6 Planning and Navigation

6.1 Introduction

This book has focused on the elements of a mobile robot that are critical to robust mobility: the kinematics of locomotion, sensors for determining the robot's environmental context, and techniques for localizing with respect to its map. We now turn our attention to the robot's cognitive level. Cognition generally represents the purposeful decision making and execution that a system utilizes to achieve its highest-order goals.

In the case of a mobile robot, the specific aspect of cognition directly linked to robust mobility is *navigation competence*. Given partial knowledge about its environment and a goal position or series of positions, navigation encompasses the ability of the robot to act based on its knowledge and sensor values so as to reach its goal positions as efficiently and as reliably as possible. The focus of this chapter is how the tools of the previous chapters can be combined to solve this navigation problem.

Within the mobile robotics research community, a great many approaches have been proposed for solving the navigation problem. As we sample from this research background, it will become clear that in fact there are strong similarities between all of these approaches, even though they appear, on the surface, quite disparate. The key difference between various navigation architectures is the manner in which they decompose the problem into smaller subunits. In sections 6.3, 6.4, and 6.5, we describe the most popular of these architectures, contrasting their relative strengths and weaknesses.

First, however, in section 6.2 we discuss two additional key competences required for mobile robot navigation. Given a map and a goal location, *path planning* involves identifying a trajectory that will cause the robot to reach the goal location when executed. Path planning is a strategic problem-solving competence, since the robot must decide what to do over the long term to achieve its goals.

The second competence is equally important but occupies the opposite, tactical extreme. Given real-time sensor readings, *obstacle avoidance* means modulating the trajectory of the robot in order to avoid collisions. A great variety of approaches have demonstrated competent obstacle avoidance, and we survey a number of these approaches as well.

6.2 Competences for Navigation: Planning and Reacting

In the artificial intelligence community, planning and reacting are often viewed as contrary approaches or even opposites. When applied to physical systems such as mobile robots, however, planning and reacting have a strong complementarity, each being critical to the other's success. The navigation challenge for a robot involves executing a course of action (or plan) to reach its goal position. During execution, the robot must react to unforeseen events (e.g., obstacles) in such a way as to still reach the goal. Without reacting, the planning effort will not pay off because the robot will never physically reach its goal. Without planning, the reacting effort cannot guide the overall robot behavior to reach a distant goal—again, the robot will never reach its goal.

An information-theoretic formulation of the navigation problem will make this complementarity clear. Suppose that a robot R at time i has a map M_i and an initial belief state b_i. The robot's goal is to reach a position p while satisfying some temporal constraints: $loc_g(R) = p \; ; (g \leq n)$. Thus, the robot must be at location p at or before timestep n.

Although the goal of the robot is distinctly physical, the robot can only really sense its belief state, not its physical location, and therefore we map the goal of reaching location p to reaching a belief state b_g, corresponding to the belief that $loc_g(R) = p$. With this formulation, a plan q is nothing more than one or more trajectories from b_i to b_g. In other words, plan q will cause the robot's belief state to transition from b_i to b_g if the plan is executed from a world state consistent with both b_i and M_i.

Of course, the problem is that the latter condition may not be met. It is entirely possible that the robot's position is not quite consistent with b_i, and it is even likelier that M_i is either incomplete or incorrect. Furthermore, the real-world environment is dynamic. Even if M_i is correct as a single snapshot in time, the planner's model regarding how M changes over time is usually imperfect.

In order to reach its goal nonetheless, the robot must incorporate new information gained during plan execution. As time marches forward, the environment changes and the robot's sensors gather new information. This is precisely where reacting becomes relevant. In the best of cases, reacting will modulate robot behavior locally in order to correct the planned-upon trajectory so that the robot still reaches the goal. At times, unanticipated new information will require changes to the robot's strategic plans, and so ideally the planner also incorporates new information as that new information is received.

Taken to the limit, the planner would incorporate every new piece of information in real time, instantly producing a new plan that in fact reacts to the new information appropriately. This extreme, at which point the concept of planning and the concept of reacting merge, is called *integrated planning and execution* and is discussed in section 6.5.4.3.

Completeness. A useful concept throughout this discussion of robot architecture involves whether particular design decisions sacrifice the system's ability to achieve a desired goal whenever a solution exists. This concept is termed *completeness*. More formally, the robot system is *complete* if and only if, for all possible problems (i.e., initial belief states, maps, and goals), when there exists a trajectory to the goal belief state, the system will achieve the goal belief state (see [40] for further details). Thus when a system is incomplete, then there is at least one example problem for which, although there is a solution, the system fails to generate a solution. As you may expect, achieving completeness is an ambitious goal. Often, completeness is sacrificed for computational complexity at the level of representation or reasoning. Analytically, it is important to understand how completeness is compromised by each particular system.

In the following sections, we describe key aspects of planning and reacting as they apply to mobile robot path planning and obstacle avoidance and describe how representational decisions impact the potential completeness of the overall system. For greater detail, refer to [32, 44, chapter 25].

6.3 Path Planning

Even before the advent of affordable mobile robots, the field of path planning was heavily studied because of its applications in the area of industrial manipulator robotics. Interestingly, the path-planning problem for a manipulator with, for instance, six degrees of freedom is far more complex than that of a differential-drive robot operating in a flat environment. Therefore, although we can take inspiration from the techniques invented for manipulation, the path-planning algorithms used by mobile robots tend to be simpler approximations owing to the greatly reduced degrees of freedom. Furthermore, industrial robots often operate at the fastest possible speed because of the economic impact of high throughput on a factory line. So, the dynamics and not just the kinematics of their motions are significant, further complicating path planning and execution. In contrast, a number of mobile robots operate at such low speeds that dynamics are rarely considered during path planning, further simplifying the mobile robot instantiation of the problem.

Configuration space. Path planning for manipulator robots and, indeed, even for most mobile robots, is formally done in a representation called *configuration space*. Suppose that a robot arm (e.g., SCARA robot) has k degrees of freedom. Every state or configuration of the robot can be described with k real values: q_1, …, q_k. The k-values can be regarded as a point p in a k-dimensional space called the configuration space C of the robot. This description is convenient because it allows us to describe the complex 3D shape of the robot with a single k-dimensional point.

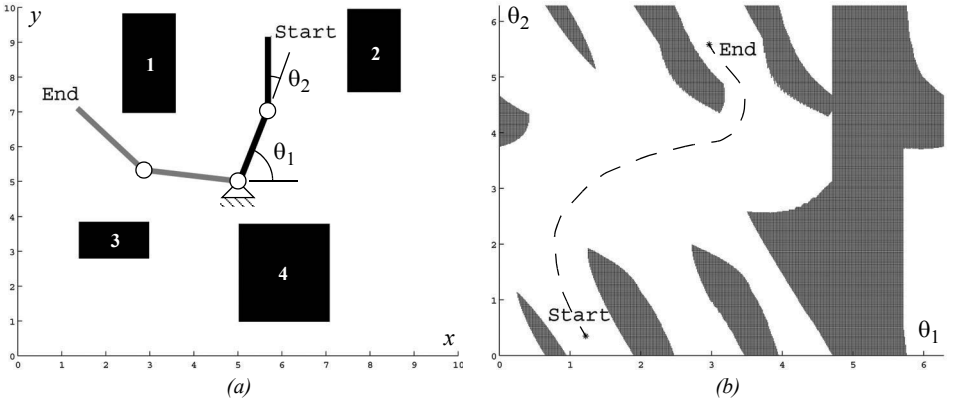

Figure 6.1
Physical space (a) and configuration space (b): (a) A two-link planar robot arm has to move from the configuration *start* to *end*. The motion is thereby constraint by the obstacles 1 to 4. (b) The corresponding configuration space shows the free space in joint coordinates (angle θ_1 and θ_2) and a path that achieves the goal.

Now consider the robot arm moving in an environment where the workspace (i.e., its physical space) contains known obstacles. The goal of path planning is to find a path in the physical space from the initial position of the arm to the goal position, avoiding all collisions with the obstacles. This is a difficult problem to visualize and solve in the physical space, particularly as k grows large. But in configuration space the problem is straightforward. If we define the *configuration space obstacle* O as the subspace of C where the robot arm bumps into something, we can compute the free space $F = C - O$ in which the robot can move safely.

Figure 6.1 shows a picture of the physical space and configuration space for a planar robot arm with two links. The robot's goal is to move its end effector from position *start* to *end*. The configuration space depicted is 2D because each of two joints can have any position from 0 to 2π. It is easy to see that the solution in C-space is a line from *start* to *end* that remains always within the free space of the robot arm.

For mobile robots operating on flat ground, we generally represent robot position with three variables (x, y, θ), as in chapter 3. But, as we have seen, most robots are nonholonomic, using differential-drive systems or Ackerman steered systems. For such robots, the nonholonomic constraints limit the robot's velocity $(\dot{x}, \dot{y}, \dot{\theta})$ in each configuration (x, y, θ). For details regarding the construction of the appropriate *free space* to solve such path-planning problems, see [32, p. 405].

In mobile robotics, the most common approach is to assume for path-planning purposes that the robot is in fact holonomic, simplifying the process tremendously. This is especially

common for differential-drive robots because they can rotate in place, and so a holonomic path can be easily mimicked if the rotational position of the robot is not critical.

Furthermore, mobile roboticists will often plan under the further assumption that the robot is simply a *point*. Thus we can further reduce the configuration space for mobile robot path planning to a 2D representation with just x- and y-axes. The result of all this simplification is that the configuration space looks essentially identical to a 2D (i.e., flat) version of the physical space, with one important difference. Because we have reduced the robot to a point, we must inflate each obstacle by the size of the robot's radius to compensate. With this new, simplified configuration space in mind, we can now introduce common techniques for mobile robot path planning.

Path-planning overview. The robot's environment representation can range from a continuous geometric description to a decomposition-based geometric map or even a topological map, as described in section 5.5. The first step of any path-planning system is thus to transform this possibly continuous environmental model into a discrete map suitable for the chosen path-planning algorithm. Path planners differ in how they use this discrete decomposition. In this book, we describe two general strategies:

1. Graph search: a connectivity graph in free space is first constructed and then searched. The graph construction process is often performed offline.
2. Potential field planning: a mathematical function is imposed directly on the free space. The gradient of this function can then be followed to the goal.

6.3.1 Graph search

Graph search techniques have traditionally been strongly rooted in the field of mathematics. Nonetheless, in recent years much of the innovation has been devised in the robotics community. This may be largely attributed to the need for real-time capable algorithms, which can accommodate evolving maps and thus changing graphs. For most of these methods we distinguish two main steps: graph construction, where nodes are placed and connected via edges, and graph search, where the computation of an (optimal) solution is performed.

6.3.1.1 Graph construction

Starting from a representation of free and occupied space, several methods are known to decompose this representation into a graph that can then be searched using any of the algorithms described in section 6.3.1.2 and 6.3.1.3. The challenge lies in constructing a set of nodes and edges that enable the robot to go anywhere in its free space while limiting the total size of the graph.

First, we describe two road map approaches that achieve this result with dramatically different types of roads. In the case of the *visibility graph*, roads come as close as possible

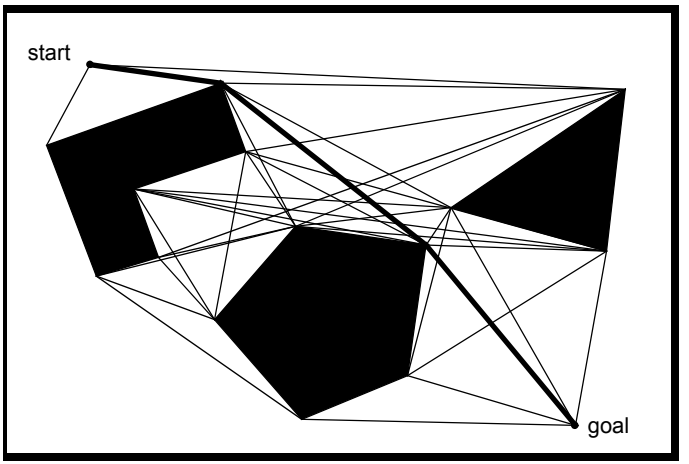

Figure 6.2
Visibility graph [32]. The nodes of the graph are the initial and goal points and the vertices of the configuration space obstacles (polygons). All nodes which are visible from each other are connected by straight-line segments, defining the road map. This means there are also edges along each polygon's sides.

to obstacles and resulting optimal paths are minimum-length solutions. In the case of the *Voronoi diagram*, roads stay as far away as possible from obstacles. We then detail cell decomposition methods where the idea is to discriminate between free and occupied geometric areas. Exact cell decomposition is a lossless decomposition, whereas approximate cell decomposition represents an approximation of the original map. A graph is then formed through a specified connectivity relation between cells. Finally, we describe the construction of lattice graphs, which are formed by shifting an underlying base set of edges over the free space. Lattice graphs are typically constructed by employing a mathematical model of the robot so that their edges become directly executable.

Visibility graph. The visibility graph for a polygonal configuration space C consists of edges joining all pairs of vertices that can see each other (including both the initial and goal positions as vertices as well). The unobstructed straight lines (roads) joining those vertices are obviously the shortest distances between them. The task of the path planner is then to find a (shortest) path from the initial position to the goal position along the roads defined by the visibility graph (figure 6.2).

Visibility graphs are moderately popular in mobile robotics, partly because their implementation is quite simple. Particularly when the environmental representation describes

objects in the environment as polygons in either continuous or discrete space, the visibility graph can employ the obstacle polygon descriptions readily.

There are, however, two important caveats when employing visibility graph search. First, the size of the representation and the number of edges and nodes increase with the number of obstacle polygons. Therefore, the method is extremely fast and efficient in sparse environments, but it can be slow and inefficient compared to other techniques when used in densely populated environments.

The second caveat is a much more serious potential flaw: solution paths found by graph search tend to take the robot as close as possible to obstacles on the way to the goal. More formally, we can prove that shortest solutions on the visibility graph are *optimal* in terms of path length. This powerful result also means that all sense of safety, with respect to staying a reasonable distance from obstacles, is sacrificed for this optimality. The common solution is to grow obstacles by significantly more than the robot's radius, or, alternatively, to modify the solution path after path planning to distance the path from obstacles where possible. Of course such actions sacrifice the optimal-length results of visibility graph path planning.

Voronoi diagram. Contrasting with the visibility graph approach, a Voronoi diagram is a complete road map method that tends to *maximize* the distance between the robot and obstacles in the map. For each point in free space, its distance to the nearest obstacle is computed. If you plot that distance as the height coming out of the page, it increases as you move away from an obstacle (see figure 6.3). At points that are equidistant from two or more obstacles, such a distance plot has sharp ridges. The Voronoi diagram consists of the edges formed by these sharp ridge points. When the configuration space obstacles are polygons, the Voronoi diagram consists of straight line and parabolic segments only. Algorithms that find paths on the Voronoi road map are complete, just as are visibility graph methods, because the existence of a path in the free space implies the existence of one on the Voronoi diagram as well (i.e., both methods guarantee completeness). However, the solution paths on the Voronoi diagram are usually far from optimal in the sense of total path length.

The Voronoi diagram has an important weakness in the case of limited range localization sensors. Since its edges maximize the distance to obstacles, any short-range sensor on the robot will be in danger of failing to sense its surroundings. If such short-range sensors are used for localization, then the chosen path will be quite poor from a localization point of view. On the other hand, the visibility graph method can be designed to keep the robot as close as desired to objects in the map.

There is, however, an important subtle advantage that the Voronoi diagram method has over most other graphs: *executability*. Given a particular planned path via Voronoi diagram planning, a robot with range sensors, such as a laser rangefinder or ultrasonics, can follow a Voronoi edge in the physical world using simple control rules that match those used to

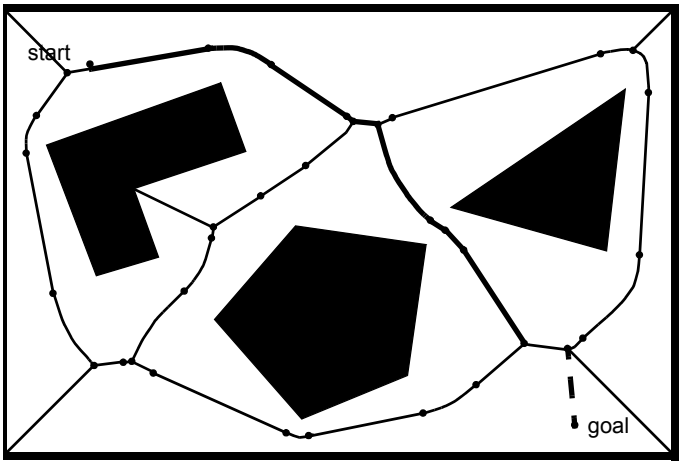

Figure 6.3
Voronoi diagram [32]. The Voronoi diagram consists of the lines constructed from all points that are equidistant from two or more obstacles. The initial q_{init} and goal q_{goal} configurations are mapped into the Voronoi diagram to q'_{init} and q'_{goal}, each by drawing the line along which its distance to the boundary of the obstacles increases the fastest. The points on the Voronoi diagram represent transitions from straight line segments (minimum distance between two lines) to parabolic segments (minimum distance between a line and a point).

create the Voronoi diagram: the robot maximizes the readings of local minima in its sensor values. This control system will naturally keep the robot on Voronoi edges, so that Voronoi motion can mitigate encoder inaccuracy. This interesting physical property of the Voronoi diagram has been used to conduct automatic mapping of an environment by finding and moving on unknown Voronoi edges, then constructing a consistent Voronoi map of the environment [103].

Exact cell decomposition. Figure 6.4 depicts exact cell decomposition, whereby the boundary of cells is based on geometric criticality. The resulting cells are each either completely free or completely occupied, and therefore path planning in the network is complete, like the road-map–based methods seen earlier. The basic abstraction behind such a decomposition is that the particular position of the robot within each cell of free space does not matter; what matters is rather the robot's ability to traverse from each free cell to adjacent free cells.

The key disadvantage of exact cell decomposition is that the number of cells and, therefore, the overall computational planning efficiency depends on the density and complexity of objects in the environment, just as with road-map–based systems. The key advantage is

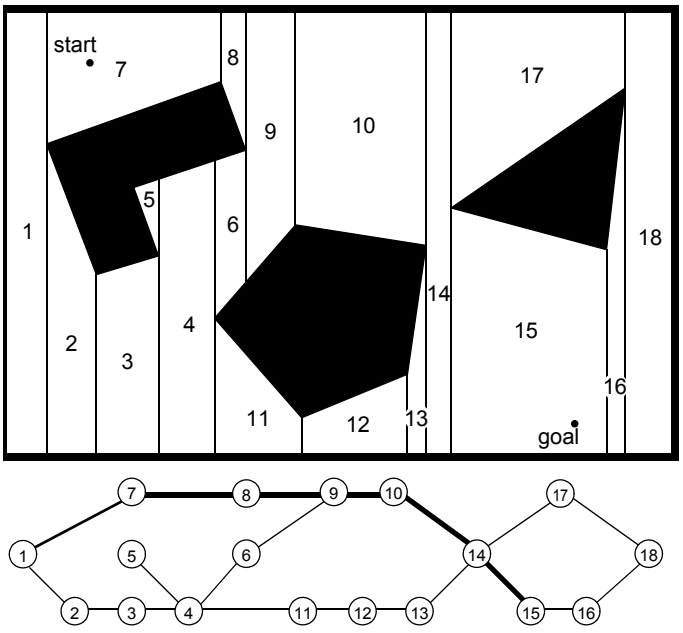

Figure 6.4
Example of exact cell decomposition. Cells are for example divided according to the horizontal coordinate of extremal obstacle points.

a result of this same correlation. In environments that are extremely sparse, the number of cells will be small, even if the geometric size of the environment is very large. Thus the representation will be efficient in the case of large, sparse environments. Practically speaking, due to complexities in implementation, the exact cell decomposition technique is used relatively rarely in mobile robot applications, although it remains a solid choice when a lossless representation is highly desirable—for instance, to preserve completeness fully.

Approximate cell decomposition. By contrast, approximate cell decomposition is one of the most popular graph construction techniques in mobile robotics. This is partly due to the popularity of grid environmental representations. These grid representations are themselves fixed grid-size decompositions and so they are identical to an approximate cell decomposition of the environment.

The most popular form of this, shown in figure 5.15, is the fixed-size cell decomposition. The cell size is not dependent on the particular objects in an environment, and so narrow passage ways can be lost due to the inexact nature of the tessellation. Practically

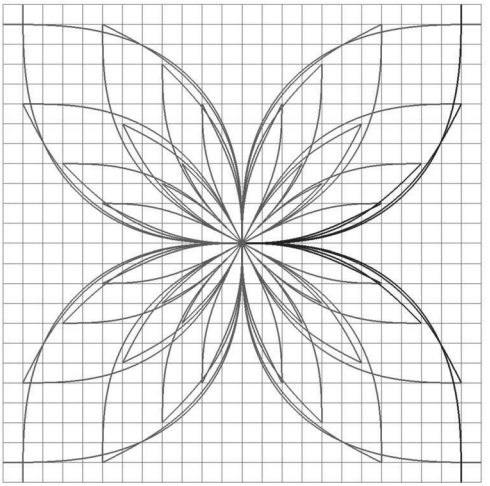

Figure 6.5 16-directional state lattice constructed for a planetary exploration rover. The state includes 2D position, heading, and curvature (x, y, θ, k). Note that straight segments are part of the lattice set but occluded by longer curved segments. The lattice is 2D-shift invariant and partially invariant to rotation. All successor edges of state (0, 0, 0, 0) are depicted in black. Image courtesy of M. Pivtoraiko [260].

speaking, this is rarely a problem owing to the very small cell size used (e.g., 5 cm on each side).

Figure 5.16 illustrates a variable-size approximate cell decomposition method. The free space is externally bounded by a rectangle and internally bounded by three polygons. The rectangle is recursively decomposed into smaller rectangles. Each decomposition generates four identical new rectangles. At each level of resolution only the cells whose interiors lie entirely in the free space are used to construct the connectivity graph. Path planning in such adaptive representations can proceed in a hierarchical fashion. Starting with a coarse resolution, the resolution is reduced until either the path planner identifies a solution or a limit resolution is attained (e.g, k • size of robot).

The great benefit of approximate cell decomposition is the low computational complexity induced to path planning.

Lattice graph. Lattice structures have only recently been adapted to graph search. They are formed by first constructing a base set of edges (such as the one depicted in figure 6.5) and then repeating it over the whole configuration space to form a graph. As such, the approximate cell decomposition technique could be interpreted as a simple lattice: the neighborhood structure of each cell forms a cross, which is then repeated by 2D shifts of

multiples of a single cell increment. The main benefit with respect to other graph construction methods lies in the design freedom in creating feasible edges, that is, edges that can be inherently executed by a robotic platform, however. To this end, Bicchi et al. [73] succeeded in applying an input discretization to a mathematical model of their robotic platform resulting in a configuration space lattice for certain simple kinematic vehicle models. Later, more broadly applicable methods have been devised in the configuration space directly: Pivtoraiko et al. [260] a priori fixed a problem specific configuration space discretization and dimensionality (e.g., in 2D position, orientation, curvature; figure 6.5). They then computed solutions to two point boundary problems between any two discrete states in the configuration space by also using a robot model. In a final step, the resulting large number of edges was pruned to a more manageable subset (the base lattice) by discarding edges which are similar to or can be decomposed into other edges already part of the subset.

Lattice graphs are typically precomputed for a given robotic platform and stored in memory. They thus belong to the class of approximate decomposition methods. Due to their inherent executability, edges along the solution path may be directly used as feed-forward commands to the controller.

Discussion. The fundamental cost of any fixed decomposition approach is memory. For a large environment, even when sparse, the grid must be represented in its entirety. Practically, because of the falling cost of RAM computer memory, this disadvantage has been mitigated in recent years.

In contrast to the exact decomposition methods, approximate approaches can sacrifice completeness but are mathematically less involved and thus easier to implement. In contrast to the fixed-size decompositions, variable-size decompositions will adapt to the complexity of the environment. Sparse environments will therefore contain appropriately fewer nodes and edges and consume dramatically less memory.

6.3.1.2 Deterministic graph search

Suppose now that our environment map has been converted into a connectivity graph using one of the graph generation methods presented earlier. Whatever map representation is chosen, the goal of path planning is to find the *best* path in the map's connectivity graph between the start and the goal, where *best* refers to the selected optimization criteria (e.g., the shortest path). In this section, we present several search algorithms that have become quite popular in mobile robotics. For an in-depth study on graph-search techniques we refer the reader to [44].

Discriminators. Due to the similarity between many graph search algorithms, we begin this section with an elaboration on their respective differences. To this end, it is beneficent to introduce the concepts of expected total cost $f(n)$, path cost $g(n)$, edge traversal cost $c(n, n')$, and heuristic cost $h(n)$, which are all functions of the node n (and an adjacent node

n'). In particular, we denote the accumulated cost from the start node to any given node n with $g(n)$. The cost from a node n to an adjacent node n' becomes $c(n, n')$, and the expected cost (heuristic cost) from a node n to the goal node is described with $h(n)$. The total expected cost from start to goal via state n can then be written as

$$f(n) = g(n) + \varepsilon \cdot h(n), \tag{6.1}$$

where ε is a parameter that assumes algorithm-dependent values.

In the special case that every individual edge in the graph assumes the same traversal cost (such as in an occupancy grid, introduced in section 5.5.2), optimal implementations may be developed in a simpler form and obtain faster execution speeds compared to the general instance. Examples of such algorithms include *depth-first* and *breadth-first* searches. On the other hand, Dijkstra's algorithm and variants allow for the computation of optimal paths in nonuniform cost maps as well. This comes at the cost of higher algorithmic complexity, however. In all of these implementations, $\varepsilon = 0$.

In the case of $\varepsilon \neq 0$, a heuristic function $h(n)$ is employed, which essentially incorporates additional information about the problem set and thus often allows for faster convergence of the search query. In this book, we restrict our attention to heuristics that are both consistent and underestimate the true cost. Most practical heuristics fulfill these requirements. For $\varepsilon = 1$, the optimal A* algorithm results, whereas for $\varepsilon > 1$ suboptimal or greedy A* variants are obtained.

Now that we have a general idea on the relation between some of the most popular graph search algorithms, we can proceed to introduce them in more detail.

Breadth-first search. This graph-search algorithm begins with the start node (denoted by A in figure 6.6) and explores all of its neighboring nodes. Then, for each of these nodes, it explores all their unexplored neighbors and so on. This process (that is, marking a node "active", exploring each of its neighbors and marking them "open", and finally marking the parent node "visited") is called node expansion. In breadth-first search, nodes are expanded in order of proximity to the start node with proximity defined as the shortest number of edge transitions. The algorithm proceeds until it reaches the goal node where it terminates. The computation of a solution is fast, since a reordering of nodes waiting for expansion is not necessary. They are already sorted in increasing order of proximity to the start node. Figure 6.6 illustrates the working principle of the breadth-first algorithm for a given graph.

It can be seen that the search always returns the path with the fewest number of edges between the start and goal node. If we assume that the cost of all individual edges in the graph is constant, then breadth-first search is also optimal in that it always returns the minimum-cost path. In this case, a node's reexpansion (as in the case of node G) can be easily circumvented by assigning a flag to a visited node. This addition does not affect solution

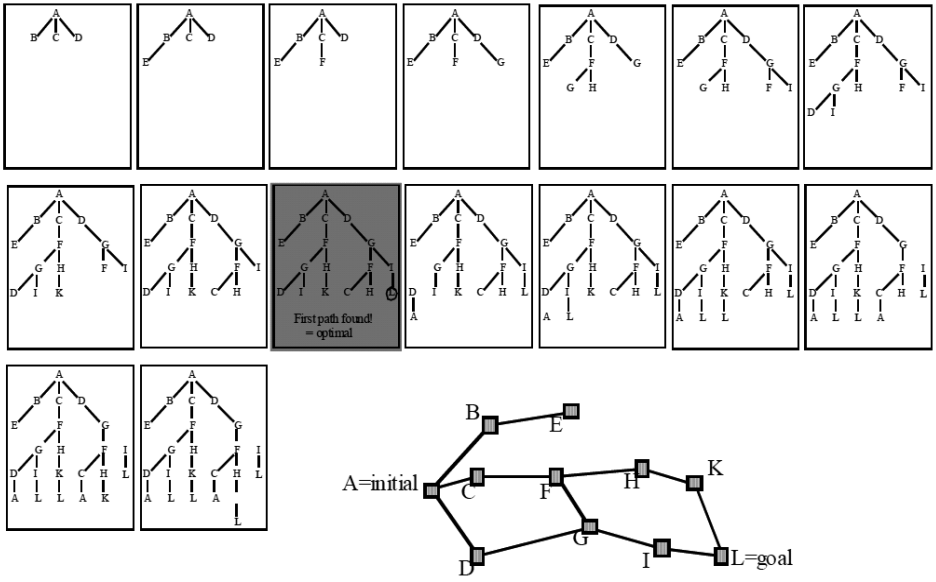

Figure 6.6 Working principle of breadth-first search.

optimality, since nodes are expanded in order of proximity to the start. However, if the graph has nonuniform costs associated with each edge, then breadth-first search is not guaranteed to be cost-optimal. Indeed, the path with the minimum number of edges does not necessarily coincide with the cheapest path, since there might be another path with more edges but lower total cost.

An example of breadth-first search algorithm in the context of robotics is the *wavefront expansion algorithm*, which is also known as *NF1* or *grassfire* [183]. This algorithm is an efficient and simple-to-implement technique for finding routes in fixed-size cell arrays. The algorithm employs wavefront expansion from the goal position outward, marking for each cell its L^1 (Manhattan) distance to the goal cell [154] (see figure 6.7). This process continues until the cell corresponding to the initial robot position is reached. At this point, the path planner can estimate the robot's distance to the goal position as well as recover a specific solution trajectory by simply linking together cells that are adjacent and always closer to the goal.

Given that the entire array can be in memory, each cell is only visited once when looking for the shortest discrete path from the initial position to the goal position. So, the search is linear in the number of cells only. Thus, complexity does not depend on the sparseness and

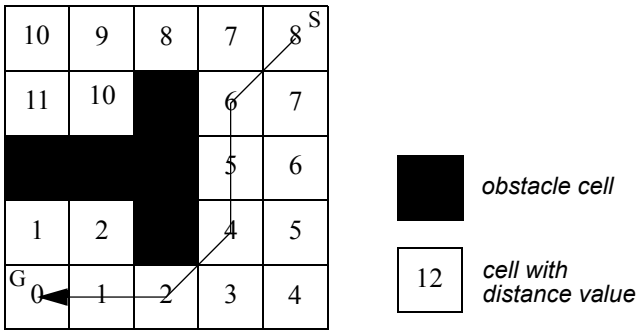

Figure 6.7
An example of the distance transform and the resulting path as it is generated by the NF1 function. S denotes the start, G the goal. The neighbors of each cell *i* are defined as the four adjacent cells that share an edge with *i* (4-neighborhood).

density of the environment, nor on the complexity of the objects' shapes in the environment.

Depth-first search. The working principle of depth-first search algorithm is shown in figure 6.8. In contrast to breadth-first search, depth-first search expands each node up to the deepest level of the graph (until the node has no more successors). As those nodes are expanded, their branch is removed from the graph and the search backtracks by expanding the next neighboring node of the start node until its deepest level and so on. An inconvenience of this algorithm is that it may revisit previously visited nodes or enter redundant paths. However, these situations may be easily avoided through an efficient implementation. A significant advantage of depth-first over breadth-first is space complexity. In fact, depth-first needs to store only a single path from the start node to the goal node along with all the remaining unexpanded neighboring nodes for each node on the path. Once each node has been expanded and all its children nodes have been explored, it can be removed from memory.

Dijkstra's algorithm. Named after its inventor, E.W. Dijkstra, this algorithm is similar to breadth-first search, except that edge costs may assume any positive value and the search still guarantees solution optimality [114]. This introduces some additional complexity into the algorithm for which we need to introduce the concept of the *heap*, a specialized tree-based data structure. Its elements (which are comprise to-be-expanded graph nodes) are ordered according to a key, which in our case amounts to the expected total path cost $f(n)$ at that given node n. Dijkstra's algorithm then expands nodes starting from the start similar

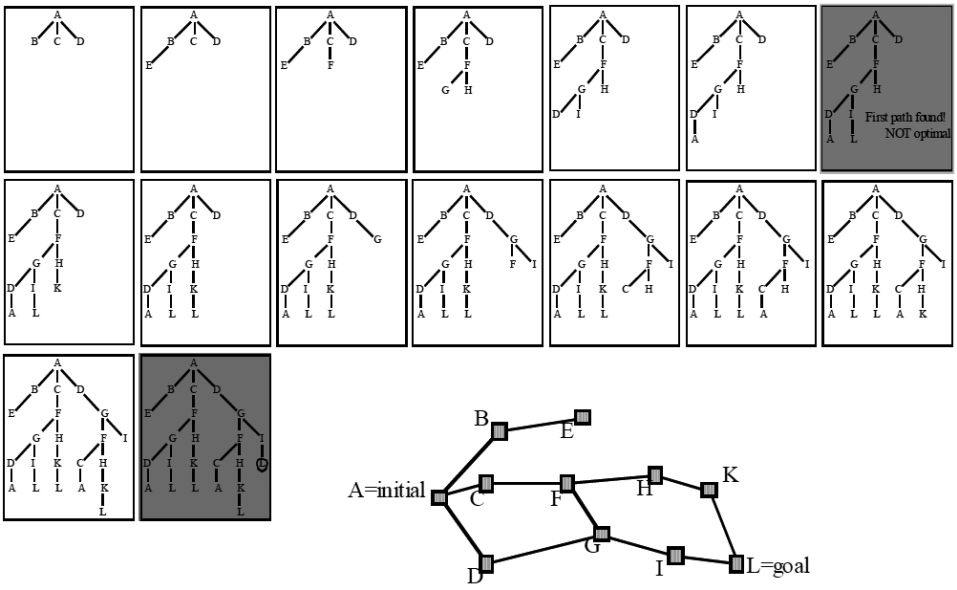

Figure 6.8 Working principle of depth-first search.

to breadth-first search, except that the neighbors of the expanded node are placed in the heap and reordered according to their $f(n)$ value, which corresponds to $g(n)$ since no heuristic is used. Subsequently, the cheapest state on the heap (the top element after reordering) is extracted and expanded. This process continues until the goal node is expanded, or no more nodes remain on the heap. A solution can then be backtracked from the goal to the start. Due to reorder operations on the heap, the time complexity rises from $O(n+m)$ in breadth-first search to $O(n\log(n)+m)$, with n the number of nodes, and m the number of edges.

In robotic applications, Dijkstra's search is typically computed from the robot's goal position. Consequently, not only the best path from the start node to the goal is computed, but also all lowest cost paths from any starting position in the graph to the goal node. The robot may thus localize and determine the best route toward the goal based on its current position. After moving some distance along this path, the process is repeated until the goal is reached, or the environment changes (which would require a recomputation of the solution). This technique allows the robot to reach the goal without replanning even in presence of localization and actuation noise.

A* algorithm. For consistent heuristics, the A* algorithm (pronounced "a star") [147] is similar to Dijkstra's algorithm. However, the inclusion of a heuristic function $h(n)$, which

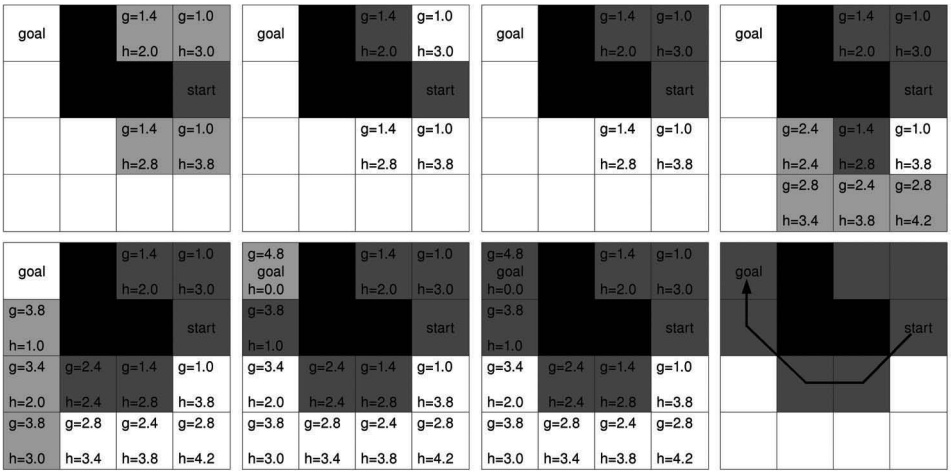

Figure 6.9 Working principle of the A* algorithm. Nodes are expanded in order of lowest $f(n) = g(n) + h(n)$ cost. $g(n)$ is indicated at the top left corner, $h(n)$ at the bottom right corner of each cell. The neighborhood of each cell is selected as the 8-neighborhood (all eight adjacent cells). Diagonal moves cost $\sqrt{2}$ times as much as horizontal and vertical moves. Obstacle cells are colored in black, expanded cells in dark gray, and cells put on the heap during this expansion step in light gray. Image courtesy of M. Rufli.

encodes additional knowledge about the graph, makes this algorithm especially efficient for single node to single node queries. In order to guarantee solution optimality, the heuristic is required to be an underestimating function of the cost to go. In robotics, A* is mainly employed on a grid, and the heuristic is then often chosen as the distance between any cell and the goal cell in absence of any obstacles. If such knowledge is available, it can be used to guide the search toward the goal node. Generally, this dramatically reduces the number of node expansions required to arrive at a solution compared to Dijkstra's algorithm.

A* search begins by expanding the start node and placing all of its neighbors on a heap. In contrast to Dijkstra's algorithm, the heap is ordered according to the smallest $f(n)$ value that includes the heuristic function $h(n)$. The lowest cost state is then extracted and expanded. This continues until the goal node is explored. The lowest cost solution can again be backtracked from the goal. For an example, see figure 6.9. The time complexity of A* largely depends on the chosen heuristic $h(n)$. On average, much better performance than with Dijkstra's algorithm can be expected, however.

Often it is not necessary to obtain an optimal solution, as long as there are guarantees on its suboptimality level. In such cases, a solution that costs at most ε times the (unknown) optimal solution may be obtained by setting $\varepsilon > 1$. The solution may then be improved as

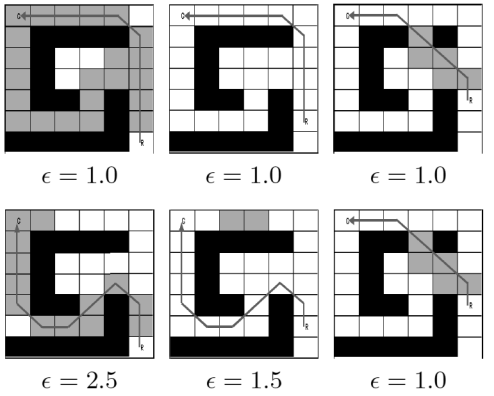

Figure 6.10 comparison of the number of expanded cells for D* (top) and Anytime D* (bottom, starting with a sub-optimality value of 2.5) in a planning and re-planning scenario. Note that an opening in the top wall is detected in the third frame, after the robot has moved upward twice. Obstacle cells are colored in black, cells expanded during a given time-step in gray. Image courtesy of M. Rufli.

search time allows, by reusing parts of the previous queries. This procedure results in the Anytime Replanning A* algorithm [191]. If the heuristic is accurate, far fewer states can be expected to be expanded than for optimal A*.

D* algorithm. The D* algorithm [304, 170] represents an incremental replanning version of A*, where the term incremental refers to the algorithm's reuse of previous search effort in subsequent search iterations. Let us illustrate this with an example (see figure 6.10): our robot is initially provided with a crude map of the environment (i.e., obtained from an aerial image). In this map, the navigation module plans an initial path by employing A*. After executing this path for a while, the robot observes some changes in the environment with its onboard sensors. Subsequent to updating the map, a new solution path needs to be computed. This is where D* comes into play. Instead of generating a new solution from scratch (as A* would do), only states affected by the added (or removed) obstacle cells are recomputed. Because changes to the map are most often observed locally (due to proprioceptive sensors), the planning problem is usually reversed; node expansion begins from the robot goal state. In this way, large parts of the previous solution remain valid for the new computation. Compared to A*, search time may decrease by a factor of one to two orders of magnitude. For more detail and a description on computing affected states, consult [170].

Analogous to A*, the D* algorithm has also been extended to an anytime version, called Anytime D* [192].

6.3.1.3 Randomized graph search

When encountering complex high-dimensional path planning problems (such as in manipulation tasks on robotic arms, or molecule folding and docking queries for drug placement, and so on) it becomes infeasible to solve them exhaustively within reasonable time limits. Reverting to heuristic search methods is often not possible due to the lack of an appropriate heuristic function and a reduction of the problem dimensionality frequently fails due to velocity and acceleration constraints imposed on the model, which should not be violated for security reasons. In such situations, randomized search becomes useful, since it forgoes solution optimality for faster solution computation.

Rapidly Exploring Random Trees (RRTs). RRTs typically grow a graph online during the search process and thus a priori only require an obstacle map but no graph decomposition. The algorithm begins with an initial tree (which might be empty) and then successively adds nodes, connected via edges, until a termination condition is triggered. Specifically, during each step a random configuration q_{rand} in the free space is selected. The tree node that is closest to q_{rand}, denoted as q_{near}, is then computed. Starting from q_{near}, an edge (with fixed length) is grown toward q_{rand} using an appropriate robot motion model. The configuration q_{new} at the end of this edge is then added to the tree, if the connecting edge is collision-free [185]

Typical extensions to the algorithm aim at speeding-up solution computation: bidirectional versions grow partial trees from both the start and goal configuration. Besides parallelization capability, faster convergence in nonconvex environments can be expected [186]. Another often employed modification biases the process of selecting a random free space configuration q_{rand}. The goal node is then selected instead of q_{rand} with a fixed nonzero probability, thus guiding tree growth toward the goal state. This process is especially efficient in sparse environments, but it may lead to a slowdown in presence of concave obstacles [33].

Even though the RRT algorithm and its extensions lack solution optimality guarantees and deterministic completeness, it can be proven that they are probabilistically complete. This signifies that if a solution exists, the algorithm will eventually find it as the number of nodes added to the tree grows toward infinity (see figure 6.11).

6.3.2 Potential field path planning

Potential field path planning creates a field, or gradient, across the robot's map that directs the robot to the goal position from multiple prior positions (see [32]). This approach was originally invented for robot manipulator path planning and is used often and under many variants in the mobile robotics community. The potential field method treats the robot as a point under the influence of an artificial potential field $U(q)$. The robot moves by follow-

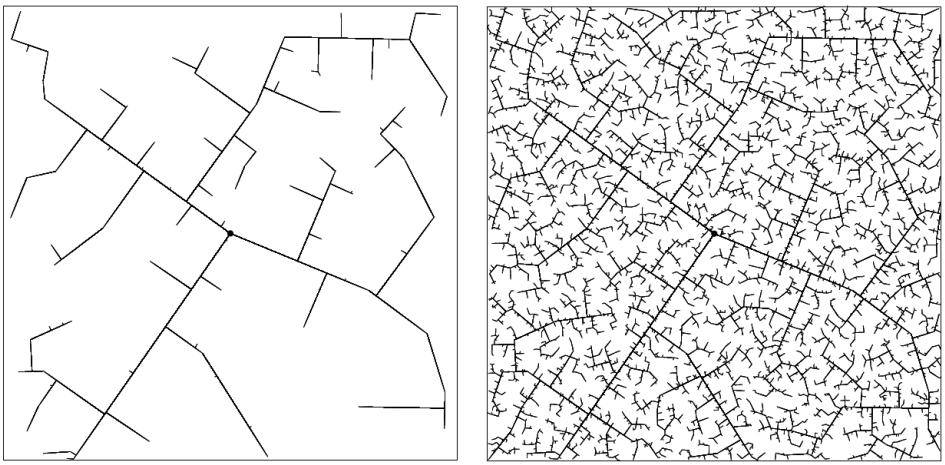

Figure 6.11 The evolution of a RRT. Image courtesy of S.M. LaValle [33].

ing the field, just as a ball would roll downhill. The goal (a minimum in this space) acts as an attractive force on the robot, and the obstacles act as peaks, or repulsive forces. The superposition of all forces is applied to the robot, which, in most cases, is assumed to be a point in the configuration space (see figure 6.12). Such an artificial potential field smoothly guides the robot toward the goal while simultaneously avoiding known obstacles.

It is important to note, though, that this is more than just path planning. The resulting field is also a control law for the robot. Assuming the robot can localize its position with respect to the map and the potential field, it can always determine its next required action based on the field.

The basic idea behind all potential field approaches is that the robot is attracted toward the goal, while being repulsed by the obstacles that are known in advance. If new obstacles appear during robot motion, one could update the potential field in order to integrate this new information. In the simplest case, we assume that the robot is a point; thus the robot's orientation θ is neglected, and the resulting potential field is only 2D (x, y). If we assume a differentiable potential field function $U(q)$, we can find the related artificial force $F(q)$ acting at the position $q = (x, y)$.

$$F(q) = -\nabla U(q), \tag{6.2}$$

where $\nabla U(q)$ denotes the gradient vector of U at position q.

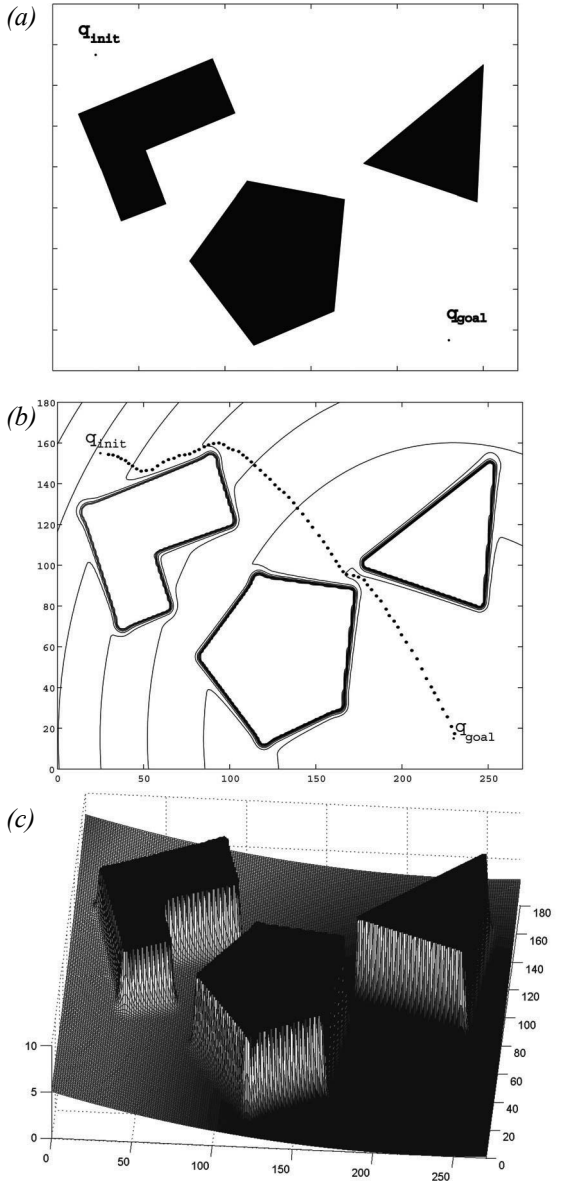

Figure 6.12
Typical potential field generated by the attracting goal and two obstacles (see [32]). (a) Configuration of the obstacles, start (top left) and goal (bottom right). (b) Equipotential plot and path generated by the field. (c) Resulting potential field generated by the goal attractor and obstacles.

$$\nabla U = \begin{bmatrix} \frac{\partial U}{\partial x} \\ \frac{\partial U}{\partial y} \end{bmatrix}. \tag{6.3}$$

The potential field acting on the robot is then computed as the sum of the attractive field of the goal and the repulsive fields of the obstacles:

$$U(q) = U_{att}(q) + U_{rep}(q). \tag{6.4}$$

Similarly, the forces can also be separated in a attracting and repulsing part:

$$\begin{aligned} F(q) &= F_{att}(q) - F_{rep}(q) \\ &= -\nabla U_{att}(q) - \nabla U_{rep}(q). \end{aligned} \tag{6.5}$$

Attractive potential. An attractive potential can, for example, be defined as a parabolic function.

$$U_{att}(q) = \frac{1}{2} k_{att} \cdot \rho_{goal}^2(q), \tag{6.6}$$

where k_{att} is a positive scaling factor and $\rho_{goal}(q)$ denotes the Euclidean distance $\|q - q_{goal}\|$. This attractive potential is differentiable, leading to the attractive force F_{att}

$$F_{att}(q) = -\nabla U_{att}(q) \tag{6.7}$$

$$= -k_{att} \cdot \rho_{goal}(q) \nabla \rho_{goal}(q) \tag{6.8}$$

$$= -k_{att} \cdot (q - q_{goal}) \tag{6.9}$$

that converges linearly toward 0 as the robot reaches the goal.

Repulsive potential. The idea behind the repulsive potential is to generate a force away from all known obstacles. This repulsive potential should be very strong when the robot is close to the object, but it should not influence its movement when the robot is far from the object. One example of such a repulsive field is

$$U_{rep}(q) = \begin{cases} \frac{1}{2}k_{rep}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho(q) \le \rho_0 \\ 0 & \text{if } \rho(q) \ge \rho_0 \,, \end{cases} \tag{6.10}$$

where k_{rep} is again a scaling factor, $\rho(q)$ is the minimal distance from q to the object and ρ_0 the distance of influence of the object. The repulsive potential function U_{rep} is positive or zero and tends to infinity as q gets closer to the object.

If the object boundary is convex and piecewise differentiable, $\rho(q)$ is differentiable everywhere in the free configuration space. This leads to the repulsive force F_{rep}:

$$F_{rep}(q) = -\nabla U_{rep}(q) \tag{6.11}$$

$$= \begin{cases} k_{rep}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)\frac{1}{\rho^2(q)}\frac{q - q_{obstacle}}{\rho(q)} & \text{if } \rho(q) \le \rho_0 \\ 0 & \text{if } \rho(q) \ge \rho_0 \,. \end{cases}$$

The resulting force $F(q) = F_{att}(q) + F_{rep}(q)$ acting on a point robot exposed to the attractive and repulsive forces moves the robot away from the obstacles and toward the goal (see figure 6.12). Under ideal conditions, by setting the robot's velocity vector proportional to the field force vector, the robot can be smoothly guided toward the goal, similar to a ball rolling around obstacles and down a hill.

However, there are some limitations with this approach. One is local minima that appear dependent on the obstacle shape and size. Another problem might appear if the objects are concave. This might lead to a situation for which several minimal distances $\rho(q)$ exist, resulting in oscillation between the two closest points to the object, which could obviously sacrifice completeness. For more detailed analyses of potential field characteristics, refer to [32].

The extended potential field method. Khatib and Chatila proposed the extended potential field approach [164]. Like all potential field methods, this approach makes use of attractive and repulsive forces that originate from an artificial potential field. However, two additions to the basic potential field are made: the *rotation potential field* and the *task potential field*.

The rotation potential field assumes that the repulsive force is a function of the distance from the obstacle and the orientation of the robot relative to the obstacle. This is done using

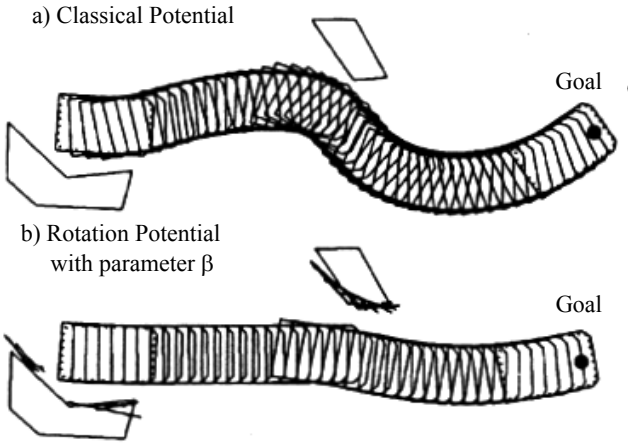

Figure 6.13
Comparison between a classical potential field and an extended potential field. Image courtesy of Raja Chatila [164].

a gain factor that reduces the repulsive force when an obstacle is parallel to the robot's direction of travel, since such an object does not pose an immediate threat to the robot's trajectory. The result is enhanced wall following, which was problematic for earlier implementations of potential fields methods.

The task potential field considers the present robot velocity, and from that it filters out those obstacles that should not affect the near-term potential based on robot velocity. Again a scaling is made, this time of all obstacle potentials when there are no obstacles in a sector named Z in front of the robot. The sector Z is defined as the space that the robot will sweep during its next movement. The result can be smoother trajectories through space. An example comparing a classical potential field and an extended potential field is depicted in figure 6.13.

Other extensions. A great variety of improvements to the artificial potential field method have been proposed and implemented since the development of Khatib's original approach in 1986 [163]. The most promising of these methods seems to be related to the harmonic potential field which is a solution to the Laplace equation [126, 230]

$$\nabla^2 U(q) \equiv 0\,,\ q \in \Omega\,, \tag{6.12}$$

where U again denotes the potential field as a function of the robot configuration q and Ω represents the workspace the robot operates in.

The main benefit of the harmonic potential field method with respect to earlier implementations is the complete absence of local minima inside the workspace. A unique solution may be generated through equation (6.12) and the specification of boundary conditions at start and goal locations and along the obstacle and workspace borders. In particular, the start location is pulled up to a high potential, whereas the goal position is pulled down to ground. For obstacle and workspace borders, we distinguish between two types of boundary conditions each resulting in a characteristic potential field. The Dirichlet condition requires that the potential is a known function along object boundaries (denoted with Γ)

$$U(q) = f(q), \; q \in \Gamma. \tag{6.13}$$

For $f(q) = const$, obstacle boundaries become equipotential lines. The robot then follows a path perpendicular to objects in their close vicinity. Excessively long but safe paths tend to emerge.

On the other hand, the von Neumann boundary condition requires

$$\frac{\partial U(q)}{\partial q} = g(q), \; q \in \Gamma. \tag{6.14}$$

where n is the normal vector to the obstacle boundary Γ. For $g(q) = 0$, robot motion parallel to object boundaries emerges. For all but the most elementary of obstacle geometries the Laplace equation needs to be solved numerically through a discretization of the workspace into cells. An iterative update rule (e.g the Gauss-Seidel method [155]) can then be applied until convergence. In several extensions, regional and directional constraints have been added to the harmonic potential field method to account for uneven terrain, nonholonomic and kinematic vehicle constraints [195], one-way roads [206], and external forces acting on the robot [207].

Potential fields are extremely easy to implement, much like the breadth-first search described on page 380. Thus, it has become a common tool in mobile robot applications in spite of its theoretical limitations.

This completes our brief summary of the path-planning techniques that are most popular in mobile robotics. Of course, as the complexity of a robot increases (e.g., large degree of freedom nonholonomics) and, particularly, as environment dynamics becomes more significant, then the path-planning techniques described earlier become inadequate for grappling with the full scope of the problem. However, for robots moving in largely flat terrain, the mobility decision-making techniques roboticists use often fall under one of the preceding categories.

But a path planner can take into consideration only the environmental obstacles that are known to the robot in *advance*. During path execution the robot's actual sensor values may disagree with expected values due to map inaccuracy or a dynamic environment. Therefore, it is critical that the robot modify its path in real time based on actual sensor values. This is the competence of *obstacle avoidance*, which we discuss next.

6.4 Obstacle avoidance

Local obstacle avoidance focuses on changing the robot's trajectory as informed by its sensors during robot motion. The resulting robot motion is both a function of the robot's current or recent sensor readings *and* its goal position and relative location to the goal position. The obstacle avoidance algorithms presented here depend to varying degrees on the existence of a global map and on the robot's precise knowledge of its location relative to the map. Despite their differences, all of the algorithms can be termed obstacle avoidance algorithms because the robot's local sensor readings play an important role in the robot's future trajectory. We first present the simplest obstacle avoidance systems that are used successfully in mobile robotics. The Bug algorithm represents such a technique in that only the most recent robot sensor values are used, and the robot needs, in addition to current sensor values, only approximate information regarding the direction of the goal. More sophisticated algorithms are presented afterward, taking into account recent sensor history, robot kinematics, and even dynamics.

6.4.1 Bug algorithm

The Bug algorithm [198, 199] is perhaps the simplest obstacle-avoidance algorithm one could imagine. The basic idea is to follow the contour of each obstacle in the robot's way and thus circumnavigate it.

With Bug1, the robot fully circles the object first, then departs from the point with the shortest distance toward the goal (figure 6.14). This approach is, of course, very inefficient but it guarantees that the robot will reach any reachable goal.

With Bug2 the robot begins to follow the object's contour, but departs immediately when it is able to move directly toward the goal. In general this improved Bug algorithm will have significantly shorter total robot travel, as shown in figure 6.15. However, one can still construct situations in which Bug2 is arbitrarily inefficient (i.e., nonoptimal).

A number of variations and extensions of the Bug algorithm exist. We mention one more, the Tangent Bug [161], which adds range sensing and a local environmental representation termed the local tangent graph (LTG). Not only can the robot move more efficiently toward the goal using the LTG, but it can also go along shortcuts when contouring obstacles and switch back to goal seeking earlier. In many simple environments, Tangent Bug approaches globally optimal paths.

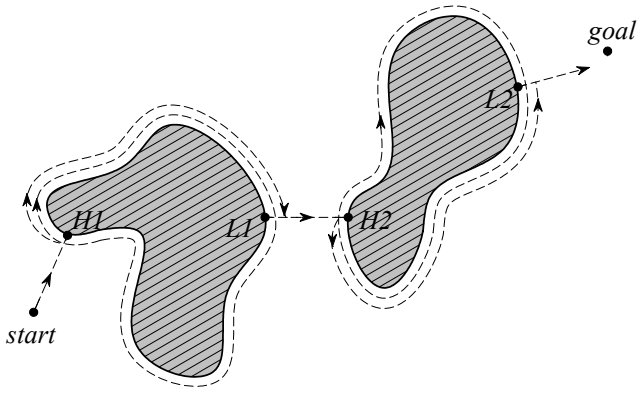

Figure 6.14
Bug1 algorithm with H1, H2, hit points, and L1, L2, leave points [199].

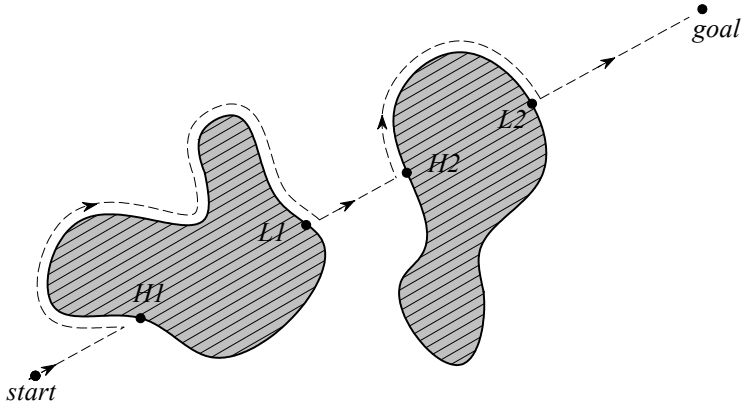

Figure 6.15
Bug2 algorithm with H1, H2, hit points, and L1, L2, leave points [199].

Practical application: example of Bug2. Because of the popularity and simplicity of Bug2, we present a specific example of obstacle avoidance using a variation of this technique. Consider the path taken by the robot in figure 6.15. One can characterize the robot's motion in terms of two states, one that involves moving toward the goal and a second that involves moving around the contour of an obstacle. We will call the former state `GOAL-`

SEEK and the latter WALLFOLLOW. If we can describe the motion of the robot as a function of its sensor values and the relative direction to the goal for each of these two states, and if we can describe when the robot should switch between them, then we will have a practical implementation of Bug2. The following pseudocode provides the highest-level control code for such a decomposition:

```
public void bug2(position goalPos){
 boolean atGoal = false;

 while( ! atGoal){
   position robotPos = robot.GetPos(&sonars);
   distance goalDist = getDistance(robotPos, goalPos);
  angle goalAngle = Math.atan2(goalPos, robotPos)-robot.GetAngle();
   velocity forwardVel, rotationVel;

   if(goalDist < atGoalThreshold){
     System.out.println("At Goal!");
     forwardVel = 0;
     rotationVel = 0;
     robot.SetState(DONE);
     atGoal = true;
   }
   else{
     forwardVel = ComputeTranslation(&sonars);
     if(robot.GetState() == GOALSEEK){
        rotationVel = ComputeGoalSeekRot(goalAngle);
        if(ObstaclesInWay(goalAngle, &sonars))
          robot.SetState(WALLFOLLOW);
     }
     if(robot.GetState() == WALLFOLLOW){
        rotationVel = ComputeRWFRot(&sonars);
        if( ! ObstaclesInWay(goalAngle, &sonars))
          robot.SetState(GOALSEEK);
     }
   }
   robot.SetVelocity(forwardVel, rotationVel);
 }
}
```

In the ideal case, when encountering an obstacle one would choose between left wall following and right wall following depending on which direction is more promising. In this simple example we have only right wall following, a simplification for didactic purposes that ought not find its way into a real mobile robot program.

Now we consider specifying each remaining function in detail. Consider for our purposes a robot with a ring of sonars placed radially around the robot. This imagined robot will be differential-drive, so that the sonar ring has a clear "front" (aligned with the forward direction of the robot). Furthermore, the robot accepts motion commands of the form shown above, with a rotational velocity parameter and a translational velocity parameter. Mapping these two parameters to individual wheel speeds for each of the two differential-drive chassis' drive wheels is a simple matter.

There is one condition we must define in terms of the robot's sonar readings, `ObstaclesInWay()`. We define this function to be true whenever any sonar range reading in the direction of the goal (within 45 degrees of the goal direction) is short:

```
private boolean ObstaclesInWay(angle goalAngle, sensorvals sonars) {
    int minSonarValue;
    minSonarValue=MinRange(sonars, goalAngle
                           -(pi/4),goalAngle+(pi/4));
    return (minSonarValue < 200);
} // end ObstaclesInWay() //
```

Note that the function `ComputeTranslation()` computes translational speed whether the robot is wall-following or heading toward the goal. In this simplified example, we define translation speed as being proportional to the largest range readings in the robot's approximate forward direction:

```
private int ComputeTranslation(sensorvals sonars) {
    int minSonarFront;
    minSonarFront = MinRange(sonars, -pi/4.0, pi/4.0);
    if (minSonarFront < 200) return 0;
    else return (Math.min(500, minSonarFront - 200));
} // end ComputeTranslation() //
```

There is a marked similarity between this approach and the potential field approach described in section 6.3.2. Indeed, some mobile robots implement obstacle avoidance by treating the current range readings of the robot as force vectors, simply carrying out vector addition to determine the direction of travel and speed. Alternatively, many will consider short-range readings to be repulsive forces, again engaging in vector addition to determine an overall motion command for the robot.

When faced with range sensor data, a popular way of determining rotation direction and speed is to simply subtract left and right range readings of the robot. The larger the difference, the faster the robot will turn in the direction of the longer range readings. The following two rotation functions could be used for our Bug2 implementation:

```
private int ComputeGoalSeekRot(angle goalAngle) {
    if (Math.abs(goalAngle) < pi/10) return 0;
    else return (goalAngle * 100);
} // end ComputeGoalSeekRot() //

private int ComputeRWFRot(sensorvals sonars) {
    int minLeft, minRight, desiredTurn;
    minRight = MinRange(sonars, -pi/2, 0);
    minLeft = MinRange(sonars, 0, pi/2);
    if (Math.max(minRight,minLeft) < 200) return (400);
                                          // hard left turn
    else {
        desiredTurn = (400 - minRight) * 2;
        desiredTurn = Math.inttorange(-400, desiredTurn, 400);
        return desiredTurn;
    } // end else
} // end ComputeRWFRot() //
```

Note that the rotation function for the case of right wall following combines a general avoidance of obstacles with a bias to turn right when there is open space on the right, thereby staying close to the obstacle's contour. This solution is certainly not the best solution for implementation of Bug2. For example, the wall follower could do a far better job by mapping the contour locally and using a PID control loop to achieve and maintain a specific distance from the contour during the right wall following action.

Although such simple obstacle avoidance algorithms are often used in simple mobile robots, they have numerous shortcomings. For example, the Bug2 approach does not take into account robot kinematics, which can be especially important with nonholonomic robots. Furthermore, since only the most recent sensor values are used, sensor noise can have a serious impact on real-world performance. The following obstacle avoidance techniques are designed to overcome one or more of these limitations.

6.4.2 Vector field histogram

Borenstein, together with Koren, developed the vector field histogram (VFH) [77]. Their previous work, which was concentrated on potential fields [176], was abandoned due to the method's instability and inability to pass through narrow passages. Later, Borenstein, together with Ulrich, extended the VFH algorithm to yield VFH+ [323] and VFH*[322].

One of the central criticisms of Bug-type algorithms is that the robot's behavior at each instant is generally a function of only its most recent sensor readings. This can lead to undesirable and yet preventable problems in cases where the robot's instantaneous sensor readings do not provide enough information for robust obstacle avoidance. The VFH techniques overcome this limitation by creating a local map of the environment around the robot. This local map is a small occupancy grid, as described in section 5.7 populated only by relatively

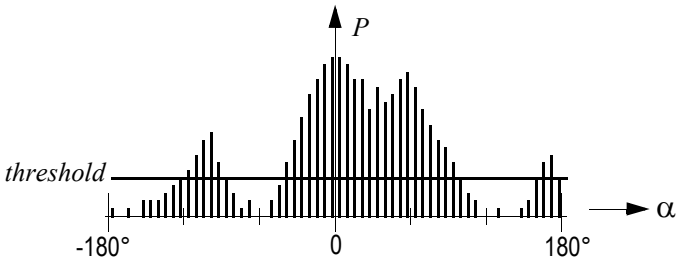

Figure 6.16
Polar histogram [177].

recent sensor range readings. For obstacle avoidance, VFH generates a polar histogram as shown in figure 6.16. The x-axis represents the angle α at which the obstacle was found, and the y-axis represents the probability P that there really is an obstacle in that direction based on the occupancy grid's cell values.

From this histogram a steering direction is calculated. First, all openings large enough for the vehicle to pass through are identified. Then a cost function is applied to every such candidate opening. The passage with the lowest cost is chosen. The cost function G has three terms:

$$G = a \cdot \text{target\_direction} + b \cdot \text{wheel\_orientation} + c \cdot \text{previous\_direction} \tag{6.15}$$

target_direction = alignment of the robot path with the goal;

wheel_orientation = difference between the new direction and the current wheel orientation;

previous_direction = difference between the previously selected direction and the new direction.

The terms are calculated such that a large deviation from the goal direction leads to a big cost in the term "target direction." The parameters a, b, c in the cost function G tune the behavior of the robot. For instance, a strong goal bias would be expressed with a large value for a. For a complete definition of the cost function, refer to [176].

In the VFH+ improvement, one of the reduction stages takes into account a simplified model of the moving robot's possible trajectories based on its kinematic limitations (e.g., turning radius for an Ackerman vehicle). The robot is modeled to move in arcs or straight lines. An obstacle thus blocks all of the robot's allowable trajectories that pass through the obstacle (figure 6.17a). This results in a masked polar histogram where obstacles are

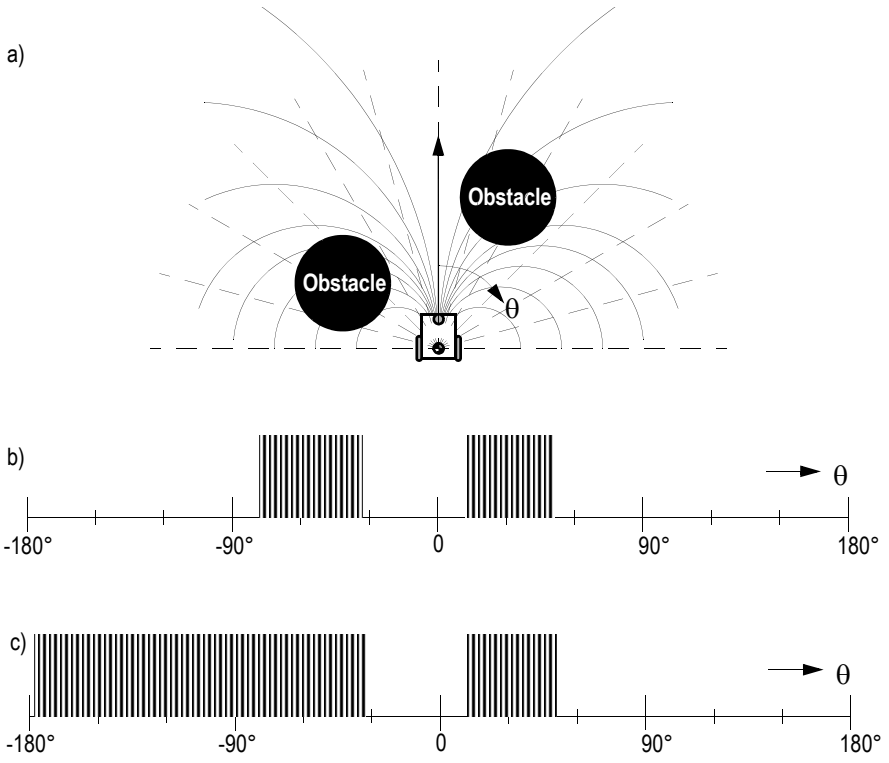

Figure 6.17
Example of blocked directions and resulting polar histograms [54]. (a) Robot and blocking obstacles. (b) Polar histogram. (b) Masked polar histogram.

enlarged so that all kinematically blocked trajectories are properly taken into account (figure 6.17c).

6.4.3 The bubble band technique

This idea is an extension for nonholonomic vehicles of the elastic band concept suggested by Khatib and Quinlan [166]. The original elastic band concept applied only to holonomic vehicles and so we focus only on the bubble band extension made by Khatib, Jaouni, Chatila, and Laumod [165].

A *bubble* is defined as the maximum local subset of the free space around a given configuration of the robot that which can be traveled in any direction without collision. The bubble is generated using a simplified model of the robot in conjunction with range information available in the robot's map. Even with a simplified model of the robot's geometry, it is possible to take into account the actual shape of the robot when calculating the bubble's

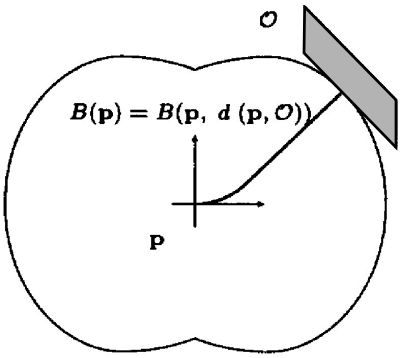

Figure 6.18
Shape of the bubbles around the vehicle. Courtesy of Raja Chatila [165].

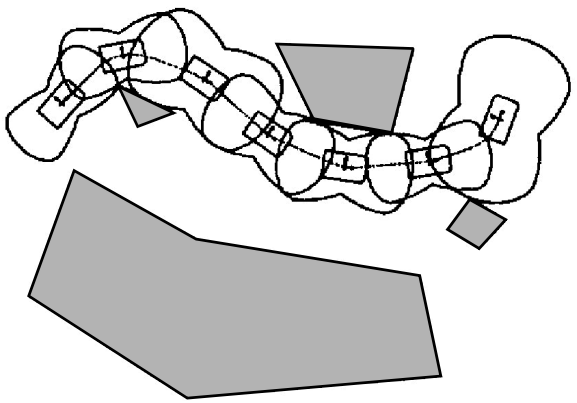

Figure 6.19
A typical bubble band. Courtesy of Raja Chatila [165].

size (figure 6.18). Given such bubbles, a band or string of bubbles can be used along the trajectory from the robot's initial position to its goal position to show the robot's expected free space throughout its path (see figure 6.19).

Clearly, computing the bubble band requires a global map and a global path planner. Once the path planner's initial trajectory has been computed and the bubble band is calculated, then modification of the planned trajectory ensues. The bubble band takes into account forces from modeled objects and internal forces. These internal forces try to minimize the "slack" (energy) between adjacent bubbles. This process, plus a final smoothing

operation, makes the trajectory smooth in the sense that the robot's free space will change as smoothly as possible during path execution.

Of course, so far this is more akin to path optimization than obstacle avoidance. The obstacle avoidance aspect of the bubble band strategy comes into play during robot motion. As the robot encounters unforeseen sensor values, the bubble band model is used to deflect the robot from its originally intended path in a way that minimizes bubble band *tension*.

An advantage of the bubble band technique is that one can account for the actual dimensions of the robot. However, the method is most applicable only when the environment configuration is well known ahead of time, just as with offline path-planning techniques.

6.4.4 Curvature velocity techniques

The basic curvature velocity approach. The curvature velocity approach (CVM) from Simmons [291] enables the actual kinematic constraints and even some dynamic constraints of the robot to be taken into account during obstacle avoidance, which is an advantage over more primitive techniques. CVM begins by adding physical constraints from the robot and the environment to a velocity space. The velocity space consists of rotational velocity ω and translational velocity v, thus assuming that the robot travels only along arcs of circles with curvature $c = \omega / v$.

Two types of constraints are identified: those derived from the robot's limitations in acceleration and speed, typically $-v_{max} < v < v_{max}$, $-\omega_{max} < \omega < \omega_{max}$; and, second, the constraints from obstacles blocking certain v and ω values due to their positions. The obstacles begin as objects in a Cartesian grid but are then transformed to the velocity space by calculating the distance from the robot position to the obstacle following some constant curvature robot trajectory, as shown in figure 6.20. Only the curvatures that lie within c_{min} and c_{max} are considered, since that curvature space will contain all legal trajectories.

To achieve real-time performance, the obstacles are approximated by circular objects, and the contours of the objects are divided into few intervals. The distance from an endpoint of an interval to the robot is calculated and in between the endpoints the distance function is assumed to be constant.

The final decision of a new velocity (v and ω) is made by an objective function. This function is only evaluated on that part of the velocity space that fulfills the kinematic and dynamic constraints as well as the constraints due to obstacles. The use of a Cartesian grid for initial obstacle representation enables straightforward sensor fusion if, for instance, a robot is equipped with multiple types of ranging sensors.

CVM takes into consideration the dynamics of the vehicle in useful manner. However a limitation of the method is the circular simplification of obstacle shape. In some environments this is acceptable, while in other environments such a simplification can cause seri-

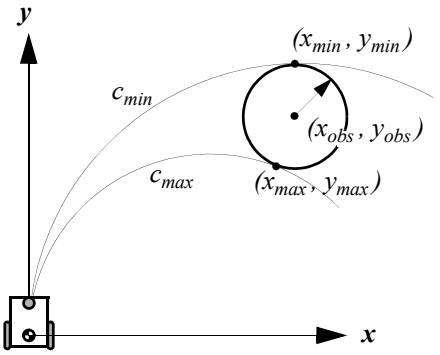

Figure 6.20
Tangent curvatures for an obstacle (from [291]).

ous problems. The CVM method can also suffer from local minima, since no *a priori* knowledge is used by the system.

The lane curvature method. Ko and Simmons presented an improvement of the CVM that they named the lane curvature method (LCM) [168], based on their experiences with the shortcomings of CVM, which had difficulty guiding the robot through intersections of corridors. The problems stemmed from the approximation that the robot moves only along fixed arcs, whereas in practice the robot can change direction many times before reaching an obstacle.

LCM calculates a set of desired lanes, trading off lane length and lane width to the closest obstacle. The lane with the best properties is chosen using an objective function. The local heading is chosen in such way that the robot will transition to the best lane if it is not in that lane already.

Experimental results have demonstrated better performance as compared to CVM. One caveat is that the parameters in the objective function must be chosen carefully to optimize system behavior.

6.4.5 Dynamic window approaches

Another technique for taking into account robot kinematics constraints is the dynamic window obstacle avoidance method. A simple but very effective dynamic model gives this approach its name. Two such approaches are represented in the literature. The dynamic window approach [130] of Fox, Burgard, and Thrun, and the global dynamic window approach [81] of Brock and Khatib.

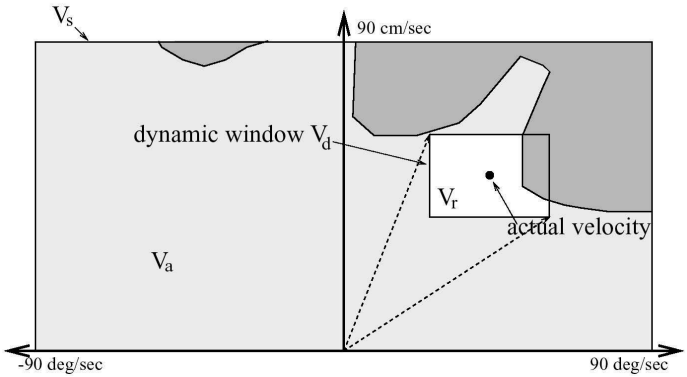

Figure 6.21
The dynamic window approach (courtesy of Dieter Fox [130]). The rectangular window shows the possible speeds (v, ω) and the overlap with obstacles in configuration space.

The local dynamic window approach. In the local dynamic window approach, the kinematics of the robot is taken into account by searching a well-chosen velocity space. The velocity space is all possible sets of tuples (v, ω) where v is the velocity and ω is the angular velocity. The approach assumes that robots move only in circular arcs representing each such tuple, at least during one timestamp.

Given the current robot speed, the algorithm first selects a *dynamic window* of all tuples (v, ω) that can be reached within the next sample period, taking into account the acceleration capabilities of the robot and the cycle time. The next step is to reduce the *dynamic window* by keeping only those tuples that ensure that the vehicle can come to a stop before hitting an obstacle. The remaining velocities are called admissible velocities. In figure 6.21, a typical dynamic window is represented. Note that the shape of the dynamic window is rectangular, which follows from the approximation that the dynamic capabilities for translation and rotation are independent.

A new motion direction is chosen by applying an objective function to all the admissible velocity tuples in the dynamic window. The objective function prefers fast forward motion, maintenance of large distances to obstacles and alignment to the goal heading. The objective function O has the form

$$O = a \cdot heading(v, \omega) + b \cdot velocity(v, \omega) + c \cdot dist(v, \omega) \tag{6.16}$$

heading = Measure of progress toward the goal location;

velocity = Forward velocity of the robot $\rightarrow$ encouraging fast movements;

dist = Distance to the closest obstacle in the trajectory.

The global dynamic window approach. The global dynamic window approach adds, as the name suggests, global thinking to the algorithm presented above. This is done by adding *NF1*, or grassfire, to the objective function *O* presented above (see section and figure 6.7). Recall that NF1 labels the cells in the occupancy grid with the total distance L to the goal. To make this faster, the global dynamic window approach calculates the NF1 only on a selected rectangular region that is directed from the robot toward the goal. The width of the region is enlarged and recalculated if the goal cannot be reached within the constraints of this chosen region.

This allows the global dynamic window approach to achieve some of the advantages of global path planning without complete a priori knowledge. The occupancy grid is updated from range measurements as the robot moves in the environment. The NF1 is calculated for every new updated version. If the NF1 cannot be calculated because the robot is surrounded by obstacles, the method degrades to the dynamic window approach. This keeps the robot moving so that a possible way out may be found and NF1 can resume.

The global dynamic window approach promises real-time, dynamic constraints, global thinking, and minimal free obstacle avoidance at high speed. An implementation has been demonstrated with an omnidirectional robot using a 450 MHz on-board PC. This system produced a cycle frequency of about 15 Hz when the occupancy grid was 30×30 m with a 5 cm resolution. Average robot speed in the tests was greater than 1 m/s.

6.4.6 The Schlegel approach to obstacle avoidance

Schlegel [280] presents an approach that considers the dynamics as well as the actual shape of the robot. The approach is adopted for raw laser data measurements and sensor fusion using a Cartesian grid to represent the obstacles in the environment. Real-time performance is achieved by use of precalculated lookup tables.

As with previous methods we have described, the basic assumption is that a robot moves in trajectories built up by circular arcs, defined as curvatures i_c. Given a certain curvature i_c Schlegel calculates the distance l_i to collision between a single obstacle point $[x, y]$ in the Cartesian grid and the robot, depicted in figure 6.22. Since the robot is allowed to be any shape, this calculation is time-consuming, and the result is therefore precalculated and stored in a lookup table.

For example, the search space window V_s is defined for a differential-drive robot to be all the possible speeds of the left and right wheels, v_r, v_l. The dynamic constraints of the robot are taken into account by refining V_s to only those values that are reachable within the next timestep, given the present robot motion. Finally, an objective function chooses the best speed and direction by trading off goal direction, speed, and distance until collision.

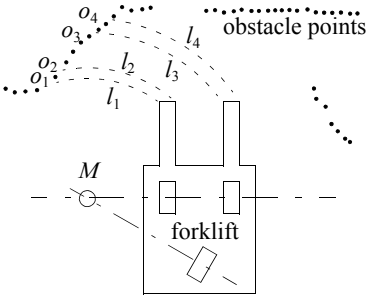

Figure 6.22
Distances l_i resulting from the curvature i_c, when the robot rotates around M (from [280]).

During testing Schlegel used a wavefront path planner. Two robot chassis were used, one with synchro-drive kinematics and one with tricycle kinematics. The tricycle-drive robot is of particular interest because it was a forklift with a complex shape that had a significant impact on obstacle avoidance. Thus the demonstration of reliable obstacle avoidance with the forklift is an impressive result. Of course, a disadvantage of this approach is the potential memory requirements for the lookup table. In their experiments, the authors used lookup tables of up to 2.5 Mb using a 6×6 m Cartesian grid with a resolution of 10 cm and 323 different curvatures.

6.4.7 Nearness diagram

Attempting to close a model fidelity gap in obstacle avoidance methods, the nearness diagram (ND) [222] can be considered to have some similarity to a VFH but solves several of its shortcomings, especially in very cluttered spaces. It was also used in [223] to take into account more precise geometric, kinematic, and dynamic constraints. This was achieved by breaking the problem down into generating the most promising direction of travel with the sole constraint a circular robot, then adapting this to the kinematic and dynamic constraints of the robot, followed by a correction for robot shape if is noncircular (only rectangular shapes were supported in the original publication). Global reasoning was added to the approach and termed the global nearness diagram (GND) in [225], somewhat similar to the GDWA extension to the DWA, but based on a workspace representation (instead of configuration space) and updating free space in addition to obstacle information.

6.4.8 Gradient method

Realizing that current computer technology allows fast recalculation of wavefront propagation techniques, the gradient method [171] formulates a grid global path planning that takes into account closeness to obstacles and allows generating continuous interpolations

of the gradient direction at any given point in the grid. The NF1 is a special case of the proposed algorithm, which calculates a navigation function at each timestep and uses the resulting gradient information to drive the robot toward the goal on a smooth path and not grazing obstacles unless necessary.

6.4.9 Adding dynamic constraints

Attempting to address the lack of dynamic models in most of the obstacle avoidance approaches discussed above, a new kind of space representation was proposed by Minguez, Montano, and Khatib in [224]. The ego-dynamic space is equally applicable to workspace and configuration space methods. It transforms obstacles into distances that depend on the braking constraints and sampling time of the underlying obstacle avoidance method. In combination with the proposed spatial window (PF) to represent acceleration capabilities, the approach was tested in conjunction with the ND and PF methods and gives satisfactory results for circular holonomic robots, with plans to extend it to nonholonomic, noncircular architectures.

6.4.10 Other approaches

The approaches described above are some of the most popularly referenced obstacle avoidance systems. There are, however, a great many additional obstacle avoidance techniques in the mobile robotics community. For example Tzafestas and Tzafestas [321] provide an overview of fuzzy and neurofuzzy approaches to obstacle avoidance. Inspired by nature, Chen and Quinn [98] present a biological approach in which they replicate the neural network of a cockroach. The network is then applied to a model of a four-wheeled vehicle.

The Liapunov functions form a well known theory that can be used to prove stability for nonlinear systems. In Vanualailai, Nakagiri, and Ha [326], the Liapunov functions are used to implement a control strategy for two-point masses moving in a known environment. All obstacles are defined as antitargets with an exact position and a circular shape. The antitargets are then used to build the control laws for the system.

6.4.11 Overview

Table 6.1 gives an overview on the presented obstacle-avoidance approaches.

Table 6.1
Overview of the most popular obstacle-avoidance algorithms

method																		
Vector Field Histogram (VFH)				Bug			model fidelity			view	other requisites			sensors	tested robots	performance		
VFH* [322]	VFH+ [176, 323]	VFH [77]	Tangent Bug [161]	Bug2 [198, 199]	Bug1 [198, 199]	shape	kinematics	dynamics		local map	global map	path planner			cycle time	architecture	remarks
circle	circle	simplistic	point	point	point				local				tactile				very inefficient, robust
basic	basic																
simplistic	simplistic																
essentially local	local	local	local	local	local												inefficient, robust
histogram grid	histogram grid	histogram grid	local tangent graph										tactile				
sonars	sonars	range	range														efficient in many cases, robust
nonholonomic (GuideCane)	nonholonomic (GuideCane)	synchro-drive (hexagonal)															
6 ... 242 ms	6 ms	27 ms															
66 MHz, 486 PC	66 MHz, 486 PC	20 MHz, 386 AT															
fewer local minima	local minima	local minima, oscillating trajectories															

Table 6.1
Overview of the most popular obstacle-avoidance algorithms

method		Dynamic window		Curvature velocity		Bubble band		model fidelity			view	other requisites			sensors	tested robots	performance		
								shape	kinematics	dynamics		local map	global map	path planner			cycle time	architecture	remarks
Global dynamic window [81]	Dynamic window approach [130]	Lane curvature method [168]	Curvature velocity method [291]	Bubble band [165]	Elastic band [166]														
circle	circle	circle	circle	C-space	C-space														
(holonomic)	exact	exact	exact	exact															
basic	basic	basic	basic																
global	local	local	local	local	global														
	obstacle line field	histogram grid	histogram grid																
C-space grid				polygonal	polygonal														
NF1				required	required														
180° FOV SCK laser scanner	24 sonars ring, 56 infrared ring, stereo camera	24 sonars ring, 30° FOV laser	24 sonars ring, 30° FOV laser																
holonomic (circular)	synchro-drive (circular)	synchro-drive (circular)	synchro-drive (circular)	various	various														
6.7 ms	250 ms	125 ms	125 ms																
450 MHz, PC	486 PC	200 MHz, Pentium	66 MHz, 486 PC																
turning into corridors	local minima	local minima	local minima, turning into corridors																

Table 6.1
Overview of the most popular obstacle-avoidance algorithms

method			model fidelity			view	other requisites			sensors	tested robots	performance		
			shape	kinematics	dynamics		local map	global map	path planner			cycle time	architecture	remarks
Other	Schlegel [280]	Nearness diagram [222, 223]	polygon	exact	basic	global		grid	wavefront	360° FOV laser scanner	synchrodrive (circular), tricycle (forklift)			allows shape change
			circle (but general formulation)	(holonomic)		local				180° FOV SCK laser scanner	holonomic (circular)			local minima
		Global nearness diagram [225]	circle (but general formulation)	(holonomic)		global	grid	NF1		180° FOV SCK laser scanner	holonomic (circular)			
	Gradient method [171]		circle	exact	basic	global		local perceptual space	fused	180° FOV distance sensor	nonholonomic (approx. circle)	100 ms (core algorithm: 10 ms)	266 MHz, Pentium	

6.5 Navigation Architectures

Given techniques for path planning, obstacle avoidance, localization, and perceptual interpretation, how do we combine all of these into one complete robot system for a real-world application? One way to proceed would be to custom-design an application-specific, monolithic software system that implements everything for a specific purpose. This may be efficient in the case of a trivial mobile robot application with few features and even fewer planned demonstrations. But for any sophisticated and long-term mobile robot system, the

issue of mobility architecture should be addressed in a principled manner. The study of *navigation architectures* is the study of principled designs for the software modules that constitute a mobile robot navigation system. Using a well-designed navigation architecture has a number of concrete advantages:

6.5.1 Modularity for code reuse and sharing

Basic software engineering principles embrace software modularity, and the same general motivations apply equally to mobile robot applications. But modularity is of even greater importance in mobile robotics because in the course of a single project the mobile robot hardware or its physical environmental characteristics can change dramatically, a challenge most traditional computers do not face. For example, one may introduce a Sick laser rangefinder to a robot that previously used only ultrasonic rangefinders. Or one may test an existing navigator robot in a new environment where there are obstacles that its sensors cannot detect, thereby demanding a new path-planning representation.

We would like to change part of the robot's competence without causing a string of side effects that force us to revisit the functioning of other robot competences. For instance we would like to retain the obstacle avoidance module intact, even as the particular ranging sensor suite changes. In a more extreme example, it would be ideal if the nonholonomic obstacle avoidance module could remain untouched even when the robot's kinematic structure changes from a tricycle chassis to a differential-drive chassis.

6.5.2 Control localization

Localization of robot control is an even more critical issue in mobile robot navigation. The basic reason is that a robot architecture includes multiple types of control functionality (e.g., obstacle avoidance, path planning, path execution, etc.). By localizing each functionality to a specific unit in the architecture, we enable individual testing as well as a principled strategy for control composition. For example, consider collision avoidance. For stability in the face of changing robot software, as well as for focused verification that the obstacle avoidance system is correctly implemented, it is valuable to localize all software related to the robot's obstacle avoidance process. At the other extreme, high-level planning and task decision making are required for robots to perform useful roles in their environment. It is also valuable to localize such high-level decision-making software, enabling it to be tested exhaustively in simulation and thus verified even without a direct connection to the physical robot. A final advantage of localization is associated with learning. Localization of control can enable a specific learning algorithm to be applied to just one aspect of a mobile robot's overall control system. Such targeted learning is likely to be the first strategy that yields successful integration of learning and traditional mobile robotics.

The advantages of localization and modularity provide a compelling case for the use of principled navigation architectures.

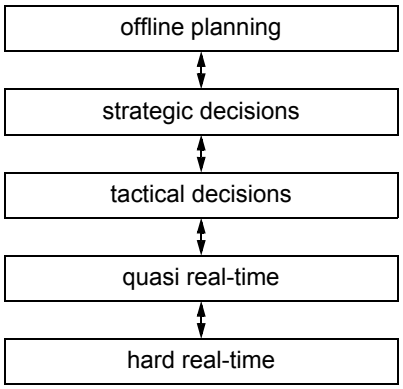

Figure 6.23
Generic temporal decomposition of a navigation architecture.

One way to characterize a particular architecture is by its decomposition of the robot's software. There are many favorite robot architectures, especially when one considers the relationship between artificial intelligence level decision making and lower-level robot control. For descriptions of such high-level architectures, refer to [2] and [39]. Here we concentrate on navigation competence. For this purpose, two decompositions are particularly relevant: temporal decomposition and control decomposition. In section 6.5.3 we define these two types of decomposition, then present an introduction to *behaviors*, which are a general tool for implementing control decomposition. Then, in section 6.5.4 we present three types of navigation architectures, describing for each architecture an implemented mobile robot case study.

6.5.3 Techniques for decomposition

Decompositions identify axes along which we can justify discrimination of robot software into distinct modules. Decompositions also serve as a way to classify various mobile robots into a more quantitative taxonomy. *Temporal decomposition* distinguishes between real-time and non real-time demands on mobile robot operation. *Control decomposition* identifies the way in which various control outputs within the mobile robot architecture combine to yield the mobile robot's physical actions. We will describe each type of decomposition in greater detail.

6.5.3.1 Temporal decomposition

A temporal decomposition of robot software distinguishes between processes that have varying real-time and non-real-time demands. Figure 6.23 depicts a generic temporal decomposition for navigation. In this figure, the most real-time processes are shown at the

bottom of the *stack*, with the highest category being occupied by processes with no real-time demands.

The lowest level in this example captures functionality that must proceed with a guaranteed fast cycle time, such as a 40 Hz bandwidth. In contrast, a quasi real-time layer may capture processes that require, for example, 0.1 second response time, with large allowable worst-case individual cycle times. A tactical layer can represent decision making that affects the robot's immediate actions and is therefore subject to some temporal constraints, while a strategic or offline layer represents decisions that affect the robot's behavior over the long term, with few temporal constraints on the module's response time.

Four important, interrelated trends correlate with temporal decomposition. These are not set in stone; there are exceptions. Nevertheless, these general properties of temporal decompositions are enlightening.

Sensor response time. A particular module's sensor response time can be defined as the amount of time between acquisition of a sensor event and a corresponding change in the output of the module. As one moves up the stack in figure 6.23, the sensor response time tends to increase. For the lowest-level modules, the sensor response time is often limited only by the raw processor and sensor speeds. At the highest-level modules, sensor response can be limited by slow and deliberate decision-making processes.

Temporal depth. Temporal depth is a useful concept applying to the temporal window that affects the module's output, both backward and forward in time. *Temporal horizon* describes the amount of look ahead used by the module during the process of choosing an output. *Temporal memory* describes the historical time span of sensor input that is used by the module to determine the next output. Lowest-level modules tend to have very little temporal depth in both directions, whereas the deliberative processes of highest-level modules make use of a large temporal memory and consider actions based on their long-term consequences, making note of large temporal horizons.

Spatial locality. Hand in hand with temporal span, the spatial impact of layers increases dramatically as one moves from low-level modules to high-level modules. Real-time modules tend to control wheel speed and orientation, controlling spatially localized behavior. High-level strategic decision making has little or no bearing on local position, but it informs global position far into the future.

Context specificity. A module makes decisions as a function not only of its immediate inputs but also as a function of the robot's context as captured by other variables, such as the robot's representation of the environment. Lowest-level modules tend to produce outputs directly as a result of immediate sensor inputs, using little context and therefore being

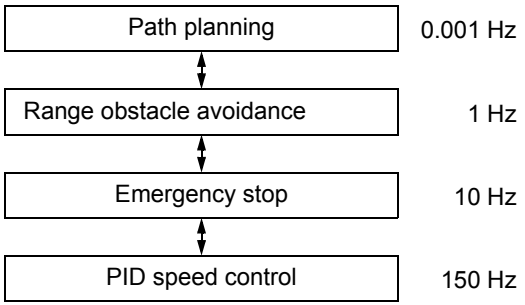

Figure 6.24
Sample four-level temporal decomposition of a simple navigating mobile robot. The column on the right indicates realistic bandwidth values for each module.

relatively context insensitive. Highest-level modules tend to exhibit very high context specificity. For strategic decision making, given the same sensor values, dramatically different outputs are nevertheless conceivable depending on other contextual parameters.

An example demonstrating these trends is depicted in figure 6.24, which shows a temporal decomposition of a simplistic navigation architecture into four modules. At the lowest level, the PID control loop provides feedback to control motor speeds. An emergency stop module uses short-range optical sensors and bumpers to cut current to the motors when it predicts an imminent collision. Knowledge of robot dynamics means that this module by nature has a greater temporal horizon than the PID module. The next module uses longer-range laser rangefinding sensor returns to identify obstacles well ahead of the robot and make minor course deviations. Finally, the path planner module takes the robot's initial and goal positions and produces an initial trajectory for execution, subject to change based on actual obstacles that the robot collects along the way.

Note that the cycle time, or bandwidth, of the modules changes by orders of magnitude between adjacent modules. Such dramatic differences are common in real navigation architectures, and so temporal decomposition tends to capture a significant axis of variation in a mobile robot's navigation architecture.

6.5.3.2 Control decomposition

Whereas temporal decomposition discriminates based on the time behavior of software modules, control decomposition identifies the way in which each module's output contributes to the overall robot control outputs. Presentation of control decomposition requires the evaluator to understand the basic principles of discrete systems representation and analysis. For a lucid introduction to the theory and formalism of discrete systems, see [25, 136].

Consider the robot algorithm and the physical robot instantiation (i.e., the robot form and its environment) to be members of an overall system whose connectivity we wish to

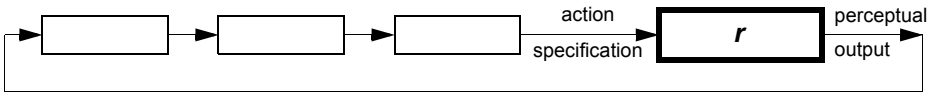

Figure 6.25
Example of a pure serial decomposition.

examine. This overall system S comprises a set M of modules, each module m connected to other modules via inputs and outputs. The system is *closed*, meaning that the input of every module m is the output of one or more modules in M. Each module has precisely one output and one or more inputs. The one output can be connected to any number of other modules inputs.

We further name a special module r in M to represent the physical robot and environment. Usually by r we represent the physical object on which the robot algorithm is intended to have impact, and from which the robot algorithm derives perceptual inputs. The module r contains one input and one output line. The input of r represents the complete action specification for the physical robot. The output of r represents the complete perceptual output to the robot. Of course the physical robot may have many possible degrees of freedom and, equivalently, many discrete sensors. But for this analysis we simply imagine the entire input/output vector, thus simplifying *r* to just one input and one output. For simplicity, we will refer to the input of r as O and to the robot's sensor readings I. From the point of view of the rest of the control system, the robot's sensor values I are inputs, and the robot's actions O are the outputs, explaining our choice of I and O.

Control decomposition discriminates between different types of control pathways through the portion of this system comprising the robot algorithm. At one extreme, depicted in figure 6.25 we can consider a perfectly linear, or sequential control pathway.

Such a serial system uses the internal state of all associated modules and the value of the robot's percept I in a sequential manner to compute the next robot action O. A pure serial architecture has advantages relating to predictability and verifiability. Since the state and outputs of each module depend entirely on the inputs it receives from the module upstream, the entire system, including the robot, is a single well-formed loop. Therefore, the overall behavior of the system can be evaluated using well-known discrete forward simulation methods.

Figure 6.26 depicts the extreme opposite of pure serial control, a fully parallel control architecture. Because we choose to define *r* as a module with precisely one input, this parallel system includes a special module n that provides a single output for the consumption of r. Intuitively, the fully parallel system distributes responsibility for the system's control output O across multiple modules, possibly simultaneously. In a pure sequential system, the control flow is a linear sequence through a string of modules. Here, the control flow

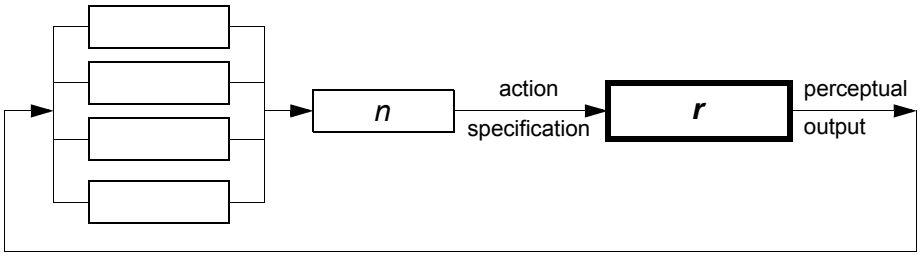

Figure 6.26
Example of a pure parallel decomposition.

contains a *combination* step at which point the result of multiple modules may impact O in arbitrary ways.

Thus parallelization of control leads to an important question: how will the output of each component module inform the overall decision concerning the value of O? One simple combination technique is temporal switching. In this case, called *switched parallel*, the system has a parallel decomposition but at any particular instant in time the output O can be attributed to one specific module. The value of O can of course depend on a different module at each successive time instant, but the instantaneous value of O can always be determined based on the functions of a single module. For instance, suppose that a robot has an obstacle avoidance module and a path-following module. One switched control implementation may involve execution of the path-following recommendation whenever the robot is more than 50 cm from all sensed obstacles and execution of the obstacle-avoidance recommendation when any sensor reports a range closer than 50 cm.

The advantage of such switched control is particularly clear if switching is relatively rare. If the behavior of each module is well understood, then it is easy to characterize the behavior of the switched control robot: it will obstacle avoid at times, and it will path-follow other times. If each module has been tested independently, there is a good chance the switched control system will also perform well. Two important disadvantages must be noted. First, the overall behavior of the robot can become quite poor if the switching is itself a high-frequency event. The robot may be unstable in such cases, switching motion modes so rapidly as to dramatically devolve into behavior that is neither path-following nor obstacle-avoiding. Another disadvantage of switched control is that the robot has no path-following bias when it is obstacle avoiding (and vice versa). Thus in cases where control *ought* to mix recommendations from among multiple modules, the switched control methodology fails.

In contrast, the much more complex *mixed parallel* model allows control at any given time to be shared between multiple modules. For example, the same robot could take the obstacle avoidance module's output at all times, convert it to a velocity vector, and combine

it with the path-following module's output using vector addition. Then the output of the robot would never be due to a single module, but would result from the mathematical combination of both modules outputs. Mixed parallel control is more general than switched control, but by that token it is also a more challenging technique to use well. Whereas with switched control most poor behavior arises out of inopportune switching behavior, in mixed control the robot's behavior can be quite poor even more readily. Combining multiple recommendations mathematically does not guarantee an outcome that is globally superior, just as combining multiple vectors when deciding on a swerve direction to avoid an obstacle can result in the very poor decision of going straight ahead. Thus, great care must be taken in mixed parallel control implementations to fashion mixture formulas and individual module specifications that lead to effective mixed results.

Both the switched and mixed parallel architectures are popular in the behavior robotics community. Arkin [2] proposes the *motor-schema* architecture in which *behaviors* (i.e., modules in the earlier discussion) map sensor value vectors to motor value vectors. The output of the robot algorithm is generated, as in mixed parallel systems, using a linear combination of the individual behavior outputs. In contrast, Maes [201, 202] produces a switched parallel architecture by creating a *behavior network* in which a behavior is chosen discretely by comparing and updating activation levels for each behavior. The subsumption architecture of Brooks [82] is another example of a switched parallel architecture, although the active model is chosen via a suppression mechanism rather than activation level. For further discussion, see [2].

One overall disadvantage of parallel control is that verification of robot performance can be extremely difficult. Because such systems often include truly parallel, multithreaded implementations, the intricacies of robot-environment interaction and sensor timing required to represent properly all conceivable module-module interactions can be difficult or impossible to simulate. So, much testing in the parallel control community is performed empirically using physical robots.

An important advantage of parallel control is its biomimetic aspect. Complex organic organisms benefit from large degrees of true parallelism (e.g., the human eye), and one goal of the parallel control community is to understand this biologically common strategy and leverage it to advantage in robotics.

6.5.4 Case studies: tiered robot architectures

We have described temporal and control decompositions of robot architecture, with the common theme that the roboticist is always composing multiple modules together to make up that architecture. Let us turn again toward the overall mobile robot navigation task with this understanding in mind. Clearly, robot behaviors play an important role at the real-time levels of robot control, for example, path-following and obstacle avoidance. At higher temporal levels, more tactical tasks need to modulate the activation of behaviors, or modules,

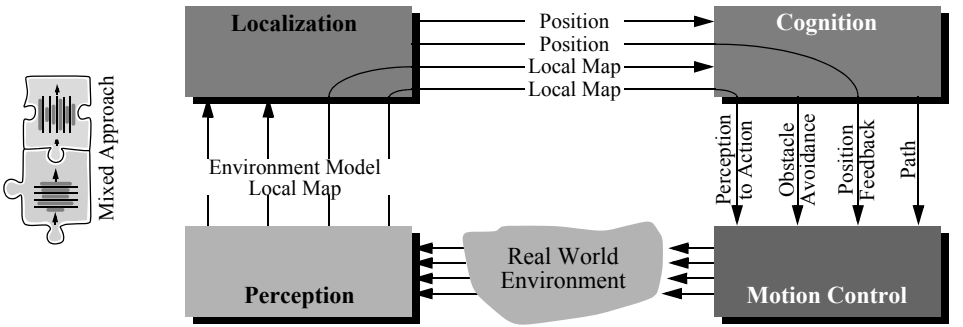

Figure 6.27
The basic architectural example used throughout this text.

in order to achieve robot motion along the intended path. Higher still, a global planner could generate paths to provide tactical tasks with global foresight.

In chapter 1, we introduced a functional decomposition showing such modules of a mobile robot navigator from the perspective of information flow. The relevant figure is shown here again as figure 6.27.

In such a representation, the arcs represent aspects of real-time and non-real-time competence. For instance, obstacle avoidance requires little input from the localization module and consists of fast decisions at the cognition level followed by execution in motion control. In contrast, PID position feedback loops bypass all high-level processing, tying the perception of encoder values directly to lowest-level PID control loops in motion control. The trajectory of arcs through the four software modules provides temporal information in such a representation.

Using the tools of this chapter, we can now present this same architecture from the perspective of a temporal decomposition of functionality. This is particularly useful because we wish to discuss the interaction of strategic, tactical, and real-time processes in a navigation system.

Figure 6.28 depicts a generic tiered architecture based on the approach of Pell and colleagues [256] used in designing an autonomous spacecraft, *Deep Space One*. This figure is similar to figure 6.24 in presenting a temporal decomposition of robot competence. However, the boundaries separating each module from adjacent modules are specific to robot navigation.

Path planning embodies strategic-level decision making for the mobile robot. Path planning uses all available global information in non-real-time to identify the right sequence of local actions for the robot. At the other extreme, *real-time control* represents competences requiring high bandwidth and tight sensor-effector control loops. At its lowest level, this

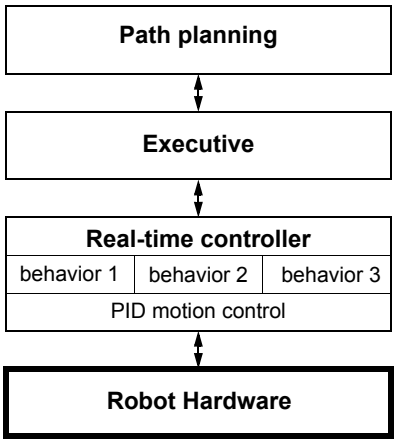

Figure 6.28
A general tiered mobile robot navigation architecture based on a temporal decomposition.

includes motor velocity PID loops. Above those, real-time control also includes low-level behaviors that may form a switch or mixed parallel architecture.

In between the path planner and real-time control tiers sits the *executive*, which is responsible for mediating the interface between planning and execution. The executive is responsible for managing the activation of behaviors based on information it receives from the planner. The executive is also responsible for recognizing failure, saving (placing the robot in a stable state), and even reinitiating the planner as necessary. It is the executive in this architecture that contains all tactical decision making as well as frequent updates of the robot's short-term memory, as is the case for localization and mapping.

It is interesting to note the similarity between this general architecture, used in many specialized forms in mobile robotics today, and the architecture implemented by Shakey, one of the very first mobile robots, in 1969 [242]. Shakey had *LLA* (low-level actions) that formed the lowest architectural tier. The implementation of each LLA included the use of sensor values in a tight loop just as in today's behaviors. Above that, the middle architectural tier included the *ILA* (intermediate-level actions), which would activate and deactivate LLA as required based on perceptual feedback during execution. Finally, the topmost tier for Shakey was STRIPS (Stanford Research Institute Planning System), which provided global look ahead and planning, delivering a series of tasks to the intermediate executive layer for execution.

Although the general architecture shown in figure 6.28 is useful as a model for robot navigation, variant implementations in the robotics community can be quite different. Next, we present three particular versions of the general tiered architecture, describing for each

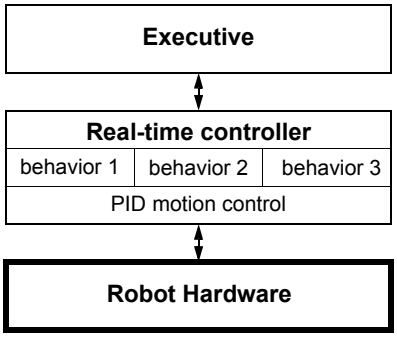

Figure 6.29
A two-tiered architecture for offline planning.

version at least one real-world mobile robot implementation. For broader discussions of various robot architectures, see [39].

6.5.4.1 Offline planning

Certainly the simplest possible integration of planning and execution is no integration at all. Consider figure 6.29, in which there are only two software tiers. In such navigation architectures, the executive does not have a planner at its disposal but must contain a priori all relevant schemes for traveling to desired destinations.

The strategy of leaving out a planner altogether is of course extremely limiting. Moving such a robot to a new environment demands a new instantiation of the navigation system, and so this method is not useful as a general solution to the navigation problem. However such robotic systems do exist, and this method can be useful in two cases.

Static route applications. In mobile robot applications where the robot operates in a completely static environment using a route navigation system, it is conceivable that the number of discrete goal positions is so small that the environmental representation can directly contain paths to all desired goal points. For example, in factory or warehouse settings, a robot may travel a single looping route by following a buried guidewire. In such industrial applications, path-planning systems are sometimes altogether unnecessary when a precompiled set of route solutions can be easily generated by the robot programmers. The Chips mobile robot is an example of a museum robot that also uses this architecture [251]. Chips operates in a unidirectional looping track defined by its colored landmarks. Furthermore, it has only twelve discrete locations at which it is allowed to stop. Due to the simplicity of this environmental model, Chips contains an executive layer that directly caches

the path required to reach each goal location rather than a generic map with which a path planner could search for solution paths.

Extreme reliability demands. Not surprisingly, another reason to avoid online planning is to maximize system reliability. Since planning software can be the most sophisticated portion of a mobile robot's software system, and since in theory at least planning can take time exponential to the complexity of the problem, imposing hard temporal constraints on successful planning is difficult if not impossible. By computing all possible solutions offline, the industrial mobile robot can trade versatility for effective constant-time planning (while sacrificing significant memory of course). A real-world example of offline planning for this reason can be seen in the contingency plans designed for space shuttle flights. Instead of requiring astronauts to solve problems online, thousands of conceivable issues are postulated on Earth, and complete conditional plans are designed and published in advance of the shuttle flights. The fundamental goal is to provide an absolute upper limit on the amount of time that passes before the astronauts begin resolving the problem, sacrificing a great deal of ground time and paperwork to achieve this performance guarantee.

6.5.4.2 Episodic planning

The fundamental information-theoretic disadvantage of planning offline is that, during runtime, the robot is sure to encounter perceptual inputs that provide information, and it would be rational to take this additional information into account during subsequent execution. Episodic planning is the most popular method in mobile robot navigation today because it solves this problem in a computationally tractable manner.

As shown in figure 6.30, the structure is three-tiered, as is the general architecture of figure 6.28. The intuition behind the role of the planner is as follows. Planning is computationally intensive, and therefore planning too frequently would have serious disadvantages. But the executive is in an excellent position to identify when it has encountered enough information (e.g., through feature extraction) to warrant a significant change in strategic direction. At such points, the executive will invoke the planner to generate, for example, a new path to the goal.

Perhaps the most obvious condition that triggers replanning is detection of a blockage on the intended travel path. For example, in [281] the path-following behavior returns failure if it fails to make progress for a number of seconds. The executive receives this failure notification, modifies the short-term occupancy grid representation of the robot's surroundings, and launches the path planner in view of this change to the local environment map.

A common technique to delay planning until more information has been acquired is called *deferred planning*. This technique is particularly useful in mobile robots with dynamic maps that become more accurate as the robot moves. For example, the commercially available Cye robot can be given a set of goal locations. Using its grassfire breadth-

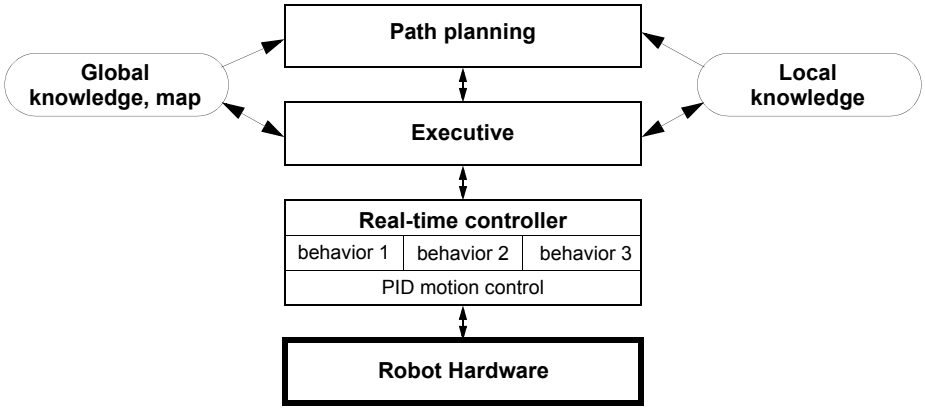

Figure 6.30
A three-tiered episodic planning architecture.

first planning algorithm, this robot will plot a detailed path to the closest goal location only and will execute this plan. Upon reaching this goal location, its map will have changed based on the perceptual information extracted during motion. Only then will Cye's executive trigger the path planner to generate a path from its new location to the next goal location.

The robot Pygmalion implements an episodic planning architecture along with a more sophisticated strategy when encountering unforeseen obstacles in its way [58, 259]. When the lowest-level behavior fails to make progress, the executive attempts to find a way past the obstacle by turning the robot 90 degrees and trying again. This is valuable because the robot is not kinematically symmetric, and so servoing through a particular obstacle course may be easier in one direction than the other.

Pygmalion's environment representation consists of a continuous geometric model as well as an abstract topological network for route planning. Thus, if repeated attempts to clear the obstacle fail, then the robot's executive will temporarily cut the topological connection between the two appropriate nodes and will launch the planner again, generating a new set of waypoints to the goal. Next, using recent laser rangefinding data as a type of local map (see figure 6.30), a geometric path planner will generate a path from the robot's current position to the next waypoint.

In summary, episodic planning architectures are extremely popular in the mobile robot research community. They combine the versatility of responding to environmental changes and new goals with the fast response of a tactical executive tier and behaviors that control real-time robot motion. As shown in figure 6.30, it is common in such systems to have both a short-term local map and a more strategic global map. Part of the executive's job in such

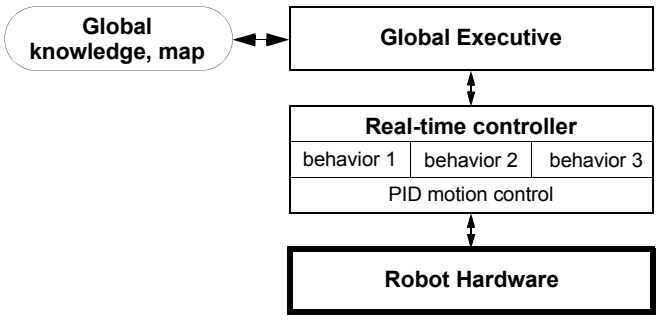

Figure 6.31
An integrated planning and execution architecture in which planning is nothing more than a real-time execution step (behavior).

dual representations is to decide when and if new information integrated into the local map is sufficiently nontransient to be copied into the global knowledge base.

6.5.4.3 Integrated planning and execution

Of course, the architecture of a commercial mobile robot must include more functionality than just navigation. But limiting this discussion to the question of *navigation* architectures leads to what may at first seem a degenerate solution.

The architecture shown in figure 6.31 may look similar to the offline planning architecture of figure 6.29, but in fact it is significantly more advanced. In this case, the planner tier has disappeared because there is no longer a temporal decomposition between the executive and the planner. Planning is simply one small part of the executive's nominal cycle of activities, where the local and global representations are the same. The advantage of this approach is that the robot's actions at every cycle are guided by a global path planner, and they are therefore optimal in view of all of the information the robot has gathered.

Integrated planning and execution has largely been made feasible due to innovations in graph search algorithms (e.g. the D* algorithm described on page page 385) and graph representation (e.g., state lattice graphs, whose edges can be inherently executed by the robotic platform). As a result, formidable real-time implementations devoid of obstacle avoidance modules have emerged: Pivtoraiko et al. [260] showed that graph search on a 3D state lattice (including 2D position and heading) can be as efficient as 2D grid search. In the work of Ferguson et al. [127], an extension to this strategy was successfully employed to navigate an autonomous car in large scale parking lots. Rufli et al. [271, 272] added velocity dimensions to the aforementioned state lattice representation which allowed them to take into account position and velocity information of nearby dynamic obstacles during the planning step. The result is a feasible, globally optimal, time-parametrized path that inherently

avoids dynamic obstacles, a task that has traditionally been carried out by local collision avoidance modules.

The described methods naturally face limits of applicability as the size of the environment increases. This issue can be accounted for by applying multiresolution graph approaches, however. Close to the robot, a high-fidelity lattice may be employed, farther away a lower-resolution one. Hundreds of meters distant, the lattice may transition into a road network such as the ones often used in commercial GPS-based navigation systems.

Still, the recent success of integrated planning and execution methods underlines the fact that the designer of a robot navigation architecture must consider not only all aspects of the robot and its environmental task but also the state of processor, GPU, and memory technology. We expect that mobile robot architecture design is sure to remain an active area of innovation for years to come. All forms of technological progress, from robot sensor inventions to processor speed increases, and further parallelization are likely to catalyze new innovations in mobile robot architecture as previously unimaginable tactics become realizable.

6.6 Problems

1. Consider completeness and optimality properties for each of:

Visibility graph
Voronoi diagram
Exact cell decomposition
Approximate cell decomposition

In the framework of path planning, categorize for each whether it is complete/incomplete, and optimal/ not guaranteed-optimal for path planning.

2. Consider an Ackerman steering 4-wheel high speed Martian rover. Consider all the obstacle avoidance techniques described in 6.2. Explain for every option its advantage or disadvantage, in one sentence each, for this specific application. Specifically do so for: Schlegel, local dynamic window, LCM, CVM, VFH, Bubble band, Bug.

3. Consider an autonomous driving robot for highway driving. Propose a temporal decomposition, as in figure 6.24, with at least five levels, describing control frequency at each level and the specific driving skills/behaviors incorporated at that level.

4. Challenge Question.
Consider a robot that navigates with range sensors that have a limited useful range r. Propose a path-planning method based on the ones in 6.3 that is complete and maintains a safe distance from objects while also staying within distance r from objects whenever possible.