is zero and, since all edge costs are positive, we know that any path found in the future must have a cost greater than or equal to `past_cost[current]`. Therefore the path to `current` must be a shortest path. (There may be other paths of the same cost.)

If the heuristic "cost-to-go" is calculated exactly, considering obstacles, then A^* will explore from the minimum number of nodes necessary to solve the problem. Of course, calculating the cost-to-go exactly is equivalent to solving the path planning problem, so this is impractical. Instead, the heuristic cost-to-go should be calculated quickly and should be as close as possible to the actual cost-to-go to ensure that the algorithm runs efficiently. Using an optimistic cost-to-go ensures an optimal solution.

The A^* algorithm is an example of the general class of **best-first** searches, which always explore from the node currently deemed "best" by some measure.

The A^* search algorithm is described in pseudocode in Algorithm 10.1.

Algorithm 10.1 A^* search.

```
1: OPEN ← {1}
2: past_cost[1] ← 0, past_cost[node] ← infinity for node ∈ {2, ..., N}
3: while OPEN is not empty do
4:   current ← first node in OPEN, remove from OPEN
5:   add current to CLOSED
6:   if current is in the goal set then
7:     return SUCCESS and the path to current
8:   end if
9:   for each nbr of current not in CLOSED do
10:    tentative_past_cost ← past_cost[current]+cost[current,nbr]
11:    if tentative_past_cost < past_cost[nbr] then
12:      past_cost[nbr] ← tentative_past_cost
13:      parent[nbr] ← current
14:      put (or move) nbr in sorted list OPEN according to
              est_total_cost[nbr] ← past_cost[nbr] +
                      heuristic_cost_to_go(nbr)
15:    end if
16:  end for
17: end while
18: return FAILURE
```

10.2.4.2 Other Search Methods

- **Dijkstra's algorithm.** If the heuristic cost-to-go is always estimated as zero then A^* always explores from the OPEN node that has been reached with minimum past cost. This variant is called Dijkstra's algorithm, which preceded A^* historically. Dijkstra's algorithm is also guaranteed to find a minimum-cost path but on many problems it runs more slowly than A^* owing to the lack of a heuristic look-ahead function to help guide the search.

- **Breadth-first search.** If each edge in $\mathcal{E}$ has the same cost, Dijkstra's algorithm reduces to breadth-first search. All nodes one edge away from the start node are considered first, then all nodes two edges away, etc. The first solution found is therefore a minimum-cost path.

- **Suboptimal A^* search.** If the heuristic cost-to-go is overestimated by multiplying the optimistic heuristic by a constant factor $\eta > 1$, the A^* search will be biased to explore from nodes closer to the goal rather than nodes with a low past cost. This may cause a solution to be found more quickly but, unlike the case of an optimistic cost-to-go heuristic, the solution will not be guaranteed to be optimal. One possibility is to run A^* with an inflated cost-to-go to find an initial solution, then rerun the search with progressively smaller values of η until the time allotted for the search has expired or a solution is found with $\eta = 1$.

10.3 Complete Path Planners

Complete path planners rely on an exact representation of the free C-space $\mathcal{C}_{\text{free}}$. These techniques tend to be mathematically and algorithmically sophisticated, and impractical for many real systems, so we do not delve into them in detail.

One approach to complete path planning, which we will see in modified form in Section 10.5, is based on representing the complex high-dimensional space $\mathcal{C}_{\text{free}}$ by a one-dimensional **roadmap** R with the following properties:

(a) **Reachability**. From every point $q \in \mathcal{C}_{\text{free}}$, a free path to a point $q' \in R$ can be found trivially (e.g., a straight-line path).

(b) **Connectivity**. For each connected component of $\mathcal{C}_{\text{free}}$, there is one connected component of R.

With such a roadmap, the planner can find a path between any two points q_{start} and q_{goal} in the same connected component of $\mathcal{C}_{\text{free}}$ by simply finding paths from q_{start} to a point $q'_{\text{start}} \in R$, from a point $q'_{\text{goal}} \in R$ to q_{goal}, and from q'_{start} to q'_{goal} on the roadmap R. If a path can be found trivially between q_{start} and q_{goal}, the roadmap may not even be used.

While constructing a roadmap of $\mathcal{C}_{\text{free}}$ is complex in general, some problems admit simple roadmaps. For example, consider a polygonal robot translating among polygonal obstacles in the plane. As can be seen in Figure 10.5, the C-obstacles in this case are also polygons. A suitable roadmap is the weighted undirected **visibility graph**, with nodes at the vertices of the C-obstacles and edges between the nodes that can "see" each other (i.e., the line segment between the vertices does not intersect an obstacle). The weight associated with each edge is the Euclidean distance between the nodes.

Not only is this a suitable roadmap R, but it allows us to use an A^* search to find a shortest path between any two configurations in the same connected component of $\mathcal{C}_{\text{free}}$, as the shortest path is guaranteed either to be a straight line from q_{start} to q_{goal} or to consist of a straight line from q_{start} to a node $q'_{\text{start}} \in R$, a straight line from a node $q'_{\text{goal}} \in R$ to q_{goal}, and a path along the straight edges of R from q'_{start} to q'_{goal} (Figure 10.9). Note that the shortest

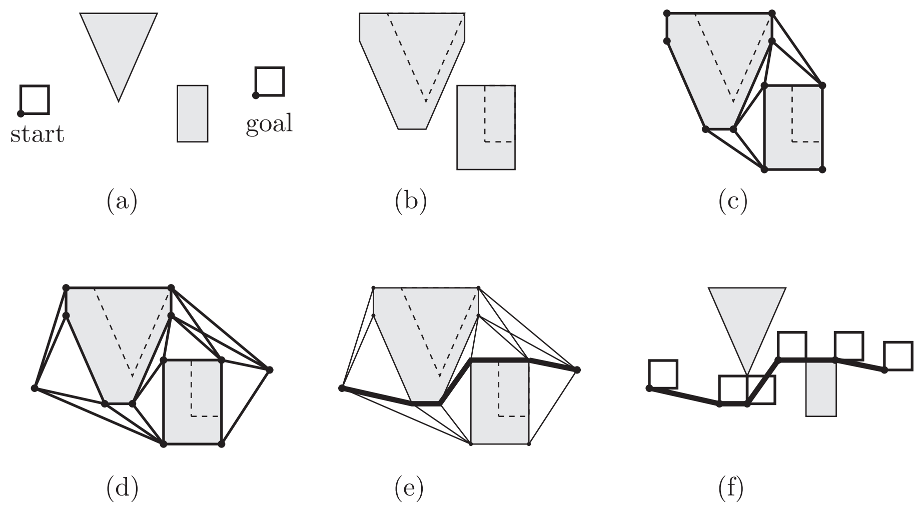


Figure 10.9: (a) The start and goal configurations for a square mobile robot (reference point shown) in an environment with a triangular and a rectangular obstacle. (b) The grown C-obstacles. (c) The visibility graph roadmap R of $\mathcal{C}_{\text{free}}$. (d) The full graph consists of R plus nodes at q_{start} and q_{goal}, along with the links connecting these nodes to the visible nodes of R. (e) Searching the graph results in the shortest path, shown in bold. (f) The robot is shown traversing the path.

path requires the robot to graze the obstacles, so we implicitly treat $\mathcal{C}_{\text{free}}$ as including its boundary.

10.4 Grid Methods

A search algorithm like A^* requires a discretization of the search space. The simplest discretization of C-space is a grid. For example, if the configuration space is n-dimensional and we desire k grid points along each dimension, the C-space is represented by k^n grid points.

The A^* algorithm can be used as a path planner for a C-space grid, with the following minor modifications:

- The definition of a "neighbor" of a grid point must be chosen: is the robot constrained to move in axis-aligned directions in configuration space or can it move in multiple dimensions simultaneously? For example, for a two-dimensional C-space, neighbors could be 4-connected (on the cardinal points of a compass: north, south, east, and west) or 8-connected (diagonals allowed), as shown in Figure 10.10(a). If diagonal motions are allowed, the cost to diagonal neighbors should be penalized appropriately. For example, the cost to a north, south, east or west neighbor could be 1, while the cost to a diagonal neighbor could be $\sqrt{2}$. If integers are desired, for efficiency of the implementation, the approximate costs 5 and 7 could be used.

- If only axis-aligned motions are used, the heuristic cost-to-go should be based on the **Manhattan distance**, not the Euclidean distance. The Manhattan distance counts the number of "city blocks" that must be traveled, with the rule that diagonals through a block are not possible (Figure 10.10(b)).

- A node `nbr` is added to `OPEN` only if the step from `current` to `nbr` is collision-free. (The step may be considered collision-free if a grown version of the robot at `nbr` does not intersect any obstacles.)

- Other optimizations are possible, owing to the known regular structure of the grid.

An A^* grid-based path planner is resolution-complete: it will find a solution if one exists at the level of discretization of the C-space. The path will be a shortest path subject to the allowed motions.

Figure 10.10(c) illustrates grid-based path planning for the 2R robot example of Figure 10.2. The C-space is represented as a grid with $k = 32$, i.e., there is a resolution of $360°/32 = 11.25°$ for each joint. This yields a total of $32^2 = 1024$ grid points.

The grid-based planner, as described, is a single-query planner: it solves each path planning query from scratch. However, if the same q_{goal} will be used in the same environment for multiple path planning queries, it may be worth preprocessing the entire grid to enable fast path planning. This is the **wavefront** planner, illustrated in Figure 10.11.

Although grid-based path planning is easy to implement, it is only appropriate for low-dimensional C-spaces. The reason is that the number of grid points, and hence the computational complexity of the path planner, increases exponentially with the number of dimensions n. For instance, a resolution $k = 100$ in a C-space with $n = 3$ dimensions leads to $k^n = 1$ million grid nodes, while $n = 5$ leads to 10 billion grid nodes and $n = 7$ leads to 100 trillion nodes. An alternative is to reduce the resolution k along each dimension, but this leads to a coarse representation of C-space that may miss free paths.

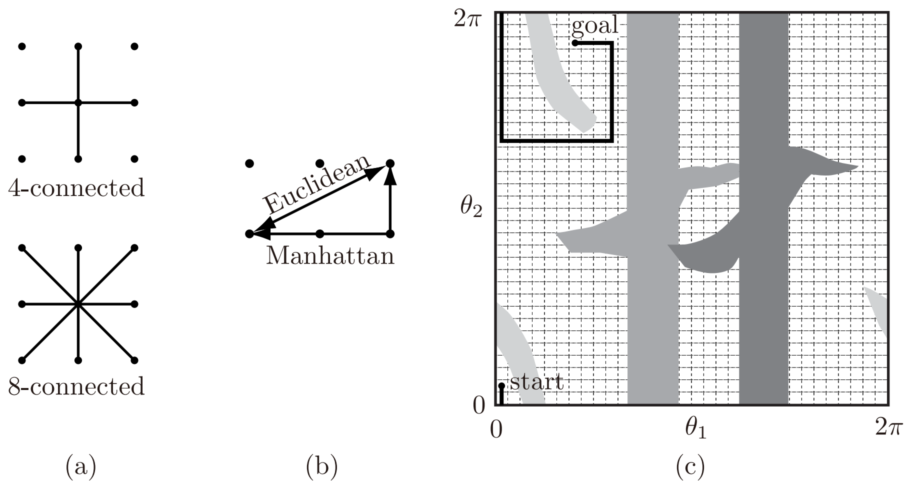


Figure 10.10: (a) A 4-connected grid point and an 8-connected grid point for a space $n = 2$. (b) Grid points spaced at unit intervals. The Euclidean distance between the two points indicated is $\sqrt{5}$ while the Manhattan distance is 3. (c) A grid representation of the C-space and a minimum-length Manhattan-distance path for the problem of Figure 10.2.

10	9	8	7	6	5	4	3	4	5	6	7	8	9	10
11	▨	▨	▨	▨	4	3	2	3	▨	▨	▨	7	8	9
12	13	14	▨	▨	3	2	1	2	▨	▨	▨	6	7	8
13	12	13	▨	▨	2	1	0	1	2	3	4	5	6	7
12	11	12	▨	▨	3	2	1	2	▨	▨	▨	▨	▨	8
11	10	▨	▨	▨	4	3	2	3	▨	▨	▨	▨	▨	9
10	9	8	7	6	5	4	3	4	▨	▨	▨	▨	▨	10

Figure 10.11: A wavefront planner on a two-dimensional grid. The goal configuration is given a score of 0. Then all collision-free 4-neighbors are given a score of 1. The process continues, breadth-first, with each free neighbor (that does not have a score already) assigned the score of its parent plus 1. Once every grid cell in the connected component of the goal configuration is assigned a score, planning from any location in the connected component is trivial: at every step, the robot simply moves "downhill" to a neighbor with a lower score. Grid points in collision receive a high score.

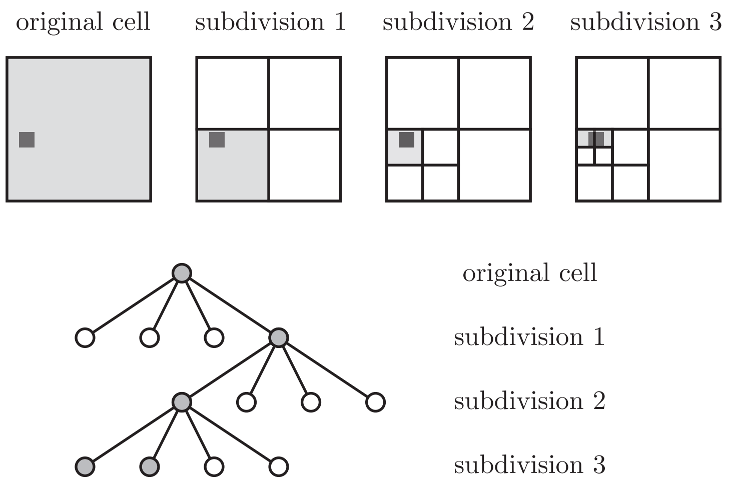


Figure 10.12: At the original C-space cell resolution, a small obstacle (indicated by the dark square) causes the whole cell to be labeled an obstacle. Subdividing the cell once shows that at least three-quarters of the cell is actually free. Three levels of subdivision results in a representation using ten total cells: four at subdivision level 3, three at subdivision level 2, and three at subdivision level 1. The cells shaded light gray are the obstacle cells in the final representation. The subdivision of the original cell is shown in the lower panel as a tree, specifically a quadtree, where the leaves of the tree are the final cells in the representation.

10.4.1 Multi-Resolution Grid Representation

One way to reduce the computational complexity of a grid-based planner is to use a multi-resolution grid representation of $\mathcal{C}_{\text{free}}$. Conceptually, a grid point is considered an obstacle if any part of the rectilinear cell centered on the grid point touches a C-obstacle. To refine the representation of the obstacle, an obstacle cell can be subdivided into smaller cells. Each dimension of the original cell is split in half, resulting in 2^n subcells for an n-dimensional space. Any cells that are still in contact with a C-obstacle are then subdivided further, up to a specified maximum resolution.

The advantage of this representation is that only the portions of C-space near obstacles are refined to high resolution, while those away from obstacles are represented by a coarse resolution. This allows the planner to find paths using short steps through cluttered spaces while taking large steps through wide open space. The idea is illustrated in Figure 10.12, which uses only 10 cells to represent an obstacle at the same resolution as a fixed grid that uses 64 cells.

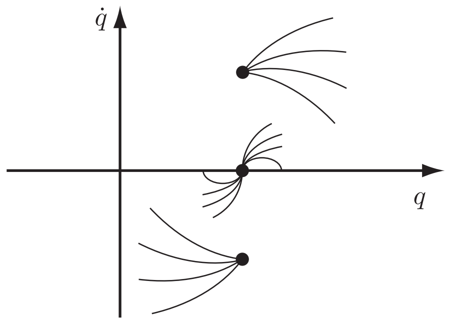


Figure 10.13: Sample trajectories emanating from three initial states in the phase space of a dynamic system with $q \in \mathbb{R}$. If the initial state has $\dot{q} > 0$, the trajectory cannot move to the left (corresponding to negative motion in q) instantaneously. Similarly, if the initial state has $\dot{q} < 0$, the trajectory cannot move to the right instantaneously.

For $n = 2$, this multi-resolution representation is called a **quadtree**, as each obstacle cell subdivides into $2^n = 4$ cells. For $n = 3$, each obstacle cell subdivides into $2^n = 8$ cells, and the representation is called an **octree**.

The multi-resolution representation of $\mathcal{C}_{\text{free}}$ can be built in advance of the search or incrementally as the search is being performed. In the latter case, if the step from `current` to `nbr` is found to be in collision, the step size can be halved until the step is free or the minimum step size is reached.

10.4.2 Grid Methods with Motion Constraints

The above grid-based planners operate under the assumption that the robot can go from one cell to any neighboring cell in a regular C-space grid. This may not be possible for some robots. For example, a car cannot reach, in one step, a "neighbor" cell that is to the side of it. Also, motions for a fast-moving robot arm should be planned in the state space, not just C-space, to take the arm dynamics into account. In the state space, the robot arm cannot move in certain directions (Figure 10.13).

Grid-based planners must be adapted to account for the motion constraints of the particular robot. In particular, the constraints may result in a directed graph. One approach is to discretize the robot controls while still making use of a grid on the C-space or state space, as appropriate. Details for a wheeled mobile robot and a dynamic robot arm are described next.

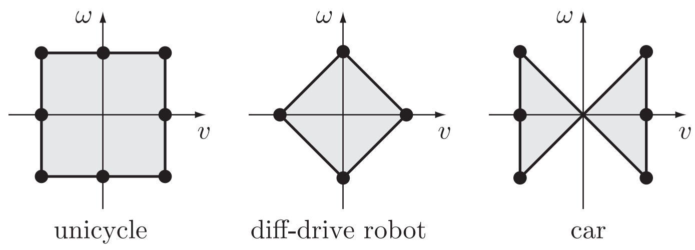


Figure 10.14: Discretizations of the control sets for unicycle, diff-drive, and car-like robots.

10.4.2.1 Grid-Based Path Planning for a Wheeled Mobile Robot

As described in Section 13.3, the controls for simplified models of unicycle, diff-drive, and car-like robots are (v, ω), i.e., the forward–backward linear velocity and the angular velocity. The control sets for these mobile robots are shown in Figure 10.14. Also shown are proposed discretizations of the controls, as dots. Other discretizations could be chosen.

Using the control discretization, we can use a variant of Dijkstra's algorithm to find short paths (Algorithm 10.2).

The search expands from q_{start} by integrating forward each control for a time Δt, creating new nodes for the paths that are collision-free. Each node keeps track of the control used to reach the node as well as the cost of the path to the node. The cost of the path to a new node is the sum of the cost of the previous node, `current`, plus the cost of the action.

Integration of the controls does not move the mobile robot to exact grid points. Instead, the C-space grid comes into play in lines 9 and 10. When a node is expanded, the grid cell it sits in is marked "occupied." Subsequently, any node in this occupied cell will be pruned from the search. This prevents the search from expanding nodes that are close by nodes reached with a lower cost.

No more than `MAXCOUNT` nodes, where `MAXCOUNT` is a value chosen by the user, are considered during the search.

The time Δt should be chosen so that each motion step is "small." The size of the grid cells should be chosen as large as possible while ensuring that integration of any control for a time Δt will move the mobile robot outside its current grid cell.

The planner terminates when `current` lies inside the goal region, or when there are no more nodes left to expand (perhaps because of obstacles), or when `MAXCOUNT` nodes have been considered. Any path found is optimal for the choice of cost function and other parameters to the problem. The planner actually

Algorithm 10.2 Grid-based Dijkstra planner for a wheeled mobile robot.

```
 1: OPEN ← {$q_{\text{start}}$}
 2: past_cost[$q_{\text{start}}$] ← 0
 3: counter ← 1
 4: while OPEN is not empty and counter < MAXCOUNT do
 5:   current ← first node in OPEN, remove from OPEN
 6:   if current is in the goal set then
 7:     return SUCCESS and the path to current
 8:   end if
 9:   if current is not in a previously occupied C-space grid cell then
10:     mark grid cell occupied
11:     counter ← counter + 1
12:     for each control in the discrete control set do
13:       integrate control forward a short time $\Delta t$ from current to $q_{\text{new}}$
14:       if the path to $q_{\text{new}}$ is collision-free then
15:         compute cost of the path to $q_{\text{new}}$
16:         place $q_{\text{new}}$ in OPEN, sorted by cost
17:         parent[$q_{\text{new}}$] ← current
18:       end if
19:     end for
20:   end if
21: end while
22: return FAILURE
```

runs faster in somewhat cluttered spaces, as the obstacles help to guide the exploration.

Some examples of motion plans for a car are shown in Figure 10.15.

10.4.2.2 Grid-Based Motion Planning for a Robot Arm

One method for planning the motion for a robot arm is to decouple the problem into a path planning problem followed by a time scaling of the path:

(a) Apply a grid-based or other path planner to find an obstacle-free path in C-space.

(b) Time scale the path to find the fastest trajectory that respects the robot's dynamics, as described in Section 9.4, or use any less aggressive time scaling.

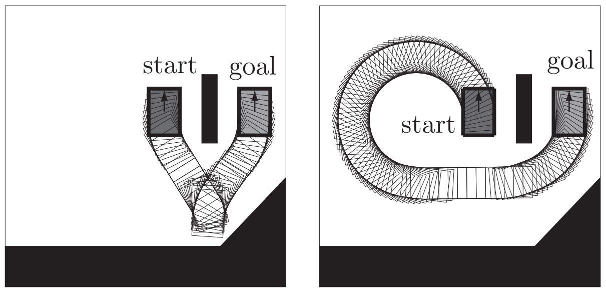


Figure 10.15: (Left) A minimum-cost path for a car-like robot where each action has identical cost, favoring a short path. (Right) A minimum-cost path where reversals are penalized. Penalizing reversals requires a modification to Algorithm 10.2.

Since the motion planning problem is broken into two steps (path planning followed by time scaling), the resultant motion will not be time-optimal in general.

Another approach is to plan directly in the state space. Given a state $(q, \dot{q})$ of the robot arm, let $\mathcal{A}(q, \dot{q})$ represent the set of accelerations that are feasible on the basis of the limited joint torques. To discretize the controls, the set $\mathcal{A}(q, \dot{q})$ is intersected with a grid of points of the form

$$\sum_{i=1}^{n} c a_i \hat{\mathrm{e}}_i,$$

where c is an integer, $a_i > 0$ is the acceleration step size in the $\ddot{q}_i$-direction, and $\hat{\mathrm{e}}_i$ is a unit vector in the ith direction (Figure 10.16).

As the robot moves, the acceleration set $\mathcal{A}(q, \dot{q})$ changes but the grid remains fixed. Because of this, and assuming a fixed integration time Δt at each "step" in a motion plan, the reachable states of the robot (after any integral number of steps) are confined to a grid in state space. To see this, consider a single joint angle of the robot, q_1, and assume for simplicity zero initial velocity, $\dot{q}_1(0) = 0$. The velocity at timestep k takes the form

$$\dot{q}_1(k) = \dot{q}_1(k-1) + c(k) a_1 \Delta t,$$

where $c(k)$ is any value in a finite set of integers. By induction, the velocity at any timestep must be of the form $a_1 k_v \Delta t$, where k_v is an integer. The position at timestep k takes the form

$$q_1(k) = q_1(k-1) + \dot{q}_1(k-1)\Delta t + \frac{1}{2} c(k) a_1 (\Delta t)^2.$$

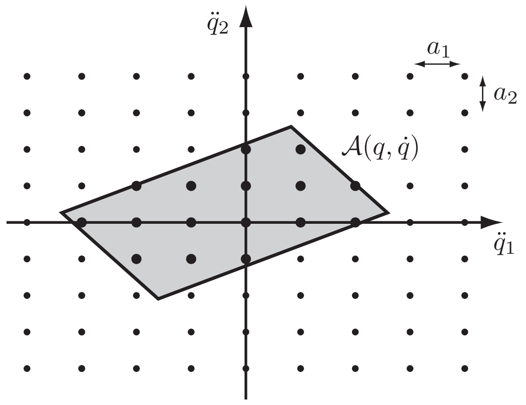


Figure 10.16: The instantaneously available acceleration set $\mathcal{A}(q, \dot{q})$ for a two-joint robot, intersected with a grid spaced at a_1 in $\ddot{q}_1$ and a_2 in $\ddot{q}_2$, gives the discretized control actions (shown as larger dots).

Substituting the velocity from the previous equation, we find that the position at any timestep must be of the form $a_1 k_p (\Delta t)^2/2 + q_1(0)$, where k_p is an integer.

To find a trajectory from a start node to a goal set, a breadth-first search can be employed to create a search tree on the state space nodes. When exploration is made from a node $(q, \dot{q})$ in the state space, the set $\mathcal{A}(q, \dot{q})$ is evaluated to find the discrete set of control actions. New nodes are created by integrating the control actions for time Δt. A node is discarded if the path to it is in collision or if it has been reached previously (i.e., by a trajectory taking the same or less time).

Because the joint angles and angular velocities are bounded, the state space grid is finite and therefore it can be searched in finite time. The planner is resolution-complete and returns a time-optimal trajectory, subject to the resolution specified in the control grid and timestep Δt.

The control-grid step sizes a_i must be chosen small enough that $\mathcal{A}(q, \dot{q})$, for any feasible state $(q, \dot{q})$, contains a representative set of points of the control grid. Choosing a finer grid for the controls, or a smaller timestep Δt, creates a finer grid in the state space and a higher likelihood of finding a solution amidst obstacles. It also allows the choice of a smaller goal set while keeping points of the state space grid inside the set.

Finer discretization comes at a computational cost. If the resolution of the control discretization is increased by a factor r in each dimension (i.e., each a_i is reduced to a_i/r), and the timestep size is divided by a factor τ, the computation time spent growing the search tree for a given robot motion duration increases by a factor $r^{n\tau}$, where n is the number of joints. For example, increasing the control-grid resolution by a factor $r = 2$ and decreasing the timestep by a factor $\tau = 4$ for a three-joint robot results in a search that is likely to take $2^{3\times4} = 4096$ times longer to complete. The high computational complexity of the planner makes it impractical beyond a few degrees of freedom.

The description above ignores one important issue: the feasible control set $\mathcal{A}(q, \dot{q})$ changes during a timestep, so the control chosen at the beginning of the timestep may no longer be feasible by the end of the timestep. For that reason, a conservative approximation $\tilde{\mathcal{A}}(q, \dot{q}) \subset \mathcal{A}(q, \dot{q})$ should be used instead. This set should remain feasible over the duration of a timestep regardless of which control action is chosen. How to determine such a conservative approximation $\tilde{\mathcal{A}}(q, \dot{q})$ is beyond the scope of this chapter, but it has to do with bounds on how rapidly the arm's mass matrix $M(q)$ changes with q and how fast the robot is moving. At low speeds $\dot{q}$ and short durations Δt, the conservative set $\tilde{\mathcal{A}}(q, \dot{q})$ is very close to $\mathcal{A}(q, \dot{q})$.

10.5 Sampling Methods

Each grid-based method discussed above delivers optimal solutions subject to the chosen discretization. A drawback of these approaches, however, is their high computational complexity, making them unsuitable for systems having more than a few degrees of freedom.

A different class of planners, known as sampling methods, relies on a random or deterministic function to choose a sample from the C-space or state space: a function to evaluate whether a sample or motion is in $\mathcal{X}_{\text{free}}$; a function to determine nearby previous free-space samples; and a simple local planner to try to connect to, or move toward, the new sample. These functions are used to build up a graph or tree representing feasible motions of the robot.

Sampling methods generally give up on the resolution-optimal solutions of a grid search in exchange for the ability to find satisficing solutions quickly in high-dimensional state spaces. The samples are chosen to form a roadmap or search tree that quickly approximates the free space $\mathcal{X}_{\text{free}}$ using fewer samples than would typically be required by a fixed high-resolution grid, where the number of grid points increases exponentially with the dimension of the search space. Most sampling methods are probabilistically complete: the probability of finding a solution, when one exists, approaches 100% as the number of samples goes to infinity.

Two major classes of sampling methods are rapidly exploring random trees (RRTs) and probabilistic roadmaps (PRMs). The former use a tree representation for single-query planning in either C-space or state space, while PRMs are primarily C-space planners that create a roadmap graph for multiple-query planning.

10.5.1 The RRT Algorithm

The RRT algorithm searches for a collision-free motion from an initial state x_{start} to a goal set $\mathcal{X}_{\text{goal}}$. It is applied to kinematic problems, where the state x is simply the configuration q, as well as to dynamic problems, where the state includes the velocity. The basic RRT grows a single tree from x_{start} as outlined in Algorithm 10.3.

Algorithm 10.3 RRT algorithm.

```
1: initialize search tree $T$ with $x_{\text{start}}$
2: while $T$ is less than the maximum tree size do
3:   $x_{\text{samp}} \leftarrow$ sample from $\mathcal{X}$
4:   $x_{\text{nearest}} \leftarrow$ nearest node in $T$ to $x_{\text{samp}}$
5:   employ a local planner to find a motion from $x_{\text{nearest}}$ to $x_{\text{new}}$ in
         the direction of $x_{\text{samp}}$
6:   if the motion is collision-free then
7:     add $x_{\text{new}}$ to $T$ with an edge from $x_{\text{nearest}}$ to $x_{\text{new}}$
8:     if $x_{\text{new}}$ is in $\mathcal{X}_{\text{goal}}$ then
9:       return SUCCESS and the motion to $x_{\text{new}}$
10:    end if
11:  end if
12: end while
13: return FAILURE
```

In a typical implementation for a kinematic problem (where x is simply q), the sampler in line 3 chooses x_{samp} randomly from an almost-uniform distribution over $\mathcal{X}$, with a slight bias toward states in $\mathcal{X}_{\text{goal}}$. The closest node x_{nearest} in the search tree T (line 4) is the node minimizing the Euclidean distance to x_{samp}. The state x_{new} (line 5) is chosen as the state a small distance d from x_{nearest} on the straight line to x_{samp}. Because d is small, a very simple local planner, e.g., one that returns a straight-line motion, will often find a motion connecting x_{nearest} to x_{new}. If the motion is collision-free, the new state x_{new} is added to the search tree T.

The net effect is that the nearly uniformly distributed samples "pull" the tree toward them, causing the tree to rapidly explore $\mathcal{X}_{\text{free}}$. An example of the effect of this pulling action on exploration is shown in Figure 10.17.

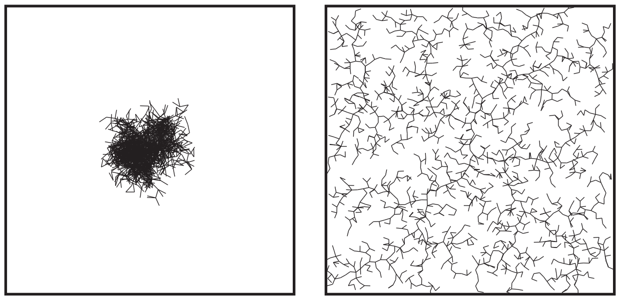


Figure 10.17: (Left) A tree generated by applying a uniformly-distributed random motion from a randomly chosen tree node does not explore very far. (Right) A tree generated by the RRT algorithm using samples drawn randomly from a uniform distribution. Both trees have 2000 nodes. Figure from [84] used with permission.

The basic algorithm leaves the programmer with many choices: how to sample from $\mathcal{X}$ (line 3), how to define the "nearest" node in T (line 4), and how to plan the motion to make progress toward x_{samp} (line 5). Even a small change to the sampling method, for example, can yield a dramatic change in the running time of the planner. A wide variety of planners have been proposed in the literature based on these choices and other variations. Some of these variations are described below.

10.5.1.1 Line 3: The Sampler

The most obvious sampler is one that samples randomly from a uniform distribution over $\mathcal{X}$. This is straightforward for Euclidean C-spaces $\mathbb{R}^n$, as well as for n-joint robot C-spaces $T^n = S^1 \times \cdots \times S^1$ (n times), where we can choose a uniform distribution over each joint angle, and for the C-space $\mathbb{R}^2 \times S^1$ for a mobile robot in the plane, where we can choose a uniform distribution over $\mathbb{R}^2$ and S^1 individually. The notion of a uniform distribution on some other curved C-spaces, for example $SO(3)$, is less straightforward.

For dynamic systems, a uniform distribution over the state space can be defined as the cross product of a uniform distribution over C-space and a uniform distribution over a bounded velocity set.

Although the name "rapidly-exploring random trees" is derived from the idea of a random sampling strategy, in fact the samples need not be generated randomly. For example, a deterministic sampling scheme that generates a progressively finer (multi-resolution) grid on $\mathcal{X}$ could be employed instead. To reflect this more general view, the approach has been called *rapidly-exploring*

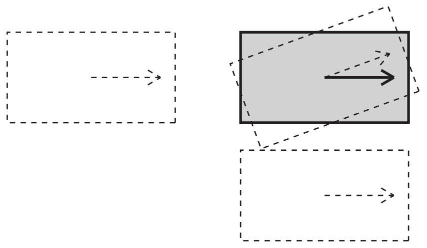


Figure 10.18: Which of the three dashed configurations of the car is "closest" to the configuration in gray?

dense trees (RDTs), emphasizing the key point that the samples should eventually become dense in the state space (i.e., as the number of samples goes to infinity, the samples become arbitrarily close to every point in $\mathcal{X}$).

10.5.1.2 Line 4: Defining the Nearest Node

Finding the "nearest" node depends on a definition of distance on $\mathcal{X}$. For an unconstrained kinematic robot on $\mathcal{C} = \mathbb{R}^n$, a natural choice for the distance between two points is simply the Euclidean distance. For other spaces, the choice is less obvious.

As an example, for a car-like robot with a C-space $\mathbb{R}^2 \times S^1$, which configuration is closest to the configuration x_{samp}: one that is rotated 20 degrees relative to x_{samp}, one that is 2 meters straight behind it, or one that is 1 meter straight to the side of it (Figure 10.18)? Since the motion constraints prevent spinning in place or moving directly sideways, the configuration that is 2 meters straight behind is best positioned to make progress toward x_{samp}. Thus defining a notion of distance requires

- combining components of different units (e.g., degrees, meters, degrees/s, meters/s) into a single distance measure; and
- taking into account the motion constraints of the robot.

The closest node x_{nearest} should perhaps be defined as the one that can reach x_{samp} the fastest, but computing this is as hard as solving the motion planning problem.

A simple choice of a distance measure from x to x_{samp} is the weighted sum of the distances along the different components of $x_{\text{samp}} - x$. The weights express the relative importance of the different components. If more is known about the set of states that the robot can reach from a state x in limited time, this information can be used in determining the nearest node. In any case, the nearest node should be computed quickly. Finding a nearest neighbor is a common problem in computational geometry, and various algorithms, such as kd trees and hashing, can be used to solve it efficiently.

10.5.1.3 Line 5: The Local Planner

The job of the local planner is to find a motion from x_{nearest} to some point x_{new} which is closer to x_{samp}. The planner should be simple and it should run quickly. Three examples are as follows.

A straight-line planner. The plan is a straight line to x_{new}, which may be chosen at x_{samp} or at a fixed distance d from x_{nearest} on the straight line to x_{samp}. This is suitable for kinematic systems with no motion constraints.

Discretized controls planner. For systems with motion constraints, such as wheeled mobile robots or dynamic systems, the controls can be discretized into a discrete set $\{u_1, u_2, \ldots\}$, as in the grid methods with motion constraints (Section 10.4.2 and Figures 10.14 and 10.16). Each control is integrated from x_{nearest} for a fixed time Δt using $\dot{x} = f(x, u)$. Among the new states reached without collision, the state that is closest to x_{samp} is chosen as x_{new}.

Wheeled robot planners. For a wheeled mobile robot, local plans can be found using Reeds–Shepp curves, as described in Section 13.3.3.

Other robot-specific local planners can be designed.

10.5.1.4 Other RRT Variants

The performance of the basic RRT algorithm depends heavily on the choice of sampling method, distance measure, and local planner. Beyond these choices, two other variants of the basic RRT are outlined below.

Bidirectional RRT. The bidirectional RRT grows two trees: one "forward" from x_{start} and one "backward" from x_{goal}. The algorithm alternates between growing the forward tree and growing the backward tree, and every so often it attempts to connect the two trees by choosing x_{samp} from the other tree. The advantage of this approach is that a single goal state x_{goal} can be reached exactly, rather than just a goal set $\mathcal{X}_{\text{goal}}$. Another advantage is that, in many environments, the two trees are likely to find each other much more quickly than a single "forward" tree will find a goal set.

The major problem is that the local planner might not be able to connect the two trees exactly. For example, the discretized controls planner of Section 10.5.1.3 is highly unlikely to create a motion exactly to a node in the other tree. In this case, the two trees may be considered more or less connected when points on each tree are sufficiently close. The "broken" discontinuous trajectory can be returned and patched by a smoothing method (Section 10.8).

RRT*. The basic RRT algorithm returns SUCCESS once a motion to $\mathcal{X}_{\text{goal}}$ is found. An alternative is to continue running the algorithm and to terminate the search only when another termination condition is reached (e.g., a maximum running time or a maximum tree size). Then the motion with the minimum cost can be returned. In this way, the RRT solution may continue to improve as time goes by. Because edges in the tree are never deleted or changed, however, the RRT generally does not converge to an optimal solution.

The RRT* algorithm is a variation on the single-tree RRT that continually rewires the search tree to ensure that it always encodes the shortest path from x_{start} to each node in the tree. The basic approach works for C-space path planning with no motion constraints, allowing exact paths from any node to any other node.

To modify the RRT to the RRT*, line 7 of the RRT algorithm, which inserts x_{new} in T with an edge from x_{nearest} to x_{new}, is replaced by a test of all the nodes $x \in \mathcal{X}_{\text{near}}$ in T that are sufficiently near to x_{new}. An edge to x_{new} is created from the $x \in \mathcal{X}_{\text{near}}$ by the local planner that (1) has a collision-free motion and (2) minimizes the total cost of the path from x_{start} to x_{new}, not just the cost of the added edge. The total cost is the cost to reach the candidate $x \in \mathcal{X}_{\text{near}}$ plus the cost of the new edge.

The next step is to consider each $x \in \mathcal{X}_{\text{near}}$ to see whether it could be reached at lower cost by a motion through x_{new}. If so, the parent of x is changed to x_{new}. In this way, the tree is incrementally rewired to eliminate high-cost motions in favor of the minimum-cost motions available so far.

The definition of $\mathcal{X}_{\text{near}}$ depends on the number of samples in the tree, details of the sampling method, the dimension of the search space, and other factors.

Unlike the RRT, the solution provided by RRT* approaches the optimal solution as the number of sample nodes increases. Like the RRT, the RRT* algorithm is probabilistically complete. Figure 10.19 demonstrates the rewiring behavior of RRT* compared to that of RRT for a simple example in $\mathcal{C} = \mathbb{R}^2$.

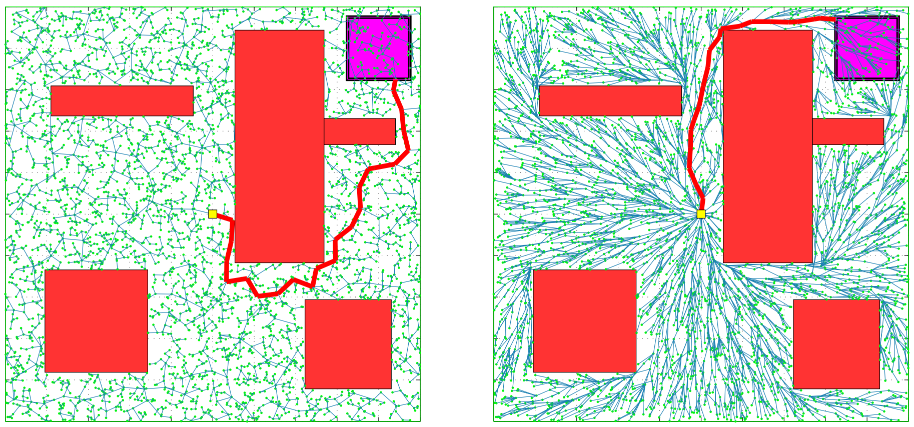


Figure 10.19: (Left) The tree generated by an RRT after 5,000 nodes. The goal region is the square at the top right corner, and the shortest path is indicated. (Right) The tree generated by RRT* after 5,000 nodes. Figure from [67] used with permission.

10.5.2 The PRM Algorithm

The PRM uses sampling to build a roadmap representation of $\mathcal{C}_{\text{free}}$ (Section 10.3) before answering any specific queries. The roadmap is an undirected graph: the robot can move in either direction along any edge exactly from one node to the next. For this reason, PRMs primarily apply to kinematic problems for which an exact local planner exists that can find a path (ignoring obstacles) from any q_1 to any other q_2. The simplest example is a straight-line planner for a robot with no kinematic constraints.

Once the roadmap is built, a particular start node q_{start} can be added to the graph by attempting to connect it to the roadmap, starting with the closest node. The same is done for the goal node q_{goal}. The graph is then searched for a path, typically using A^*. Thus the query can be answered efficiently once the roadmap has been built.

The use of PRMs allows the possibility of building a roadmap quickly and efficiently relative to constructing a roadmap using a high-resolution grid representation. The reason is that the volume fraction of the C-space that is "visible" by the local planner from a given configuration does not typically decrease exponentially with increasing dimension of the C-space.

The algorithm for constructing a roadmap R with N nodes is outlined in Algorithm 10.4 and illustrated in Figure 10.20.

A key choice in the PRM roadmap-construction algorithm is how to sample

Algorithm 10.4 PRM roadmap construction algorithm (undirected graph).

```
1: for $i = 1, \ldots, N$ do
2:   $q_i \leftarrow$ sample from $\mathcal{C}_{\text{free}}$
3:   add $q_i$ to $R$
4: end for
5: for $i = 1, \ldots, N$ do
6:   $\mathcal{N}(q_i) \leftarrow k$ closest neighbors of $q_i$
7:   for each $q \in \mathcal{N}(q_i)$ do
8:     if there is a collision-free local path from $q$ to $q_i$ and
       there is not already an edge from $q$ to $q_i$ then
9:       add an edge from $q$ to $q_i$ to the roadmap $R$
10:    end if
11:  end for
12: end for
13: return $R$
```

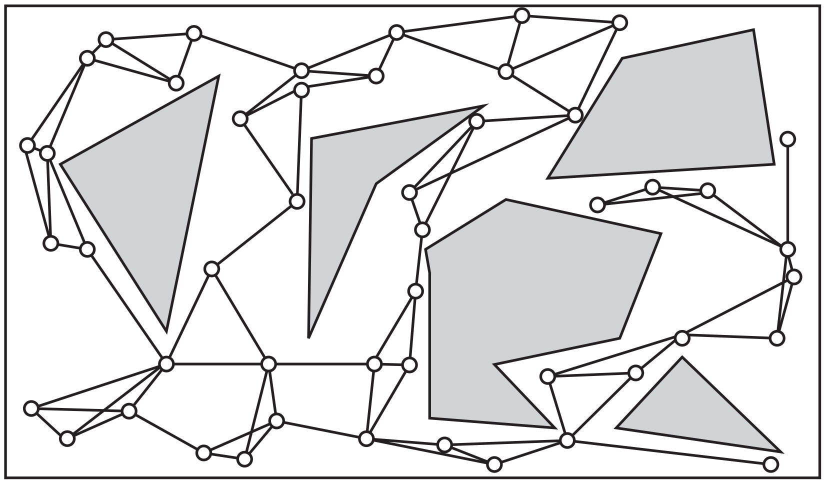


Figure 10.20: An example PRM roadmap for a point robot in $\mathcal{C} = \mathbb{R}^2$. The $k = 3$ closest neighbors are taken into consideration for connection to a sample node q. The degree of a node can be greater than three since it may be a close neighbor of many nodes.

from $\mathcal{C}_{\text{free}}$. While the default might be sampling randomly from a uniform distribution on $\mathcal{C}$ and eliminating configurations in collision, it has been shown that sampling more densely near obstacles can improve the likelihood of finding narrow passages, thus significantly reducing the number of samples needed to properly represent the connectivity of $\mathcal{C}_{\text{free}}$. Another option is deterministic multi-resolution sampling.

10.6 Virtual Potential Fields

Virtual potential field methods are inspired by potential energy fields in nature, such as gravitational and magnetic fields. From physics we know that a potential field $\mathcal{P}(q)$ defined over $\mathcal{C}$ induces a force $F = -\partial \mathcal{P}/\partial q$ that drives an object from high to low potential. For example, if h is the height above the Earth's surface in a uniform gravitational potential field ($g = 9.81$ m/s^2) then the potential energy of a mass m is $\mathcal{P}(h) = mgh$ and the force acting on it is $F = -\partial \mathcal{P}/\partial h = -mg$. The force will cause the mass to fall to the Earth's surface.

In robot motion control, the goal configuration q_{goal} is assigned a low virtual potential and obstacles are assigned a high virtual potential. Applying a force to the robot proportional to the negative gradient of the virtual potential naturally pushes the robot toward the goal and away from the obstacles.

A virtual potential field is very different from the planners we have seen so far. Typically the gradient of the field can be calculated quickly, so the motion can be calculated in real time (reactive control) instead of planned in advance. With appropriate sensors, the method can even handle obstacles that move or appear unexpectedly. The drawback of the basic method is that the robot can get stuck in local minima of the potential field, away from the goal, even when a feasible motion to the goal exists. In certain cases it is possible to design the potential to guarantee that the only local minimum is at the goal, eliminating this problem.

10.6.1 A Point in C-space

Let's begin by assuming a point robot in its C-space. A goal configuration q_{goal} is typically encoded by a quadratic potential energy "bowl" with zero energy at the goal,

$$\mathcal{P}_{\text{goal}}(q) = \frac{1}{2}(q - q_{\text{goal}})^{\mathrm{T}} K (q - q_{\text{goal}}),$$

where K is a symmetric positive-definite weighting matrix (for example, the identity matrix). The force induced by this potential is

$$F_{\text{goal}}(q) = -\frac{\partial \mathcal{P}_{\text{goal}}}{\partial q} = K(q_{\text{goal}} - q),$$

an attractive force proportional to the distance from the goal.

The repulsive potential induced by a C-obstacle $\mathcal{B}$ can be calculated from the distance $d(q, \mathcal{B})$ to the obstacle (Section 10.2.2):

$$\mathcal{P}_{\mathcal{B}}(q) = \frac{k}{2d^2(q, \mathcal{B})}, \tag{10.3}$$

where $k > 0$ is a scaling factor. The potential is only properly defined for points outside the obstacle, $d(q, \mathcal{B}) > 0$. The force induced by the obstacle potential is

$$F_{\mathcal{B}}(q) = -\frac{\partial \mathcal{P}_{\mathcal{B}}}{\partial q} = \frac{k}{d^3(q, \mathcal{B})} \frac{\partial d}{\partial q}.$$

The total potential is obtained by summing the attractive goal potential and the repulsive obstacle potentials,

$$\mathcal{P}(q) = \mathcal{P}_{\text{goal}}(q) + \sum_i \mathcal{P}_{\mathcal{B}_i}(q),$$

yielding a total force

$$F(q) = F_{\text{goal}}(q) + \sum_i F_{\mathcal{B}_i}(q).$$

Note that the sum of the attractive and repulsive potentials may not give a minimum (zero force) exactly at q_{goal}. Also, it is common to put a bound on the maximum potential and force, as the simple obstacle potential (10.3) would otherwise yield unbounded potentials and forces near the boundaries of obstacles.

Figure 10.21 shows a potential field for a point in $\mathbb{R}^2$ with three circular obstacles. The contour plot of the potential field clearly shows the global minimum near the center of the space (near the goal marked with a +), a local minimum near the two obstacles on the left, as well as saddles (critical points that are a maximum in one direction and a minimum in the other direction) near the obstacles. Saddles are generally not a problem, as a small perturbation allows continued progress toward the goal. Local minima away from the goal are a problem, however, as they attract nearby states.

To actually control the robot using the calculated $F(q)$, we have several options, two of which are:

- Apply the calculated force plus damping,

$$u = F(q) + B\dot{q}. \tag{10.4}$$

  If B is positive definite then it dissipates energy for all $\dot{q} \neq 0$, reducing oscillation and guaranteeing that the robot will come to rest. If $B = 0$, the

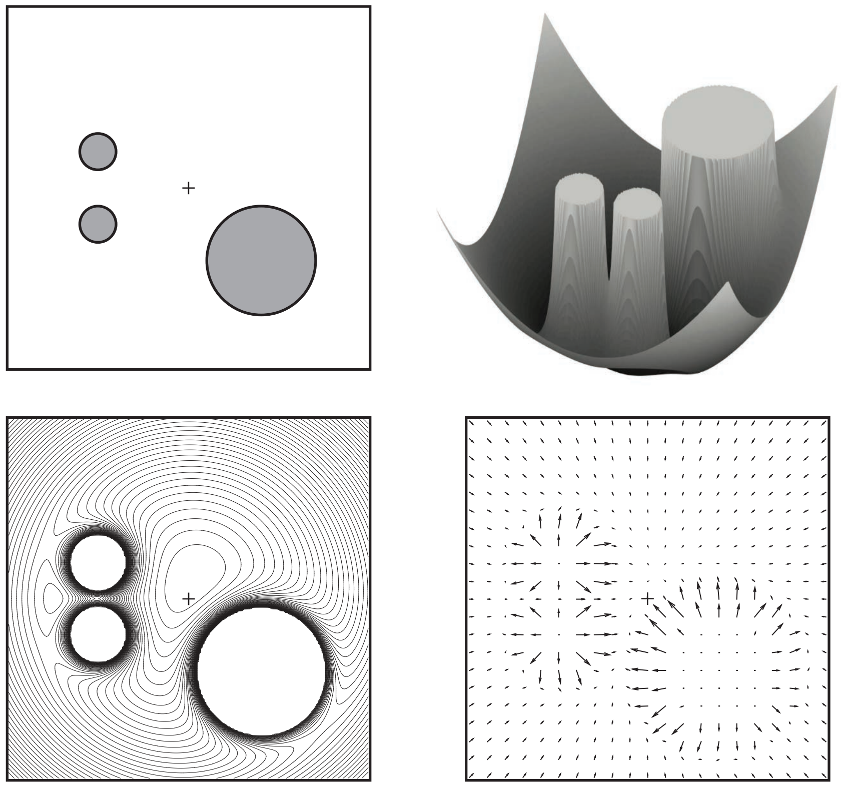


Figure 10.21: (Top left) Three obstacles and a goal point, marked with a +, in $\mathbb{R}^2$. (Top right) The potential function summing the bowl-shaped potential pulling the robot to the goal with the repulsive potentials of the three obstacles. The potential function saturates at a specified maximum value. (Bottom left) A contour plot of the potential function, showing the global minimum, a local minimum, and four saddles: between each obstacle and the boundary of the workspace, and between the two small obstacles. (Bottom right) Forces induced by the potential function.

robot continues to move while maintaining constant total energy, which is the sum of the initial kinetic energy $\frac{1}{2}\dot{q}^{\mathrm{T}}(0)M(q(0))\dot{q}(0)$ and the initial virtual potential energy $\mathcal{P}(q(0))$.

The motion of the robot under the control law (10.4) can be visualized as a ball rolling in gravity on the potential surface of Figure 10.21, where the dissipative force is rolling friction.

- Treat the calculated force as a commanded velocity instead:

$$\dot{q} = F(q). \tag{10.5}$$

  This automatically eliminates oscillations.

Using the simple obstacle potential (10.3), even distant obstacles have a nonzero effect on the motion of the robot. To speed up evaluation of the repulsive terms, distant obstacles could be ignored. We can define a range of influence of the obstacles $d_{\text{range}} > 0$ so that the potential is zero for all $d(q, \mathcal{B}) \geq d_{\text{range}}$:

$$U_{\mathcal{B}}(q) = \begin{cases} \dfrac{k}{2}\left(\dfrac{d_{\text{range}} - d(q,\mathcal{B})}{d_{\text{range}} d(q,\mathcal{B})}\right)^2 & \text{if } d(q,\mathcal{B}) < d_{\text{range}} \\ 0 & \text{otherwise.} \end{cases}$$

Another issue is that $d(q, \mathcal{B})$ and its gradient are generally difficult to calculate. An approach to dealing with this is described in Section 10.6.3.

10.6.2 Navigation Functions

A significant problem with the potential field method is local minima. While potential fields may be appropriate for relatively uncluttered spaces or for rapid response to unexpected obstacles, they are likely to get the robot stuck in local minima for many practical applications.

One method that avoids this issue is the wavefront planner of Figure 10.11. The wavefront algorithm creates a local-minimum-free potential function by a breadth-first traversal of every cell reachable from the goal cell in a grid representation of the free space. Therefore, if a solution exists to the motion planning problem then simply moving "downhill" at every step is guaranteed to bring the robot to the goal.

Another approach to local-minimum-free gradient following is based on replacing the virtual potential function with a **navigation function**. A navigation function $\varphi(q)$ is a type of virtual potential function that

1. is smooth (or at least twice differentiable) on q;
2. has a bounded maximum value (e.g., 1) on the boundaries of all obstacles;
3. has a single minimum at q_{goal}; and
4. has a full-rank Hessian $\partial^2\varphi/\partial q^2$ at all critical points q where $\partial\varphi/\partial q = 0$ (i.e., $\varphi(q)$ is a **Morse** function).

Condition 1 ensures that the Hessian $\partial^2\varphi/\partial q^2$ exists. Condition 2 puts an upper bound on the virtual potential energy of the robot. The key conditions are 3 and 4. Condition 3 ensures that of the critical points of $\varphi(q)$ (including minima, maxima, and saddles), there is only one minimum, at q_{goal}. This ensures that q_{goal} is at least locally attractive. There may be saddle points that are minima along a subset of directions, but condition 4 ensures that the set of initial states that are attracted to any saddle point has empty interior (zero measure), and therefore almost every initial state converges to the unique minimum q_{goal}.

While constructing navigation potential functions with only a single minimum is nontrivial, [152] showed how to construct them for the particular case of an n-dimensional $\mathcal{C}_{\text{free}}$ consisting of all points inside an n-sphere of radius R and outside smaller spherical obstacles $\mathcal{B}_i$ of radius r_i centered at q_i, i.e., $\{q \in \mathbb{R}^n \mid \|q\| \leq R \text{ and } \|q - q_i\| > r_i \text{ for all } i\}$. This is called a **sphere world**. While a real C-space is unlikely to be a sphere world, Rimon and Koditschek showed that the boundaries of the obstacles, and the associated navigation function, can be deformed to a much broader class of **star-shaped** obstacles. A star-shaped obstacle is one that has a center point from which the line segment to any point on the obstacle boundary is contained completely within the obstacle. A **star world** is a star-shaped C-space which has star-shaped obstacles. Thus finding a navigation function for an arbitrary star world reduces to finding a navigation function for a "model" sphere world that has centers at the centers of the star-shaped obstacles, then stretching and deforming that navigation function to one that fits the star world. Rimon and Koditschek gave a systematic procedure to accomplish this.

Figure 10.22 shows a deformation of a navigation function on a model sphere world to a star world for the case $\mathcal{C} \subset \mathbb{R}^2$.

10.6.3 Workspace Potential

A difficulty in calculating the repulsive force from an obstacle is obtaining the distance to the obstacle, $d(q, \mathcal{B})$. One approach that avoids an exact calculation is to represent the boundary of an obstacle as a set of point obstacles, and to represent the robot by a small set of control points. Let the Cartesian location of control point i on the robot be written $f_i(q) \in \mathbb{R}^3$ and boundary point j of the obstacle be $c_j \in \mathbb{R}^3$. Then the distance between the two points is $\|f_i(q) - c_j\|$, and the potential at the control point i due to the obstacle point j is

$$\mathcal{P}'_{ij}(q) = \frac{k}{2\|f_i(q) - c_j\|^2},$$

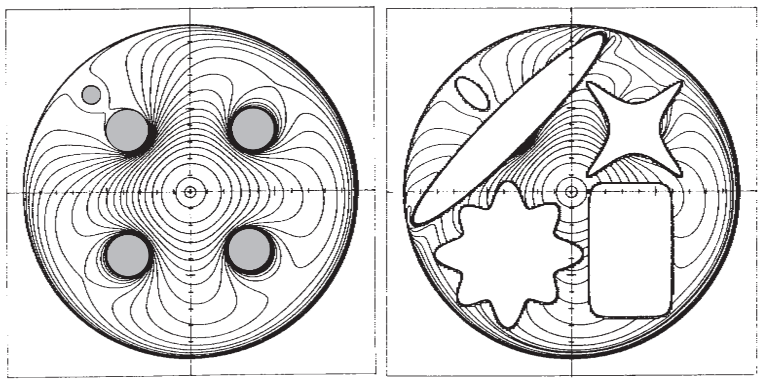


Figure 10.22: (Left) A model “sphere world” with five circular obstacles. The contour plot of a navigation function is shown. The goal is at $(0,0)$. Note that the obstacles induce saddle points near the obstacles, but no local minima. (Right) A “star world” obtained by deforming the obstacles and the potential while retaining a navigation function. Figure from [152] used with permission from the American Mathematical Society.

yielding the repulsive force at the control point

$$F'_{ij}(q) = -\frac{\partial \mathcal{P}'_{ij}}{\partial q} = \frac{k}{\|f_i(q) - c_j\|^4} \left(\frac{\partial f_i}{\partial q}\right)^{\mathrm{T}} (f_i(q) - c_j) \in \mathbb{R}^3.$$

To turn the linear force $F'_{ij}(q) \in \mathbb{R}^3$ into a generalized force $F_{ij}(q) \in \mathbb{R}^n$ acting on the robot arm or mobile robot, we first find the Jacobian $J_i(q) \in \mathbb{R}^{3\times n}$ relating $\dot{q}$ to the linear velocity of the control point $\dot{f}_i$:

$$\dot{f}_i = \frac{\partial f_i}{\partial q}\dot{q} = J_i(q)\dot{q}.$$

By the principle of virtual work, the generalized force $F_{ij}(q) \in \mathbb{R}^n$ due to the repulsive linear force $F'_{ij}(q) \in \mathbb{R}^3$ is simply

$$F_{ij}(q) = J_i^{\mathrm{T}}(q) F'_{ij}(q).$$

Now the total force $F(q)$ acting on the robot is the sum of the easily calculated attractive force $F_{\mathrm{goal}}(q)$ and the repulsive forces $F_{ij}(q)$ for all i and j.

10.6.4 Wheeled Mobile Robots

The preceding analysis assumes that a control force $u = F(q) + B\dot{q}$ (control law (10.4)) or a velocity $\dot{q} = F(q)$ (control law (10.5)) can be applied in any direction. If the robot is a wheeled mobile robot subject to rolling constraints $A(q)\dot{q} = 0$, however, the calculated $F(q)$ must be projected to controls $F_{\text{proj}}(q)$ that move the robot tangentially to the constraints. For a kinematic robot employing the control law $\dot{q} = F_{\text{proj}}(q)$, a suitable projection is

$$F_{\text{proj}}(q) = \Big(I - A^{\mathrm{T}}(q)\big(A(q)A^{\mathrm{T}}(q)\big)^{-1}A(q)\Big)F(q).$$

For a dynamic robot employing the control law $u = F_{\text{proj}}(q) + B\dot{q}$, the projection was discussed in Section 8.7.

10.6.5 Use of Potential Fields in Planners

A potential field can be used in conjunction with a path planner. For example, a best-first search such as A^* can use the potential as an estimate of the cost-to-go. Incorporating a search function prevents the planner from getting permanently stuck in local minima.

10.7 Nonlinear Optimization

The motion planning problem can be expressed as a general nonlinear optimization, with equality and inequality constraints, taking advantage of a number of software packages to solve such problems. Nonlinear optimization problems can be solved by gradient-based methods, such as sequential quadratic programming (SQP), or non-gradient methods, such as simulated annealing, Nelder–Mead optimization, and genetic programming. Like many nonlinear optimization problems, these methods are not generally guaranteed to find a feasible solution when one exists, let alone an optimal one. For methods that use gradients of the objective function and constraints, however, we can expect a locally optimal solution if we start the process with a guess that is "close" to a solution.

The general problem can be written as follows:

$$\begin{aligned}
\text{find} \quad & u(t), q(t), T && && (10.6)\\
\text{minimizing} \quad & J(u(t), q(t), T) && && (10.7)\\
\text{subject to} \quad & \dot{x}(t) = f(x(t), u(t)), && \forall t \in [0, T], && (10.8)\\
& u(t) \in \mathcal{U}, && \forall t \in [0, T], && (10.9)\\
& q(t) \in \mathcal{C}_{\text{free}}, && \forall t \in [0, T], && (10.10)\\
& x(0) = x_{\text{start}}, && && (10.11)\\
& x(T) = x_{\text{goal}}. && && (10.12)
\end{aligned}$$

To solve this problem approximately by nonlinear optimization, the control $u(t)$, trajectory $q(t)$, and equality and inequality constraints (10.8)–(10.12) must be discretized. This is typically done by ensuring that the constraints are satisfied at a fixed number of points distributed evenly over the interval $[0, T]$ and choosing a finite-parameter representation of the position and/or control histories. We have at least three choices of how to parametrize the position and controls:

(a) **Parametrize the trajectory $q(t)$.** In this case, we solve for the trajectory parameters directly. The controls $u(t)$ at any time are calculated using the equations of motion. This approach does not apply to systems with fewer controls than configuration variables, $m < n$.

(b) **Parametrize the control $u(t)$.** We solve for $u(t)$ directly. Calculating the state $x(t)$ requires integrating the equations of motion.

(c) **Parametrize both $q(t)$ and $u(t)$.** We have a larger number of variables, since we parametrize both $q(t)$ and $u(t)$. Also, we have a larger number of constraints, as $q(t)$ and $u(t)$ must satisfy the dynamic equations $\dot{x} = f(x, u)$ explicitly, typically at a fixed number of points distributed evenly over the interval $[0, T]$. We must be careful to choose the parametrizations of $q(t)$ and $u(t)$ to be consistent with each other, so that the dynamic equations can be satisfied at these points.

A trajectory or control history can be parametrized in any number of ways. The parameters can be the coefficients of a polynomial in time, the coefficients of a truncated Fourier series, spline coefficients, wavelet coefficients, piecewise constant acceleration or force segments, etc. For example, the control $u_i(t)$ could be represented by $p + 1$ coefficients a_j of a polynomial in time:

$$u_i(t) = \sum_{j=0}^{p} a_j t^j.$$

In addition to the parameters for the state or control history, the total time T may be another control parameter. The choice of parametrization has implications for the efficiency of the calculation of $q(t)$ and $u(t)$ at a given time t. It also determines the sensitivity of the state and control to the parameters and whether each parameter affects the profiles at all times $[0, T]$ or just on a finite-time support base. These are important factors in the stability and efficiency of the numerical optimization.

10.8 Smoothing

The axis-aligned motions of a grid planner and the randomized motions of sampling planners may lead to jerky motion of a robot. One approach to dealing with this issue is to let the planner handle the work of searching globally for a solution, then post-process the resulting motion to make it smoother.

There are many ways to do this; two possibilities are outlined below.

Nonlinear Optimization While gradient-based nonlinear optimization may fail to find a solution if initialized with a random initial trajectory, it can make an effective post-processing step, since the plan initializes the optimization with a "reasonable" solution. The initial motion must be converted to a parametrized representation of the controls, and the cost $J(u(t), q(t), T)$ can be expressed as a function of $u(t)$ or $q(t)$. For example, the cost function

$$J = \frac{1}{2} \int_0^T \dot{u}^{\mathrm{T}}(t)\dot{u}(t)dt$$

penalizes rapidly changing controls. This has an analogy in human motor control, where the smoothness of human arm motions has been attributed to minimization of the rate of change of torques at the joints [188].

Subdivide and Reconnect A local planner can be used to attempt a connection between two distant points on a path. If this new connection is collision-free, it replaces the original path segment. Since the local planner is designed to produce short, smooth, paths, the new path is likely to be shorter and smoother than the original. This test-and-replace procedure can be applied iteratively to randomly chosen points on the path. Another possibility is to use a recursive procedure that subdivides the path first into two pieces and attempts to replace each piece with a shorter path; then, if either portion cannot be replaced by a shorter path, it subdivides again; and so on.

10.9 Summary

- A fairly general statement of the motion planning problem is as follows. Given an initial state $x(0) = x_{\text{start}}$ and a desired final state x_{goal}, find a time T and a set of controls $u : [0, T] \to \mathcal{U}$ such that the motion satisfies $x(T) \in \mathcal{X}_{\text{goal}}$ and $q(x(t)) \in \mathcal{C}_{\text{free}}$ for all $t \in [0, T]$.

- Motion planning problems can be classified in the following categories: path planning versus motion planning; fully actuated versus constrained or underactuated; online versus offline; optimal versus satisficing; exact versus approximate; with or without obstacles.

- Motion planners can be characterized by the following properties: multiple-query versus single-query; anytime planning or not; complete, resolution complete, probabilistically complete, or none of the above; and their degree of computational complexity.

- Obstacles partition the C-space into free C-space, $\mathcal{C}_{\text{free}}$, and obstacle space, $\mathcal{C}_{\text{obs}}$, where $\mathcal{C} = \mathcal{C}_{\text{free}} \cup \mathcal{C}_{\text{obs}}$. Obstacles may split $\mathcal{C}_{\text{free}}$ into separate connected components. There is no feasible path between configurations in different connected components.

- A conservative check of whether a configuration q is in collision uses a simplified "grown" representation of the robot and obstacles. If there is no collision between the grown bodies, then the configuration is guaranteed collision-free. Checking whether a path is collision-free usually involves sampling the path at finely spaced points and ensuring that if the individual configurations are collision-free then the swept volume of the robot path is collision-free.

- The C-space geometry is often represented by a graph consisting of nodes and edges between the nodes, where edges represent feasible paths. The graph can be undirected (edges flow in both directions) or directed (edges flow in only one direction). Edges can be unweighted or weighted according to their cost of traversal. A tree is a directed graph with no cycles in which each node has at most one parent.

- A roadmap path planner uses a graph representation of $\mathcal{C}_{\text{free}}$, and path planning problems can be solved using a simple path from q_{start} onto the roadmap, a path along the roadmap, and a simple path from the roadmap to q_{goal}.

- The A^* algorithm is a popular search method that finds minimum-cost paths on a graph. It operates by always exploring from a node that is (1) unexplored and (2) on a path with minimum estimated total cost. The estimated total cost is the sum of the weights for the edges encountered in reaching the node from the start node plus an estimate of the cost-to-go to the goal. To ensure that the search returns an optimal solution, the cost-to-go estimate should be optimistic.

- A grid-based path planner discretizes the C-space into a graph consisting of neighboring points on a regular grid. A multi-resolution grid can be used to allow large steps in wide open spaces and smaller steps near obstacle boundaries.

- Discretizing the control set allows robots with motion constraints to take advantage of grid-based methods. If integrating a control does not land the robot exactly on a grid point, the new state may still be pruned if a state in the same grid cell has already been achieved with a lower cost.

- The basic RRT algorithm grows a single search tree from x_{start} to find a motion to $\mathcal{X}_{\text{goal}}$. It relies on a sampler to find a sample x_{samp} in $\mathcal{X}$, an algorithm to find the closest node x_{nearest} in the search tree, and a local planner to find a motion from x_{nearest} to a point closer to x_{samp}. The sampling is chosen to cause the tree to explore $\mathcal{X}_{\text{free}}$ quickly.

- The bidirectional RRT grows a search tree from both x_{start} and x_{goal} and attempts to join them up. The RRT* algorithm returns solutions that tend toward the optimal as the planning time goes to infinity.

- The PRM builds a roadmap of $\mathcal{C}_{\text{free}}$ for multiple-query planning. The roadmap is built by sampling $\mathcal{C}_{\text{free}}$ N times, then using a local planner to attempt to connect each sample with several of its nearest neighbors. The roadmap is searched using A^*.

- Virtual potential fields are inspired by potential energy fields such as gravitational and electromagnetic fields. The goal point creates an attractive potential while obstacles create repulsive potentials. The total potential $\mathcal{P}(q)$ is the sum of these, and the virtual force applied to the robot is $F(q) = -\partial\mathcal{P}/\partial q$. The robot is controlled by applying this force plus damping or by simulating first-order dynamics and driving the robot with $F(q)$ as a velocity. Potential field methods are conceptually simple but may get the robot stuck in local minima away from the goal.

- A navigation function is a potential function with no local minima. Navigation functions result in near-global convergence to q_{goal}. While they are difficult to design in general, they can be designed systematically for certain environments.

- Motion planning problems can be converted to general nonlinear optimization problems with equality and inequality constraints. While optimization methods can be used to find smooth near-optimal motions, they tend to get stuck in local minima in cluttered C-spaces. Optimization methods typically require a good initial guess at a solution.

- Motions returned by grid-based and sampling-based planners tend to be jerky. Smoothing the plan using nonlinear optimization or subdivide-and-reconnect can improve the quality of the motion.

10.10 Notes and References

Excellent textbooks covering motion planning broadly include the original text by Latombe [80] in 1991 and the more recent texts by Choset et al. [27] and LaValle [83]. Other summaries of the state-of-the-art in motion planning can be found in the Handbook of Robotics [70], and, particularly for robots subject to nonholonomic and actuation constraints, in the Control Handbook [101], the Encyclopedia of Systems and Control [100], and the textbook by Murray et al. [122]. Search algorithms and other algorithms for artificial intelligence are covered in detail by Russell and Norvig [155].

Landmark early work on motion planning for Shakey the Robot at SRI led to the development of A^* search in 1968 by Hart, Nilsson, and Raphael [53]. This work built on the newly established approach to dynamic programming for optimal decision-making, as described by Bellman and Dreyfus [10], and it improved on the performance of Dijkstra's algorithm [37]. A suboptimal anytime variant of A^* was proposed in [90]. Early work on multiresolution path planning is described in [65, 96, 45, 54] based on hierarchical decompositions of C-space [156].

One early line of work focused on exact characterization of the free C-space in the presence of obstacles. The visibility graph approach for polygons moving among polygons was developed by Lozano-Pérez and Wesley in 1979 [97]. In more general settings, researchers used sophisticated algorithms and mathematical methods to develop cellular decompositions and exact roadmaps of the free C-space. Important examples of this work are a series of papers by Schwartz and Sharir on the piano movers' problem [159, 160, 161] and Canny's PhD thesis [23].

As a result of the mathematical sophistication and high computational complexity needed to exactly represent the topology of C-spaces, a movement formed in the 1990s to approximately represent C-space using samples, and that movement carries on strong today. That line of work has followed two main branches, probabilistic roadmaps (PRMs) [69] and rapidly exploring random trees (RRTs) [84, 86, 85]. Due to their ability to handle complex high-dimensional C-spaces relatively efficiently, research in sampling-based planners has exploded, and some of the subsequent work is summarized in [27, 83]. The bidirectional RRT and RRT*, highlighted in this chapter, are described in [83] and [68], respectively.

The grid-based approach to motion planning for a wheeled mobile robot was introduced by Barraquand and Latombe [8], and the grid-based approach to time-optimal motion planning for a robot arm with dynamic constraints was introduced in [24, 40, 39].

The GJK algorithm for collision detection was derived in [50]. Open-source collision-detection packages are implemented in the Open Motion Planning Library (OMPL) [181] and the Robot Operating System (ROS). An approach to approximating polyhedra with spheres for fast collision detection is described in [61].

The potential field approach to motion planning and real-time obstacle avoidance was first introduced by Khatib and is summarized in [73]. A search-based planner using a potential field to guide the search is described by Barraquand et al. [7]. The construction of navigation functions, potential functions with a unique local minimum, is described in a series of papers by Koditschek and Rimon [78, 76, 77, 152, 153].

Nonlinear optimization-based motion planning has been formulated in a number of publications, including the classic computer graphics paper by Witkin and Kass [194] using optimization to generate the motions of an animated jumping lamp; work on generating motion plans for dynamic nonprehensile manipulation [103]; Newton algorithms for optimal motions of mechanisms [88]; and more recent developments in short-burst sequential action control, which solves both the motion planning and feedback control problems [3, 187]. Path smoothing for mobile robot paths by subdivide and reconnect is described by Laumond et al. [82].

10.11 Exercises

Exercise 10.1 One path is **homotopic** to another if it can be continuously deformed into the other without moving the endpoints. In other words, it can be stretched and pulled like a rubber band, but it cannot be cut and pasted back together. For the C-space in Figure 10.2, draw a path from the start to the goal that is not homotopic to the one shown.

Exercise 10.2 Label the connected components in Figure 10.2. For each connected component, draw a picture of the robot for one configuration in the connected component.

Exercise 10.3 Assume that θ_2 joint angles in the range $[175°, 185°]$ result in

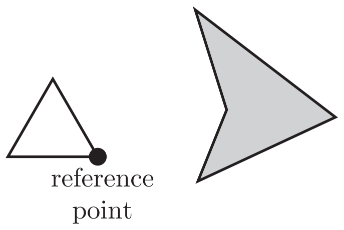


Figure 10.23: Exercise 10.4.

self-collision for the robot of Figure 10.2. Draw the new joint-limit C-obstacle on top of the existing C-obstacles and label the resulting connected components of $\mathcal{C}_{\text{free}}$. For each connected component, draw a picture of the robot for one configuration in the connected component.

Exercise 10.4 Draw the C-obstacle corresponding to the obstacle and translating planar robot in Figure 10.23.

Exercise 10.5 Write a program that accepts as input the coordinates of a polygonal robot (relative to a reference point on the robot) and the coordinates of a polygonal obstacle and produces as output a drawing of the corresponding C-space obstacle. In Mathematica, you may find the function `ConvexHull` useful. In MATLAB, try `convhull`.

Exercise 10.6 Calculating a square root can be computationally expensive. For a robot and an obstacle represented as collections of spheres (Section 10.2.2), provide a method for calculating the distance between the robot and obstacle that minimizes the use of square roots.

Exercise 10.7 Draw the visibility roadmap for the C-obstacles and q_{start} and q_{goal} in Figure 10.24. Indicate the shortest path.

Exercise 10.8 Not all edges of the visibility roadmap described in Section 10.3 are needed. Prove that an edge between two vertices of C-obstacles need not be included in the roadmap if either end of the edge does not hit the obstacle tangentially (i.e., it hits at a concave vertex). In other words, if the edge ends

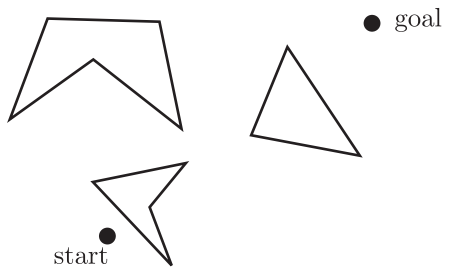


Figure 10.24: Planning problem for Exercise 10.7.

by "colliding" with an obstacle, it will never be used in a shortest path.

Exercise 10.9 Implement an A^* path planner for a point robot in a plane with obstacles. The planar region is a 100×100 area. The program generates a graph consisting of N nodes and E edges, where N and E are chosen by the user. After generating N randomly chosen nodes, the program should connect randomly chosen nodes by edges until E unique edges have been generated. The cost associated with each edge is the Euclidean distance between the nodes. Finally, the program should display the graph, search the graph using A^* for the shortest path between nodes 1 and N, and display the shortest path or indicate FAILURE if no path exists. The heuristic cost-to-go is the Euclidean distance to the goal.

Exercise 10.10 Modify the A^* planner in Exercise 10.9 to use a heuristic cost-to-go equal to ten times the distance to the goal node. Compare the running time with the original A^* when they are run on the same graphs. (You may need to use large graphs to notice any effect.) Are the solutions found with the new heuristic optimal?

Exercise 10.11 Modify the A^* algorithm from Exercise 10.9 to use Dijkstra's algorithm instead. Comment on the relative running times of A^* and Dijkstra's algorithm when each is run on the same graphs.

Exercise 10.12 Write a program that accepts the vertices of polygonal obstacles from a user, as well as the specification of a 2R robot arm, rooted at $(x, y) = (0, 0)$, with link lengths L_1 and L_2. Each link is simply a line segment.

Generate the C-space obstacles for the robot by sampling the two joint angles at k-degree intervals (e.g., $k = 5$) and checking for intersections between the line segments and the polygon. Plot the obstacles in the workspace, and in the C-space grid use a black square or dot at each configuration colliding with an obstacle. (Hint: At the core of this program is a subroutine to see whether two line segments intersect. If the segments' corresponding infinite lines intersect, you can check whether this intersection is within the line segments.)

Exercise 10.13 Write an A^* grid path planner for a 2R robot with obstacles and display on the C-space the paths you find. (See Exercise 10.12 and Figure 10.10.)

Exercise 10.14 Implement the grid-based path planner for a wheeled mobile robot (Algorithm 10.2), given the control discretization. Choose a simple method to represent obstacles and check for collisions. Your program should plot the obstacles and show the path that is found from the start to the goal.

Exercise 10.15 Write an RRT planner for a point robot moving in a plane with obstacles. Free space and obstacles are represented by a two-dimensional array, where each element corresponds to a grid cell in the two-dimensional space. The occurrence of a 1 in an element of the array means that there is an obstacle there, and a 0 indicates that the cell is in free space. Your program should plot the obstacles, the tree that is formed, and show the path that is found from the start to the goal.

Exercise 10.16 Do the same as for the previous exercise, except that obstacles are now represented by line segments. The line segments can be thought of as the boundaries of obstacles.

Exercise 10.17 Write a PRM planner to solve the same problem as in Exercise 10.15.

Exercise 10.18 Write a program to implement a virtual potential field for a 2R robot in an environment with point obstacles. The two links of the robot are line segments, and the user specifies the goal configuration of the robot, the start configuration of the robot, and the location of the point obstacles in the workspace. Put two control points on each link of the robot and transform the workspace potential forces to configuration space potential forces. In one workspace figure, draw an example environment consisting of a few point obstacles and the robot at its start and goal configurations. In a second C-space figure, plot the potential function as a contour plot over (θ_1, θ_2), and overlay a planned path from a start configuration to a goal configuration. The robot uses the kinematic control law $\dot{q} = F(q)$.

See whether you can create a planning problem that results in convergence to an undesired local minimum for some initial arm configurations but succeeds in finding a path to the goal for other initial arm configurations.