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: $q_1, \ldots, q_k$. 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.

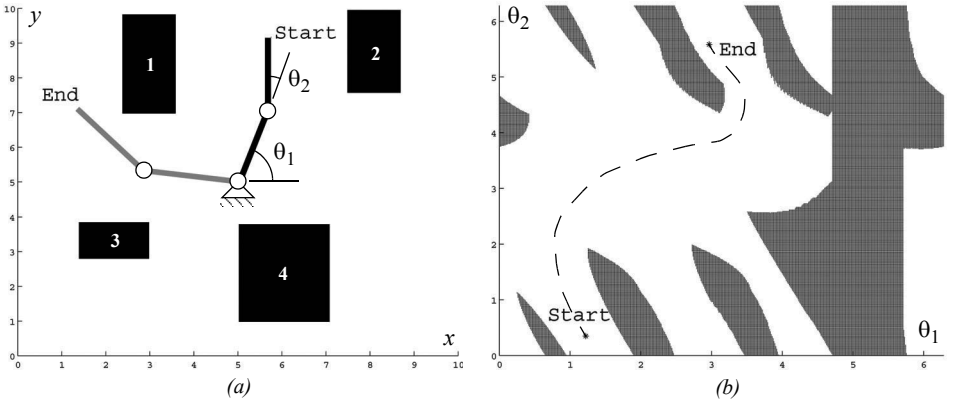

Figure 6.1
Physical space (a) and configuration space (b): (a) A two-link planar robot arm has to move from the configuration *start* to *end*. The motion is thereby constraint by the obstacles 1 to 4. (b) The corresponding configuration space shows the free space in joint coordinates (angle θ_1 and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

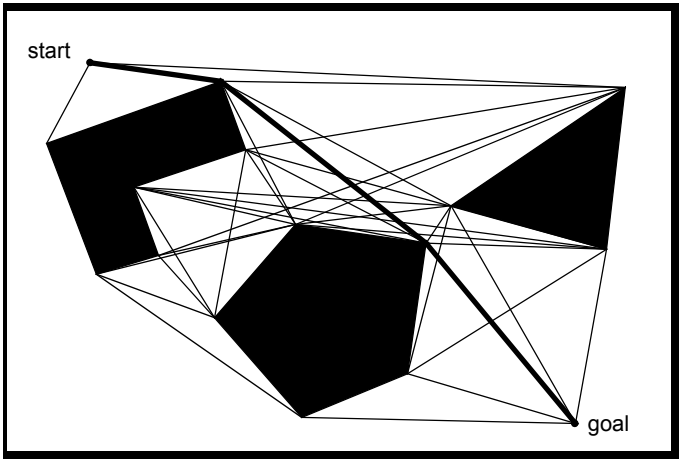

Figure 6.2
Visibility graph [32]. The nodes of the graph are the initial and goal points and the vertices of the configuration space obstacles (polygons). All nodes which are visible from each other are connected by straight-line segments, defining the road map. This means there are also edges along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

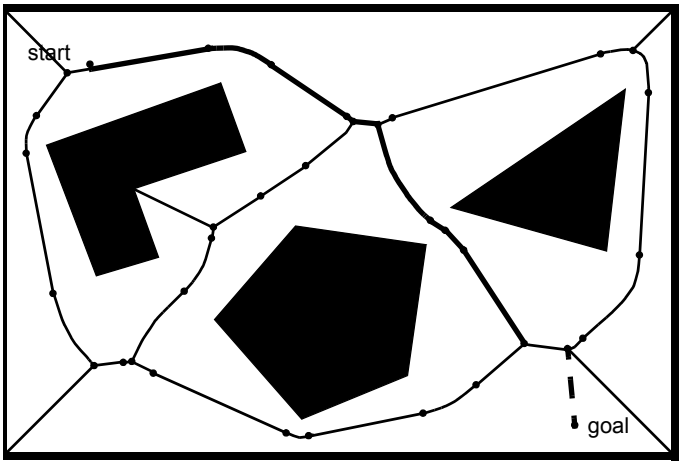

Figure 6.3
Voronoi diagram [32]. The Voronoi diagram consists of the lines constructed from all points that are equidistant from two or more obstacles. The initial q_{init} and goal q_{goal} configurations are mapped into the Voronoi diagram to q'_{init} and q'_{goal}, each by drawing the line along 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

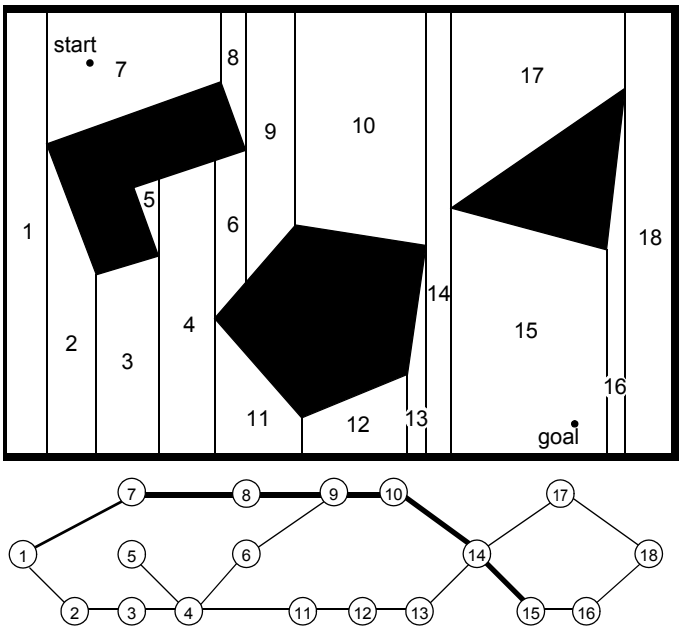

Figure 6.4
Example of exact cell decomposition. Cells are for example divided according to the horizontal coordinate of extremal obstacle points.

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

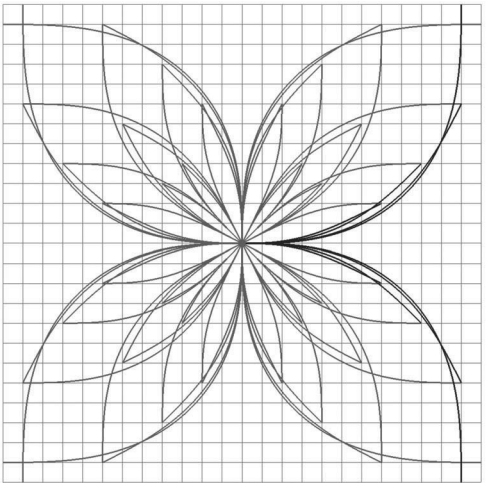

Figure 6.5 16-directional state lattice constructed for a planetary exploration rover. The state includes 2D position, heading, and curvature (x, y, θ, k). Note that straight segments are part of the lattice set but occluded by longer curved segments. The lattice is 2D-shift invariant and partially invariant 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

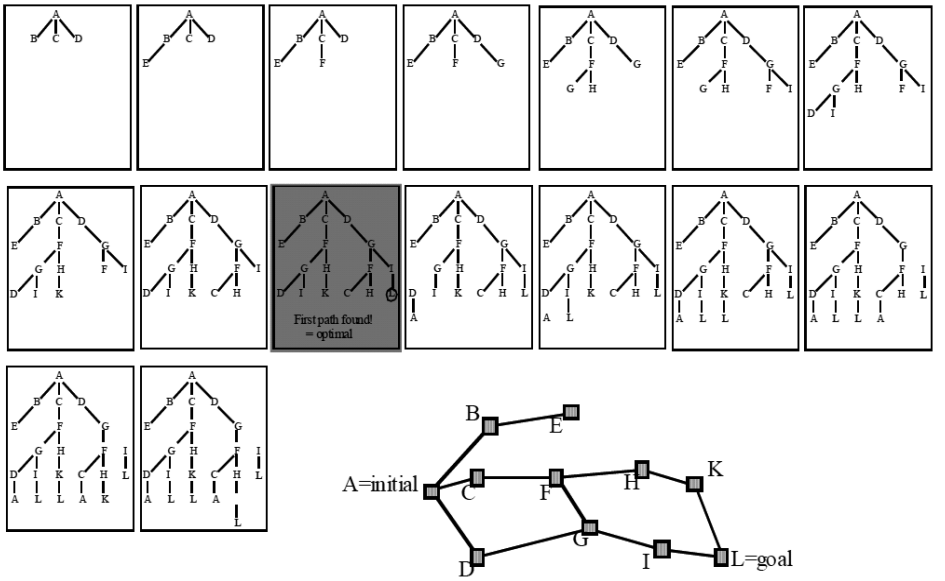

Figure 6.6 Working principle of breadth-first search.

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

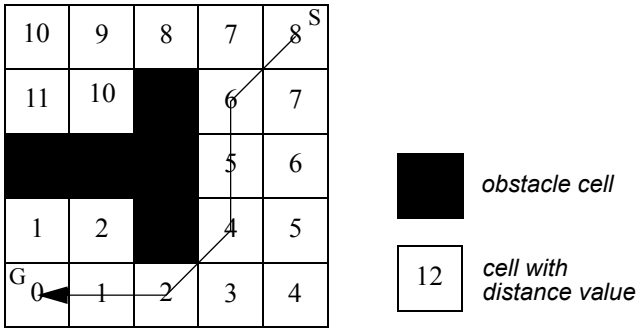

Figure 6.7
An example of the distance transform and the resulting path as it is generated by the NF1 function. S denotes the start, G the goal. The neighbors of each cell i are defined as the four adjacent cells that share an edge with i (4-neighborhood).

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

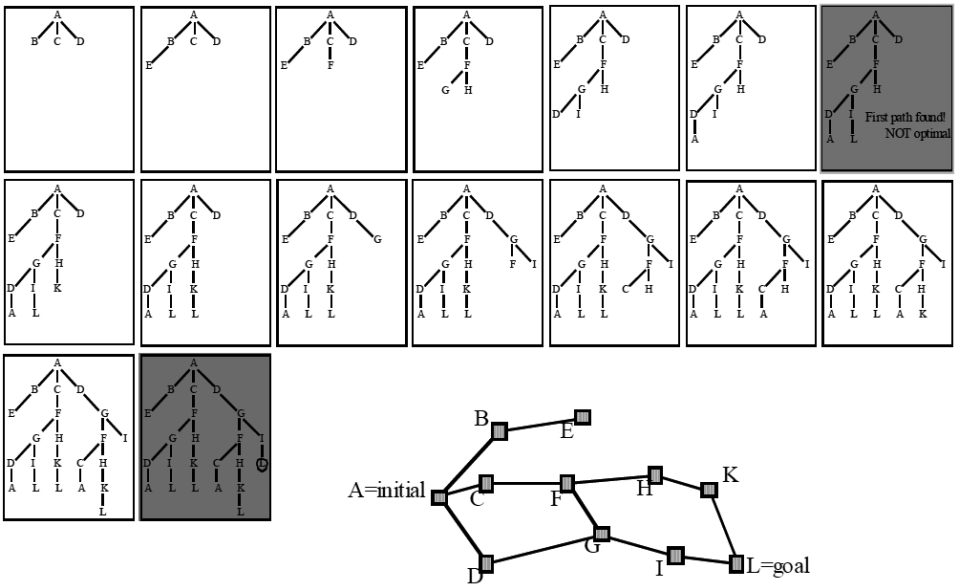

Figure 6.8 Working principle of depth-first search.

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

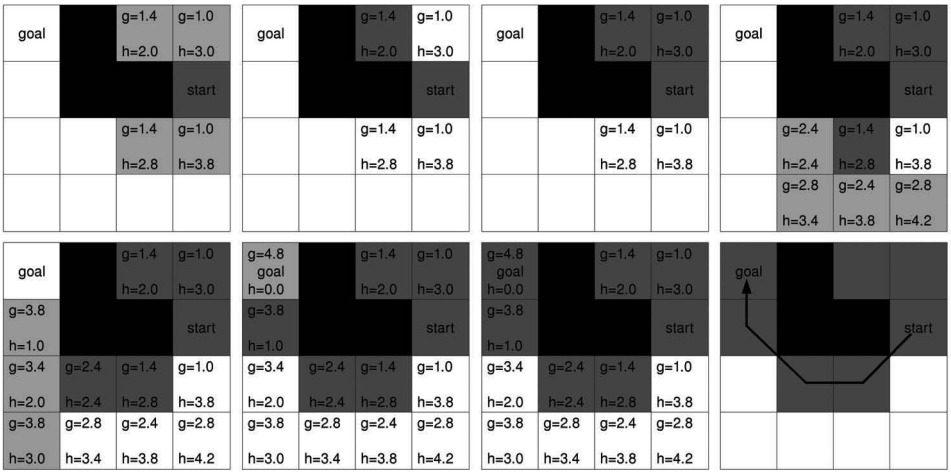

Figure 6.9 Working principle of the A* algorithm. Nodes are expanded in order of lowest $f(n) = g(n) + h(n)$ cost. $g(n)$ is indicated at the top left corner, $h(n)$ at the bottom right corner of each cell. The neighborhood of each cell is selected as the 8-neighborhood (all eight adjacent cells). Diagonal 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

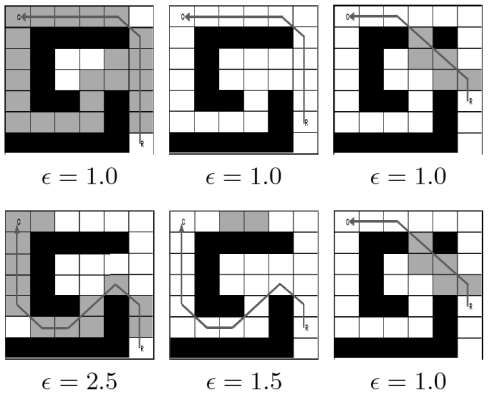

Figure 6.10 comparison of the number of expanded cells for D* (top) and Anytime D* (bottom, starting with a sub-optimality value of 2.5) in a planning and re-planning scenario. Note that an opening in the top wall is detected in the third frame, after the robot has moved upward twice. Obstacle cells 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-

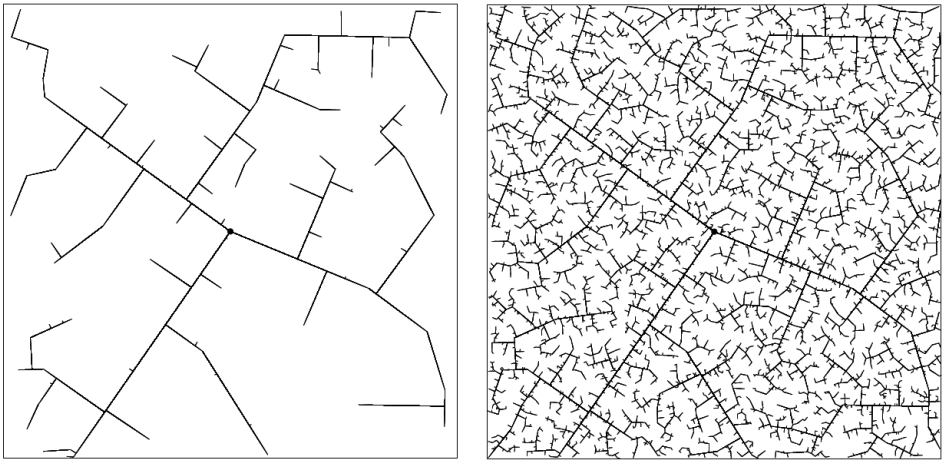

Figure 6.11 The evolution of a RRT. Image courtesy of S.M. LaValle [33].

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.

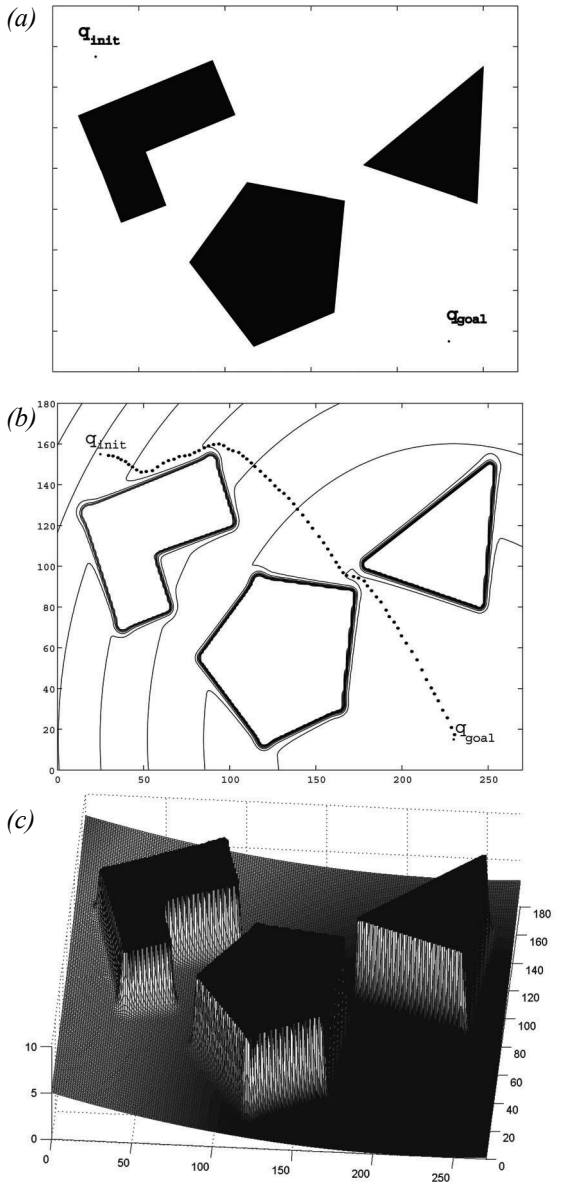

Figure 6.12
Typical potential field generated by the attracting goal and two obstacles (see [32]). (a) Configuration of the obstacles, start (top left) and goal (bottom right). (b) Equipotential plot and path generated by the field. (c) Resulting potential field generated by the goal attractor and obstacles.

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

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.

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

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