CHAPTER 4

Path Planning

4.1 INTRODUCTION

Path planning from location A to location B, simultaneous obstacle, avoidance and reacting to environment changes are simple tasks for humans but not so straightforward for an autonomous vehicle. These tasks present challenges that each wheeled mobile robot needs to overcome to become autonomous. A robot uses sensors to perceive the environment (up to some degree of uncertainty) and to build or update its environment map. In order to determine appropriate motion actions that lead to the desired goal location, it can use different decision and planning algorithms. In the process of path planning, the robot's kinematic and dynamic constraints should be considered.

Path planning is used to solve problems in different fields, from simple spatial route planning to selection of an appropriate action sequence that is required to reach a certain goal. Since the environment is not always known in advance, this type of planning is often limited to the environments designed in advance and environments that we can describe accurately enough before the planning process. Path planning can be used in fully known or partly known environments, as well as in entirely unknown environments where sensed information defines the desired robot motion.

Path planning in known environments is an active research area and presents a foundation for more complex cases where the environment is not known a priori. This chapter presents an overview of the most often used path planning approaches applicable to wheeled mobile robots. First, some basic definitions of path planning are given. For more detailed information and further reading see also [1–3].

4.1.1 Robot Environment

The environment in which the robot moves consists of *free space* and space *occupied by the obstacles* (Fig. 4.1). The start and goal configuration are located in free space. Configuration is a complete set of parameters that define the robot in space. These parameters usually include robot position and

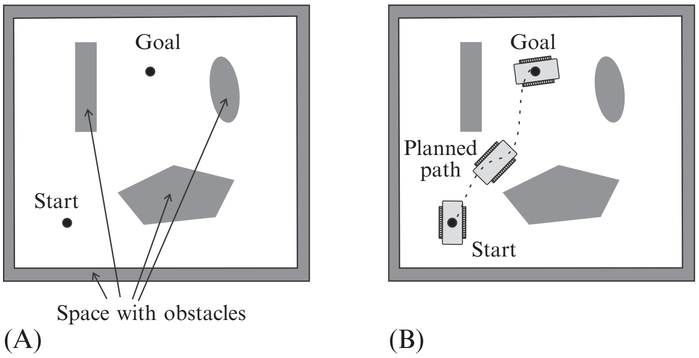

Fig. 4.1 (A) Environment with obstacles and start and goal configurations, and (B) one out of many possible paths from the start to the goal configuration.

orientation or also its joints angles. The number of these parameters equals to the number of degrees of freedom (DOF).

The environment that includes moving obstacles is the *dynamic environment*, while the *static environment* does not change with time. The environment with known positions of the obstacles is called known the environment; otherwise, the environment is unknown.

4.1.2 Path Planning

Path planning is the task of finding a continuous path that will drive the robot from the start to the goal configuration. The entire path must lie in the free space (as shown in Fig. 4.1). In path planning the mobile system uses known environment map, which is stored in the robot's memory.

The state or configuration gives a possible pose of the robot in the environment, and it can be represented as a point in the configuration space that includes all possible robot states (configurations). The robot can move from one state to the other by implementing different actions. A suitable path is therefore described as a sequence of actions that guide the robot from the start configuration (state) through some intermediate configurations to the goal configuration. Which action is chosen in the current state and which state will be next depends on the used path planning algorithm and used criteria. The algorithm chooses the next most suitable state from the set of all possible states that can be visited from the current state. This decision is made according to some criteria function, usually defined with one of the distance measures, such as the shortest Euclidean distance to the goal state.

Between certain start and goal states, there may be one or more paths, or there is no path at all that connects the states. Usually there are several feasible paths (i.e., paths that do not collide with obstacles). To narrow the selection choice additional demands or criteria are introduced that define desired optimality:

- the path length must be the shortest,
- the suitable path is the one in which the robot can pass in the shortest time,
- the path should be as far as possible from the obstacles,
- the path must be smooth without sharp turns,
- the path must consider motion constraints (e.g., nonholonomic constraints, where at the current time not all driving directions are possible).

4.1.3 Configuration and Configuration Space

Configuration is the state of the system in the environment space spanned in n dimensions. The state vector is described by $\boldsymbol{q} = [q_1, \ldots, q_n]^T$. The state $\boldsymbol{q}$ is a point in n-dimensional space called *configuration space* Q, which presents all possible configurations of the mobile system according to its kinematic model.

Part of the configuration space that is occupied by the obstacles O_i is denoted $Q_{\mathrm{obst}} = \bigcup_i O_i$. So the free part of the environment without the obstacles is defined by

$$Q_{\mathrm{free}} = Q - Q_{\mathrm{obst}}$$

Q_{free} therefore contains space where the mobile system can plan its motion.

Assume there is a circular robot that can only execute translatory motions in a plane (i.e., it has two DOF $\boldsymbol{q} = [x, y]^T$) then its configuration can be defined by a point (x, y) presenting the robot center. In this case the configuration space Q is determined by moving the robot around obstacles while keeping it in contact with the obstacle, as shown in Fig. 4.2. In doing so the robot center point describes the border between Q_{free} and Q_{obs}. In other words, the obstacles are enlarged by the known robot dimension (radius) in order to treat the robot as a point in the path planning problem.

An additional example of configuration space for a triangular robot and rectangular obstacle is shown in Fig. 4.3. The robot can only move in x and y directions ($\boldsymbol{q} = [x, y]^T$).

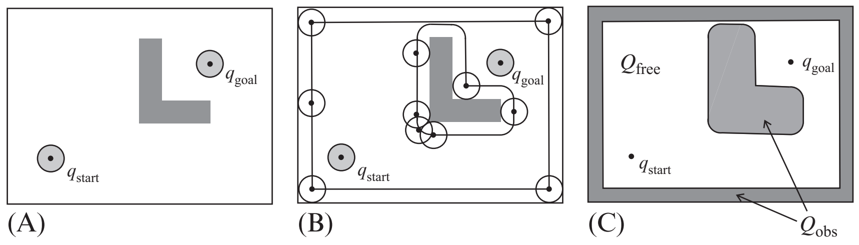

Fig. 4.2 (A) Robot with a circular shape in the environment containing an obstacle with marked start and goal configurations, (B) configuration space definition, and (C) transformed environment from (A) to a configuration space where the robot is presented simply by its center point.

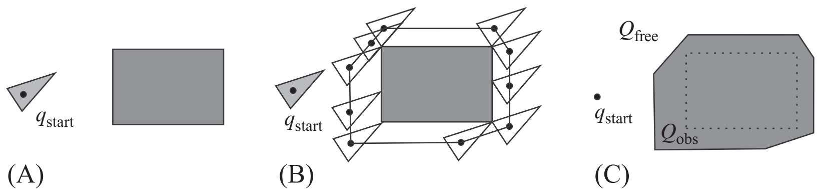

Fig. 4.3 (A) Rectangular obstacle and triangular robot with a point defining its configuration q, (B) configuration space determination, (C) free configuration Q_{free}, and space occupied by the obstacle Q_{obs}.

If the robot from Fig. 4.3 could also rotate then its configuration would have three dimensions $q = [x, y, \varphi]^T$ and the configuration space would be more complex. By simplification we could determine the configuration space using a circle that outlines the robot shape and has a center in the robot center. Obtained configuration Q_{free} is then smaller than the true free configuration because the outlined circle area is bigger than the robot area. However, this significantly simplifies the path planning problem.

4.1.4 Mathematical Description of Obstacle Shape and Pose in the Environment

The mathematical description of obstacle shape and position is required to estimate robot configuration space and additional environment simplifications. The two most common approaches to describe the obstacles are border presentation by recording vertices and presentation given with half-planes.

Border-Based Description of Obstacles

The obstacle ground plan is an m-sided polygon and has m vertices. The obstacle border (shape) can be described by its vertices that are listed in counterclockwise order. The holes in the obstacles and surrounding walls are described by vertices in the clockwise order (Fig. 4.4). Such an obstacle description is valid for convex and nonconvex polygons.

Obstacle Description Using Intersecting Half-Planes

A convex polygon with m vertices can be described by a union of m half planes, where the half plane is defined as $f(x, y) \leq 0$ for straight lines or $f(x, y, z) \leq 0$ for planes (3D obstacles). An example of how to describe a polygon with five vertices is given in Fig. 4.5. Nonconvex shapes or shapes with holes can be described with the use of operations over sets such as union, intersections, difference among sets, and the like.

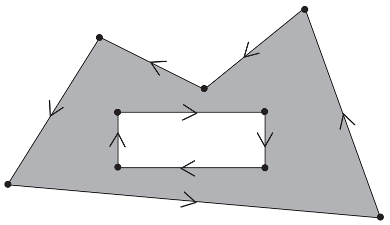

Fig. 4.4 Example of an obstacle description with listed vertices: counterclockwise for the outer obstacle boundary and clockwise for its holes. The left-hand side of each arrowed straight line segment belongs to the obstacle (*shaded area*).

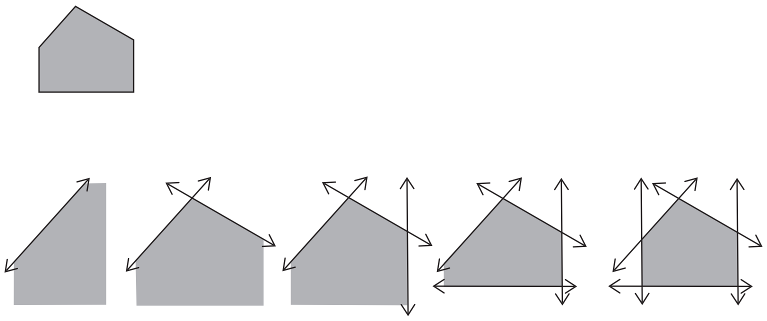

Fig. 4.5 Example of an obstacle description by half planes.

4.2 ENVIRONMENT PRESENTATION FOR PATH PLANNING PURPOSES

Before path planning, the environment needs to be presented in a unified mathematical manner that is suitable for processing in path searching algorithms.

4.2.1 Graph Representation

Configuration space consists of free space, which contains all possible configurations (states) of the mobile system, and space occupied by obstacles. If free space is reduced and presented by a subset of configurations (e.g., centers of cells) that include start and goal configurations, the desired number of intermediate configurations and transitions among them, then the *state transition graph* (also graph) is obtained. States are presented by circles and called nodes, and connections among them are given by lines. The connections (lines) therefore present actions required for the system to move among the states.

In a *weighted graph* each connection has some prescribed weight or cost that is needed for action execution and transition among the states of this connection. In a *directed graph* the connections are also marked with possible directions. In a directed graph the cost depends on the direction of transition, while in an undirected graph the transition is possible in both directions. Fig. 4.6 shows an example of a weighed and directed weighted graph. Path searching in a graph is possible with different algorithms such as A*, Dijkstra's algorithm, and the like.

4.2.2 Cell Decomposition

The environment can be partitioned to *cells*, which are usually defined as simple geometric shapes. Cells must be convex because each straight line segment connecting any two configurations inside the cell must lie entirely

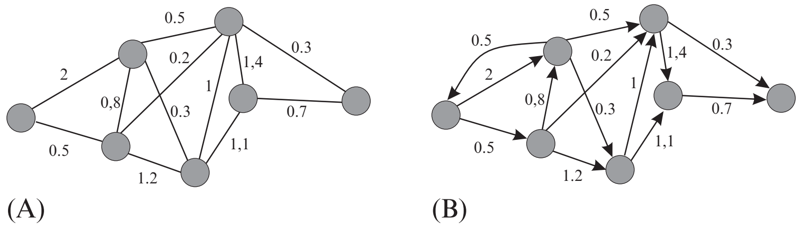

Fig. 4.6 Example of (A) weighted graph, and (B) directed and weighted graph.

in that cell. An environment partitioned to cells can be presented with a state transition graph where states are certain points inside the cells (e.g., centers) and connections among the states (nodes) are possible only between neighbor cells (e.g., the ones with a common edge).

Accurate Decomposition to Cells

Decomposition to cells is accurate if all the cells lie entirely in the free space or entirely in the space occupied by the obstacles. *Accurate decomposition* is therefore without losses because the union of all free cells equals the free configuration space Q_{free}.

An example of accurate decomposition to cells is vertical decomposition, shown in Fig. 4.7. This decomposition can be obtained by using an imaginary vertical line traveling from the left environment border to the right border. Whenever an obstacle corner appears (vertices of some polygon) a vertical border between the cells is made, traveling from the corner only up or only down, or up and down. The complexity of this approach depends on the geometry of the environment. For simple environments the number of cells and the connection among them is small; while with the increasing number of polygons (obstacles) and their vertices, the number of cells also increases.

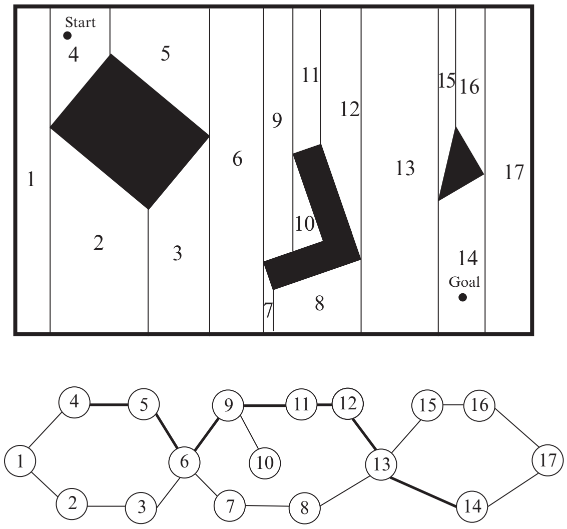

Fig. 4.7 Vertical cell decomposition (*top*) and the associated graph (*bottom*) with a marked path among the start and the goal configuration.

Accurate decomposition to cells can be presented with a state transition graph where graph nodes are, for example, the cells' centers and the transitions among the cells go, for example, through the middle points of the borders between the cells.

Approximate Decomposition to Cells

Environment decomposition to cells is approximate when it is possible that certain cells contain free configuration space as well as an obstacle or only part of some obstacle. All cells containing at least part of the obstacle are therefore normally marked as occupied, while the remaining cells are marked as free. If decomposition to the cells is made of the same size, then an *occupancy grid* (Fig. 4.8) is obtained.

The center of each cell (in Fig. 4.8, the free cell is marked with white) is presented as a node in a graph. Transitions among the cells are in four or eight directions if diagonal transitions are allowed. This approach is very simple for application, but due to the constant cell size some information about the environment might get lost (this is not lossless decomposition); for example, obstacles are enlarged and narrow passages among them might get lost. The main drawback of this approach is the memory usage, which increases either by enlarging the size of the environment or by lowering the cell size.

A smaller loss of the environment information at lower memory usage is obtained if the variable cells size is used. A representative example of the latter are quadtrees where initially a single cell is put over the environment. If entire cell is in free space or the entire cell is occupied by the obstacle then the cell remains as is. Otherwise, if this cell is only partly occupied by an obstacle, then it is split into smaller cells. This procedure is repeated until

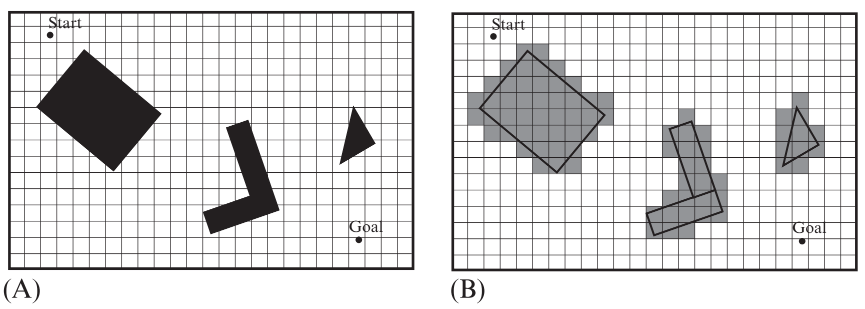

Fig. 4.8 Approximate decomposition to cells obtained in two steps: (A) laying a grid of equal cells on the environment and (B) marking cells as free or occupied (*shaded cells*).

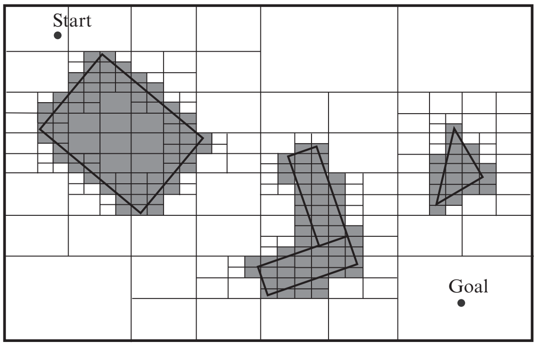

Fig. 4.9 Approximate decomposition to cells using variable cell size: quadtree. Free cells are marked as *white* and occupied cells are *shaded*.

there is a desired resolution. An example of a quadtree is given in Fig. 4.9, and its presentation can be described with a state graph. Approximate decomposition to cells is simpler than the accurate decomposition to cells; however, it can happen that due to the loss of information a path planning algorithm may not be able to find a solution although it does exist in the environment.

EXAMPLE 4.1

Write a program that performs a quadtree decomposition of some environment with a rectangular shape of the obstacles. A framework that generates a random environment with obstacles is given in Listing 4.1, where a function for the calculation of quadtree is already declared. The function accepts the obstacles, environment dimension, and desired tree depth, and it returns a quadtree in the form of a Matlab structure.

Listing 4.1 Framework for implementation of quadtree decomposition

```
bb = [0, 16, 0, 12]; % Environment dimension: xa, xb, ya, yb
N = 10; % Number of obstacles
minDim = [0.1; 0.1]; % Min dimension of an obstacle: xMin and yMin
maxDim = [2; 2]; % Max dimension of an obstacle: xMax and yMax
% Random obstacle map, vertices in a column: x1, y1, x2, y2, x3, y3, x4, y4
obst = zeros(8, N);
for i = 1:N
    p = [bb(1); bb(3)] + [diff(bb(1:2)); diff(bb(3:4))].*rand(2,1);
    phi = 2*pi*rand(); d = minDim/2 + (maxDim-minDim)/2*rand();
    R = [cos(phi),-sin(phi); sin(phi), cos(phi)];
    v = repmat(p, 1, 4) + R*([-1, 1, 1,-1; -1,-1, 1, 1].*repmat(d, 1, 4));
    obst(:,i) = reshape(v, [], 1);
end

tree = quadTree(obst, bb, 4); % Create quad tree with depth-level 4
```

(*Continued*)

EXAMPLE 4.1—cont'd

Solution

A possible implementation of a quadtree algorithm in Matlab is given in Listing 4.2. The result of the algorithm on a randomly generated map is visualized in Fig. 4.10.

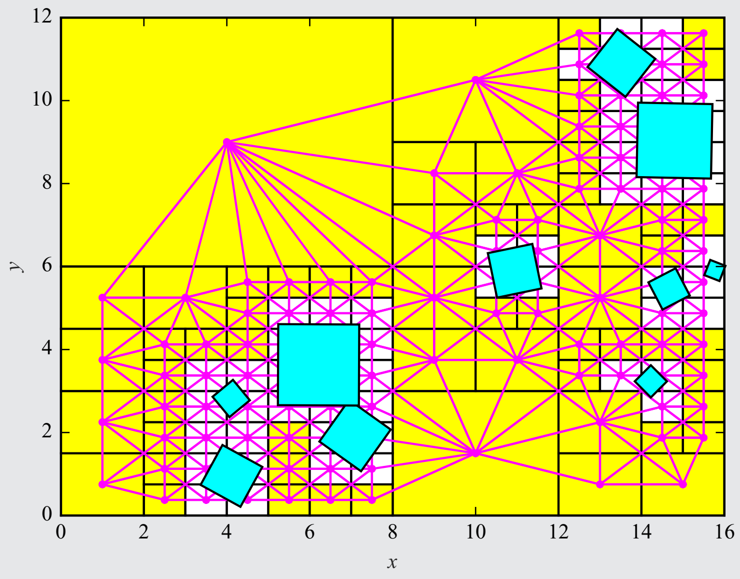

Fig. 4.10 Quadtree decomposition of the environment with random obstacles. The quadtree is overlaid with a mesh that connects the centers of all the neighboring cells.

Listing 4.2 Quadtree-decomposition algorithm

```
function tree = quadTree(obst, bb, level)
    % Generate quad $tree$ of depth $level>=0$ around obstacles $obst$
    % (every column contains the obstacle vertices: x1, y1, x2, y2, ...) in
    % an environment of dimension $bb$ (xa, xb, ya, yb).
    minDim = [diff(bb(1:2)); diff(bb(3:4))]/2^level; % Min cell dimension
    % Root node
    tree(1).leaf = true; % % Is the cell a leaf?
    tree(1).free = false; % Is the cell occupied?
    tree(1).bounds = bb; % Cell boundary: xa, xb, ya, yb
    tree(1).center = [mean(bb(1:2)); mean(bb(3:4))]; % Cell center

    id = 1; k = 2;
    while id < k
        occupied = isOccupied(tree(id).bounds, obst);

        d = [diff(tree(id).bounds(1:2)), diff(tree(id).bounds(3:4))]/2;
        if occupied && d(1) > minDim(1)/2 % Split the cell into 4 new cells
            tree(id).leaf = false;
```

EXAMPLE 4.1—cont'd

```
            tree(id).free = false;

            b = tree(id).bounds;
            bs = [b(1),      b(1)+d(1), b(3),      b(3)+d(2); ...
                  b(1)+d(1), b(2),      b(3),      b(3)+d(2); ...
                  b(1),      b(1)+d(1), b(3)+d(2), b(4); ...
                  b(1)+d(1), b(2),      b(3)+d(2), b(4)];
            for i = 1:4
                tree(k).leaf = true;
                tree(k).free = false;
                tree(k).bounds = bs(i,:);
                tree(k).center = [mean(bs(i,1:2)); mean(bs(i,3:4))];
                k = k + 1;
            end
        elseif ~occupied
            tree(id).free = true;
        end
        id = id + 1;
    end

    % Create visibility graph
    a = zeros(2,length(tree)*4); leafs = zeros(1,length(tree));
    for i = 1:length(tree)
        a(:,i*4-3:i*4) = tree(i).bounds([1, 2, 2, 1; 3, 3, 4, 4]);
        leafs(i) = tree(i).leaf;
    end
    offset = [-1, 1, 1,-1; -1,-1, 1, 1]/2.*repmat(minDim, 1, 4);
    for i = 1:length(tree)
        tree(i).neighbours = [];
        if tree(i).leaf
            b = tree(i).bounds([1, 2, 2, 1; 3, 3, 4, 4]) + offset;
            c = reshape(inpolygon(a(1,:), a(2,:), b(1,:), b(2,:)), 4, []);
            tree(i).neighbours = setdiff(find(any(c).*leafs), i);
        end
    end
end

function occupied = isOccupied(bounds, obst)
    occupied = false;
    pb = bounds([1, 2, 2, 1, 1; 3, 3, 4, 4, 3]);
    for j = 1:size(obst, 2) % Loop through the obstacles
        pa = reshape(obst(:,j), 2, []); N = size(pa, 2);
        ina = inpolygon(pa(1,:), pa(2,:), pb(1,:), pb(2,:));
        inb = inpolygon(pb(1,:), pb(2,:), pa(1,:), pa(2,:));
        if any(ina) || any(inb) % Are any vertices in the obstacle or cell?
            occupied = true; break;
        else % Check if there are any intersecting edges.
            for k = 1:size(pb, 2)-1 % Loop through the boundary edges
                for i = 1:N % Loop through the obstacle edges
                    a1 = [pa(:,i); 1]; a2 = [pa(:,mod(i,N)+1); 1];
                    b1 = [pb(:,k); 1]; b2 = [pb(:,k+1); 1];
                    pc = cross(cross(a1, a2), cross(b1, b2));% Intersection
                    if abs(pc(3))>eps
                        pc = pc/pc(3);
                        da = a2-a1; ca = pc-a1; ea = (ca.'*da)/(da.'*da);
                        db = b2-b1; cb = pc-b1; eb = (cb.'*db)/(db.'*db);
                        if eb>eps && eb<1 && ea>eps && ea<1
```

(*Continued*)

EXAMPLE 4.1—cont'd

```
                        occupied = true; break;
                    end
                end
            end
            if occupied, break; end
        end
    end
    if occupied, break; end
  end
end
```

4.2.3 Roadmap

A roadmap (map that contains roads), which consists of lines, curves, and their points of intersection, gives possible connections between points in the free space. The task of path planning is to connect the start point and the goal point with an existing road connection in the map to find a connecting sequence of roads. A roadmap depends on environment geometry. The challenge is to find a minimum number of roads that enable a mobile robot to access any free part of the environment. In the following, two different roadmap building algorithms are presented: visibility map and Voronoi graph.

Visibility Graph

A *visibility graph* consists of all possible connections among any two vertices that lie entirely in the free space of the environment. This means that for each vertex connections are made to all the other vertices that can be seen from it. The start point and the goal point are treated as vertices. Connections are also made between neighboring vertices of the same polygon. An example of a visibility graph is given in Fig. 4.11A. The path obtained using a visibility graph tends to be as close as possible to the obstacles; therefore, it is the shortest path possible. To prevent collision of the robot with the obstacles the obstacles can be enlarged by the robot diameter as described in Section 4.1.3.

Visibility graphs are simple to use but the number of road connections increases with the number of obstacles, which can result in higher complexity and therefore lower efficiency. A visibility graph can be simplified

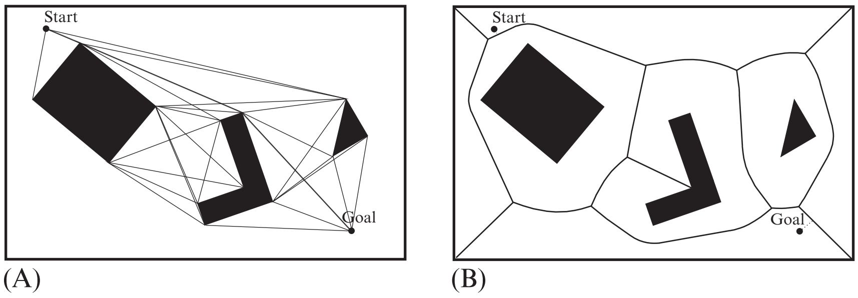

Fig. 4.11 Roadmap: (A) visibility graph and (B) Voronoi graph.

by removing redundant connections that can be replaced by existing shorter connections.

Voronoi Graph

A Voronoi graph (Fig. 4.11B) consists of road sections that have the largest distance from the obstacles. This also means that the road between any two obstacles has equal distance to both obstacles.

A basic Veronoi graph (diagram) is defined for a plane with *n* points (e.g., point obstacle). It partitions the plane into *n* regions whose borders define the roadmap. Each region has exactly one generating point and any point in a certain region is closer to its generating point than to any other generating point. An example of a general Voronoi graph for the plane environment consisting of obstacles of any shape (square, straight line, etc.) is shown in Fig. 4.11. Here, space is partitioned to regions where each region has exactly one generating obstacle. Any point in a certain region is closer to the generating obstacle than to any other obstacle. Borders between the regions define the roadmap.

Driving on such a road minimizes the risk of colliding with obstacles, which can be desired when the robot pose is known with some uncertainty (due to measurement noise or control). In environments constructed from polygons the roadmap consists of three typical *Voronoi curves* as shown in Fig. 4.12. A Voronoi curve that has equal distance between

- two vertices is a straight line,
- two edges is also a straight line, and
- a vertex and a straight line segment is a parabola.

In a roadmap the start and the goal configuration are also connected, and the obtained graph is used to find the solution.

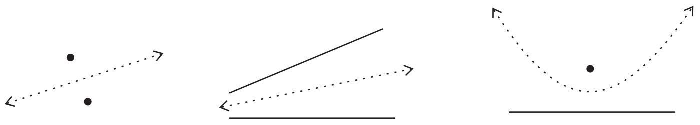
Fig. 4.12 Typical Voronoi curves.

As already mentioned this approach maximizes robot distance to the obstacles. However, the obtained path length is far from being the optimal (shortest) one. A robot with distance sensors (such as an ultrasonic or laser range finder) can apply a control to drive equally away from all the surrounding obstacles, which means that it follows roads from the Voronoi graph. Although robots using only touch or vicinity sensors cannot follow Voronoi roads because they may have problems with localization, they can easily follow roads in the visibility graph.

EXAMPLE 4.2

Compute the Voronoi graph for the environment in Fig. 4.11 using the function `voronoi` in Matlab. The environment (`o`) and object (`o1`, `o2`, `o3`) coordinates are defined in Listing 4.3.

Listing 4.3 Vertices of the obstacles in the environment

```
o  = 1000*[0.0149, 0.0693; ...
           1.6228, 0.0679; ...
           1.6241, 1.0867; ...
           0.0112, 1.0854];
o1 = 1000*[0.4263, 0.4569; ...
           0.6144, 0.6857; ...
           0.3097, 0.9414; ...
           0.1190, 0.7126];
o2 = 1000*[0.8151, 0.2079; ...
           1.0885, 0.3008; ...
           0.9644, 0.6574; ...
           0.8753, 0.6278; ...
           0.9706, 0.3573; ...
           0.7838, 0.2927];
o3 = 1000*[1.3319, 0.4865; ...
           1.4723, 0.5659; ...
           1.3845, 0.7112];
% Free space is on the right-hand side of the obstacle
obstacles = {flipud(o), o1, o2, o3};
```

Solution

In Matlab the Voronoi graph for a list of points can be drawn using a function `voronoi`. The environment in Fig. 4.11 also contains polygons with defined edges; therefore, the `voronoi` function cannot be used

EXAMPLE 4.2—cont'd

directly. But if each edge is presented by a number of equidistant (assistant) points, an approximation of the Voronoi graph can be computed (using function `voronoi`) as shown in Fig. 4.13, where 20 points are used for each obstacle edge.

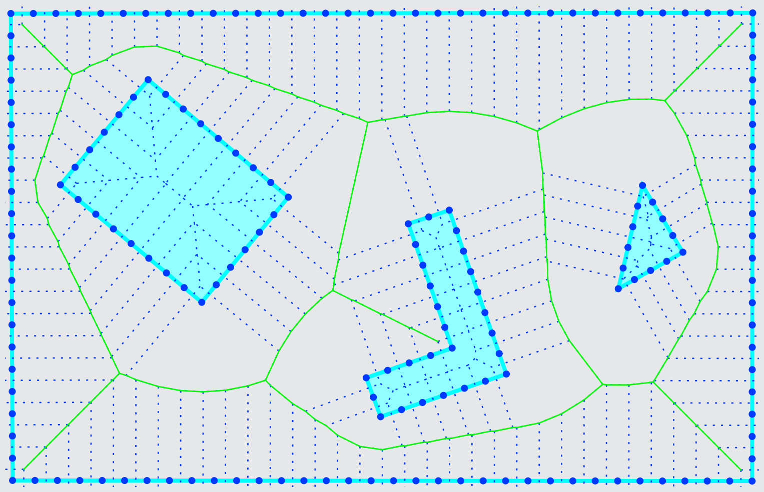

Fig. 4.13 Approximation of Voronoi graph using Matlab function `voronoi` and assistant points for the borders' presentation.

To obtain the final Voronoi graph the obtained cell borders between assistant points belonging to the same edge need to be removed; better yet, they do not need to be computed.

Listing 4.4 Approximation of Voronoi graph

```
% Break down the line segments into intermediate points
dMin = 50;
points = []; obst = [];
B = length(obstacles);
for i = 1:B
    ob = obstacles{i};
    M = size(ob, 1);
    for j = 1:M
        k = mod(j, M)+1; % j+1
        d = sqrt((ob(j,1)-ob(k,1))^2 + (ob(j,2)-ob(k,2))^2);
        n = ceil(d/dMin)+1;
        x = linspace(ob(j,1), ob(k,1), n).';
        y = linspace(ob(j,2), ob(k,2), n).';
        points = [points; x(1:end-1) y(1:end-1)];
    end
    obst = [obst; obstacles{i}([1:end,1],:); nan(1,2)];
end

% Calculate voronoi segments on a set of intermediate points
```

(*Continued*)

EXAMPLE 4.2—cont'd

```
[vx, vy] = voronoi(points(:,1), points(:,2));

% Eliminate auxilary (not in the free space) voronoi segments
s = false(1, size(vx, 2));
for j = 1:size(vx, 2)
    in = inpolygon(vx(:,j), vy(:,j), obst(:,1), obst(:,2));
    s(j) = all(in==1);
end
ux = vx(:,s); uy = vy(:,s); % Approximated Voronoi segments

% Plot
voronoi(points(:,1), points(:,2), 'b:'); hold on;
plot([ux;nan(1,size(ux,2))], [uy;nan(1,size(uy,2))], 'g-', 'LineWidth', 1);
axis equal tight;
```

Triangulation

Triangulation is a form of decomposition where the environment is split into triangular cells. Although there are various triangulation algorithms, finding a good triangulation approach that avoids narrow triangles is an open research problem [2]. One possible algorithm is the Delaunay triangulation, which is a dual presentation of the Voronoi graph. In a Delaunay graph the center of each triangle (center of the circumscribed circle) coincides with each vertex of the Voronoi polygon. An example of the latter is given in Fig. 4.14.

From accurate environment decomposition approaches, path searching algorithms can be used in order to determine if the path exists or if it does not. These approaches are therefore complete.

EXAMPLE 4.3

Obtain Delaunay triangulation for the environment given in Fig. 4.11. Points of the obstacles borders are given in Listing 4.3.

Solution

In Matlab, the Delaunay triangulation can be computed for known vertices using function `DelaunayTri` and plotted using function `triplot`, as shown in Listing 4.5.

Listing 4.5 Delaunay triangulation

```
points = cell2mat(obstacles(:));
dt = delaunayTriangulation(points);
triplot(dt, 'b-'); axis equal tight;
```

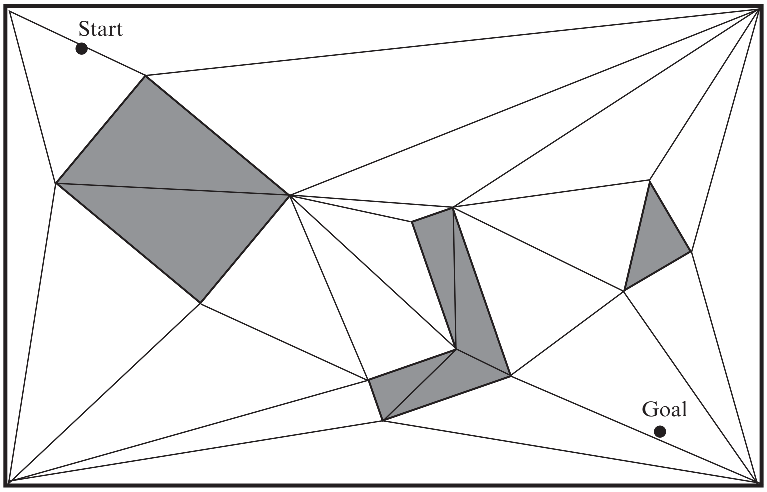

Fig. 4.14 Delaunay triangulation where some triangle edges coincide with the edges of the obstacles.

4.2.4 Potential Field

The potential field method describes the environment with the *potential field*, which can be thought of as an imaginary height. The goal point is in the bottom, and the height increases with the distance to the goal point and is even higher at the obstacles. A path planning procedure can then be explained as the motion of the ball that rolls downhill to the goal, as shown in Fig. 4.15.

A potential field is expressed as a sum of the attractive field due to the goal point $U_{\mathrm{attr}}(\boldsymbol{q})$ and a repulsive field $U_{\mathrm{rep}}(\boldsymbol{q})$ due to the obstacles:

$$U(\boldsymbol{q}) = U_{\mathrm{attr}}(\boldsymbol{q}) + U_{\mathrm{rep}}(\boldsymbol{q}) \tag{4.1}$$

The goal point is the global minimum of the potential field. Attractive potential $U_{\mathrm{attr}}(\boldsymbol{q})$ in Eq. (4.1) can be defined to be proportional to the squared Euclidean distance to the goal point $D(\boldsymbol{q}, \boldsymbol{q}_{\mathrm{goal}}) = \sqrt{(x - x_{\mathrm{goal}})^2 + (y - y_{\mathrm{goal}})^2}$, as follows:

$$U_{\mathrm{attr}}(\boldsymbol{q}) = k_{\mathrm{attr}} \frac{1}{2} D^2(\boldsymbol{q}, \boldsymbol{q}_{\mathrm{goal}}) \tag{4.2}$$

where k_{attr} is a positive constant.

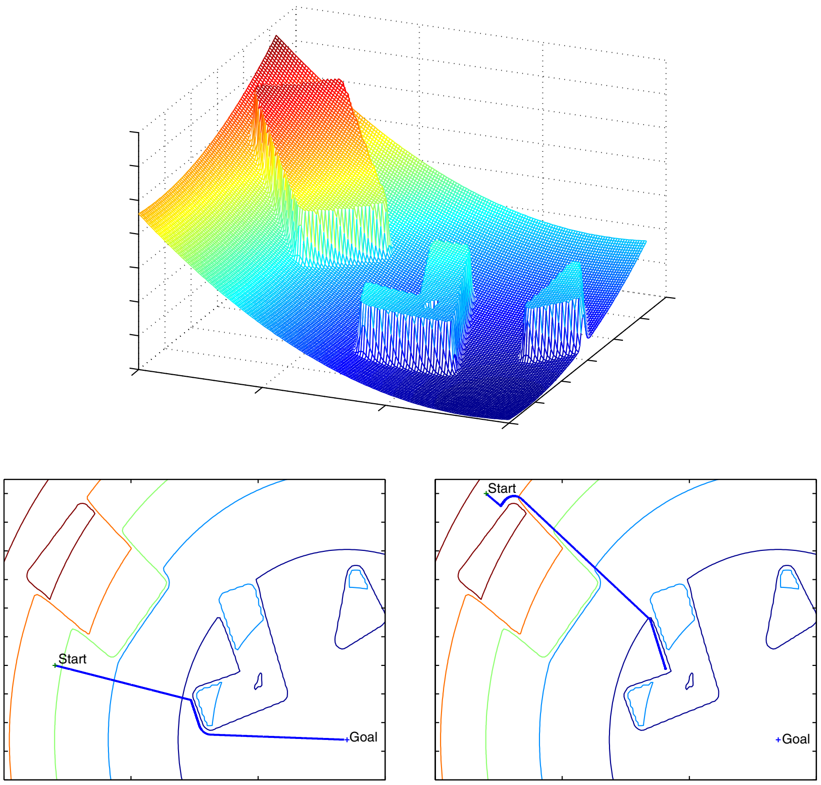

Fig. 4.15 Potential field for known goal point (*upper*) and contours with equal potential and calculated path for two start points (*lower*), where the calculated path reaches the goal (*left*) and where the path is trapped in the concave obstacle (*right*).

Repulsive potential $U_{\text{rep}}(\boldsymbol{q})$ should be very high near the obstacles and with a decreasing potential as the distance from the obstacles increases $D(\boldsymbol{q}, \boldsymbol{q}_{\text{obst}})$. It should be zero when $D(\boldsymbol{q}, \boldsymbol{q}_{\text{obst}})$ is higher than some threshold value D_0. The repulsive potential can be expressed as follows:

$$U_{\text{rep}}(\boldsymbol{q}) = \begin{cases} \frac{1}{2}k_{\text{rep}}\left(\frac{1}{D(\boldsymbol{q},\boldsymbol{q}_{\text{obst}})} - \frac{1}{D_0}\right)^2; & D(\boldsymbol{q}) \leq D_0 \\ 0; & D(\boldsymbol{q}) > D_0 \end{cases} \tag{4.3}$$

where k_{rep} is a positive constant and $D(\boldsymbol{q}, \boldsymbol{q}_{\text{obst}})$ is the distance to the nearest point on the nearest obstacle.

To obtain the path from the start to the goal point we need to follow the negative gradient of potential field ($-\nabla U(\boldsymbol{q})$).

EXAMPLE 4.4

Calculate the negative gradient of the potential field (4.1).

Solution

The negative gradient of attractive field (4.2) equals to

$$-\nabla U_{\text{attr}}(\boldsymbol{q}) = -k_{\text{attr}}\frac{1}{2}\begin{bmatrix}2(x - x_{\text{goal}})\\ 2(y - y_{\text{goal}})\end{bmatrix} = k_{\text{attr}}(\boldsymbol{q}_{\text{goal}} - \boldsymbol{q})$$

and it is in the direction from the robot pose $\boldsymbol{q}$ to the goal $\boldsymbol{q}_{\text{goal}}$ where the magnitude (norm) is proportional to the distance between the points $\boldsymbol{q}$ and $\boldsymbol{q}_{\text{goal}}$.

The negative gradient of the repulsive field (4.3) when $D_{\text{obst}} \leq D_0$ equals

$$\begin{aligned}-\nabla U_{\text{rep}}(\boldsymbol{q}) &= -k_{\text{rep}}\left(\frac{1}{D_{\text{obst}}} - \frac{1}{D_0}\right)\frac{-1}{D_{\text{obst}}^2}\nabla D_{\text{obst}}\\ &= k_{\text{rep}}\left(\frac{1}{D_{\text{obst}}} - \frac{1}{D_0}\right)\frac{1}{D_{\text{obst}}^3}(\boldsymbol{q} - \boldsymbol{q}_{\text{obst}}),\end{aligned}$$

where $D_{\text{obst}} = D(\boldsymbol{q}, \boldsymbol{q}_{\text{obst}}) = \sqrt{(x - x_{\text{obst}})^2 + (y - y_{\text{obst}})^2}$. The direction of the repulsive field is always away from the obstacle and its strength decreases with the distance to the obstacle. For situations where $D_{\text{obst}} > D_0$, the repulsive gradient is $-\nabla U_{\text{rep}}(\boldsymbol{q}) = 0$.

By using a potential field for environment presentation the robot can reach the goal point simply by following the negative gradient of the potential field. The negative gradient of the potential field is explicitly calculated from the known robot position. The main disadvantage of this approach is the possibility that the robot can become trapped (jittering behavior) in the local minimum, which can happen if the environment contains any obstacle of concave shape (Fig. 4.15) or when the robot starts to oscillate between more equally distanced points to the obstacle. The path planning using a potential field can be used to calculate the reference path that the robot needs to follow, or it can also be used online in motion control to steer the robot in the current negative gradient direction.

EXAMPLE 4.5

Calculate the potential field shown in Fig. 4.15 for the environment with obstacle coordinates given in Listing 4.3.

Solution

The potential field is calculated using relations (4.1)–(4.3), while the path is calculated by integration of the negative gradient, calculated in Example 4.4.

4.2.5 Sampling-Based Path-Planning

Up to now the presented path planning methods required explicit presentation of free environment configuration space. By increasing the dimension of the configuration space these methods tend to become time consuming. In such cases, sampling-based methods could be used for environment presentation and path planning.

In sampling-based path planning, random points (robot configurations) are sampled; then collision detection methods are applied in order to verify if these points belong to the free space [4, 5]. From a set of such sampled points and connections among them (connections must also lie in the free space) a path between the known start and the desired goal point is searched.

Sampling-based methods do not require calculation of free configuration space Q_{free}, which is a time-consuming operation for complex obstacle shapes and for many DOF. Instead, the random samples are used to present the configuration space, and independently from the environment geometry a path planning solution can be obtained for a wide variety of problems. In comparison to decomposition of the environment to cells, the sampling-based approaches do not require a high number of cells and time-consuming computation that is normally required to achieve accurate decomposition to cells. Due to inclusion of stochastic mechanisms (random walk) such as in random path planner the problem of being trapped in some local minimum (e.g., as in the potential field method) is unlikely, because motion is defined between randomly chosen samples from the free space.

To save processing time the collision detection is checked only for obstacles that are close enough and present potential danger of colliding with the robot. Additionally, the robot and obstacles can be encircled by

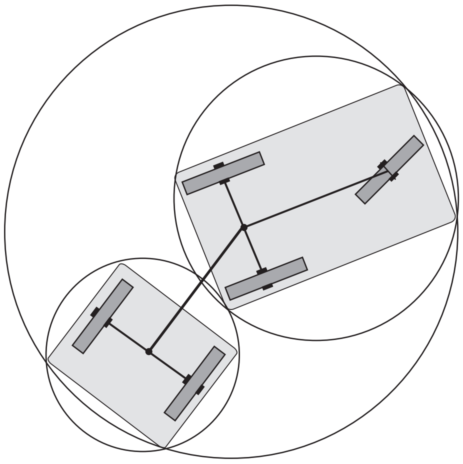

Fig. 4.16 Presenting a complex robot shape with an enclosing simple shape (*circle*) that can be split into two smaller simple shapes while performing hierarchical detection of collision of the robot.

simple shapes so that complex collision detection (between true robot shape and obstacle shapes) is performed only when those enclosing shapes are overlapping. Furthermore, collision detection can be performed hierarchically by exchanging bigger shapes that encircle the whole robot for two smaller shapes that encircle two halves of the robot, as shown in Fig. 4.16. If any of the two shapes are overlapping with the obstacle it is again split in two appropriate smaller shapes. This is continued until the collision is confirmed or rejected or until the desired resolution is reached.

Sampling-based approaches can be divided into the ones that are appropriate for a single path search and those that can be used for many path search queries. In the former approaches the path from a single start to a single goal point needs to be found as quickly as possible; therefore, the algorithms normally focus only on parts of the environment that are more promising to find the solution. New points and connections to the graph are included at runtime until the solution is found. In the latter approaches, the entire procedure of presenting the whole free environment space by an undirected graph or roadmap is performed first. The path planning solution can then be found in the obtained graph for an arbitrary pair of start and goal points. In the following, an example of both approaches is given.

RRT Method

A rapidly exploring random tree (RRT) is the method for searching the path from a certain known start point to one goal point [4]. In each iteration, the method adds a new connection in the direction from the randomly sampled point to the closest point in the existing graph.

In the first iteration of the algorithm the starting configuration q_i presents the graph tree. In each next iteration a configuration q_{rand} is chosen randomly and the closest node q_{near} from the existing graph is searched for. In the direction from q_{near} to q_{rand} at some predefined distance ε a candidate for a new node q_{new} is calculated. If q_{new} and the connection from q_{near} to q_{new} are in the free space then q_{new} is selected as the new node and its connection to q_{near} is added to the graph. The procedure is illustrated in Fig. 4.17.

The search is complete after some number of iterations are evaluated (e.g., 100 iterations) or when some probability is reached (e.g., 10%). Once the algorithm termination criterion is satisfied, the goal point is selected instead of taking a new random sample, and a check is made to determine whether the goal point can be connected to the graph [6]. Such a graph tree is quickly extending to unexplored areas as shown in Fig. 4.18. This method has only two parameters: the size of the step ε and the desired resolution or number of iterations that define the algorithm termination conditions. Therefore, the behavior of the RRT algorithm is consistent and its analysis is simple.

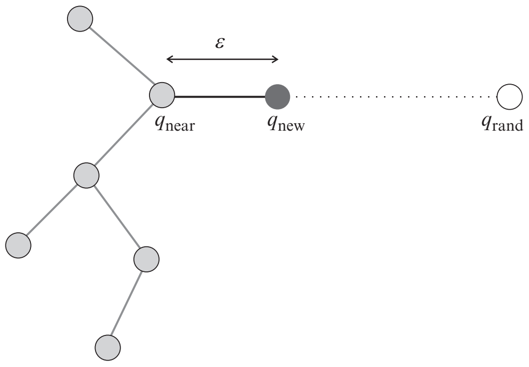

Fig. 4.17 Illustration of RRT method. Extension of the graph with a new point q_{new} in the direction of randomly sampled point q_{rand}.

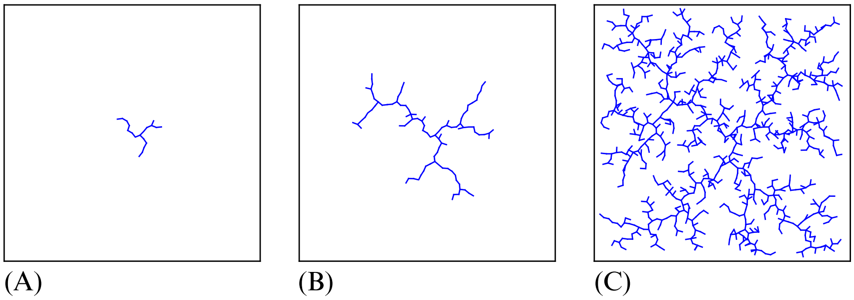

Fig. 4.18 The tree built by RRT method quickly progress to the unexplored free space. From the left to the right figure the trees contain (A) 20, (B) 100, and (C) 1000 nodes, respectively.

EXAMPLE 4.6

Implement an RRT algorithm that will produce a graph tree for a 2D free space of size 10 by 10 squared meters where the parameter $\varepsilon = 0.2$ m.

Solution

A Matlab script that produces results similar as in Fig. 4.18 could be implemented as shown in Listing 4.6.

Listing 4.6 RRT algorithm

```
xi = [5, 5]; % Initial configuration
D = 0.2; % Distance to new node
maxIter = 1000;
M = [xi]; % Map

j = 1;
while j < maxIter
    xRand = 10*rand(1,2); % Random configuration
    dMin = 100; iMin = 1; % Search for the closest point in the map M
    for i = 1:size(M,1)
        d = norm(M(i,:)-xRand);
        if d<dMin
            dMin = d;
            iMin = i;
        end
    end

    xNear = M(iMin,:);
    v = xRand - xNear;
    xNew = xNear + v/norm(v)*D; % Calculate new point

    con = [xNear; xNew];
    M = [M; xNew];
    j = j + 1;

    line(con(:,1), con(:,2), 'Color', 'b');
end
```

EXAMPLE 4.7

Extend Example 4.6 to also include simple obstacles (e.g., circular-shaped obstacles).

Solution

Assume there is an environment with simple obstacles, such as circular objects with known positions and diameters. Check if the candidate for a new node q_{new} is in the free area and if the straight line segment connecting q_{near} and q_{new} also lies in the free space.

PRM Method

A probabilistic roadmap (PRM) is the method used for path searching between more start points and more goal points [7]. The algorithm has two steps. The first step is the learning phase where a roadmap or undirected graph of the free space is constructed as shown in Fig. 4.19A. In the second step the current start and goal point are connected to the graph and some graph path searching algorithm is used to find the optimum path, as shown in Fig. 4.19B.

In the learning phase a roadmap is constructed. The roadmap is initially an empty set and it is later filled with nodes by repeating the following steps. A randomly chosen configuration q_{rand} from the free space is included in the map, and nodes Q_n that are needed to expand the map are found. These nodes can be found by choosing K nearest neighbor nodes (Q_n) or all neighbor nodes whose distance to q_{rand} is smaller than some predefined parameter D, as shown in Fig. 4.20. Note that in the first iteration or at

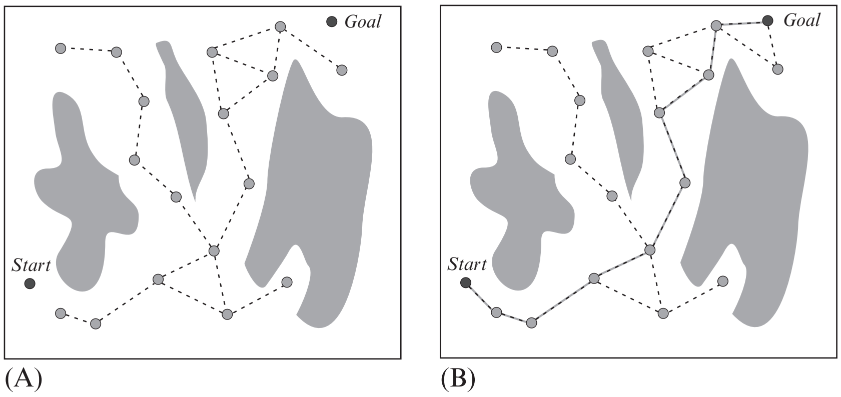

Fig. 4.19 PRM method: (A) learning phase and (B) path searching phase.

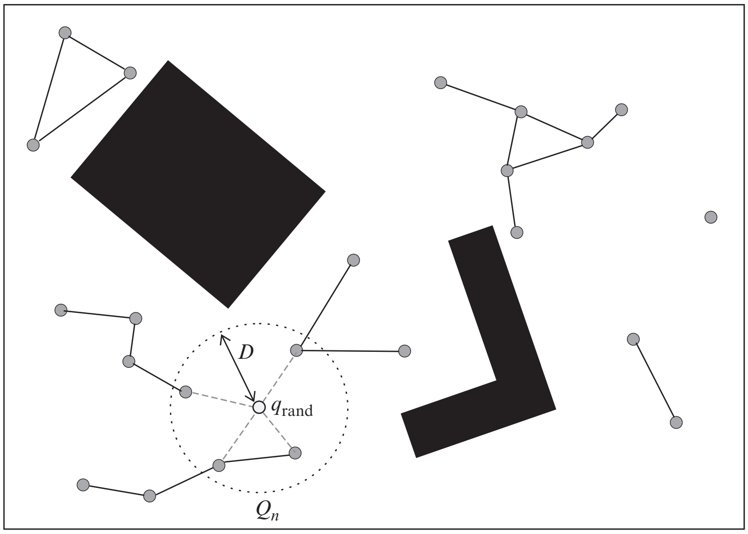

Fig. 4.20 In PRM method all possible connections from the random point q_{rand} to neighbor nodes Q_n are included in the map.

first few iterations neighbor nodes probably cannot be found. All simple connections from q_{rand} to the nodes Q_n that entirely lie in the free space are included in the map. This procedure is repeated until the map contains the desired number of N nodes.

In the path searching phase the desired start and the goal point are connected through free space with the closest possible nodes from the map. Then a path searching algorithm is applied to find the path from the start to the goal point.

These two phases do not need to be performed separately, as they can be repeated iteratively until the current number of nodes is high enough to find the solution. If the solution cannot be found, the map is further expanded with new nodes and connections until the solution is feasible. In this way we iteratively approach toward the most appropriate presentation of the free space.

This method is very effective for robots with many DOF; however, there can be problems finding the path between two regions that are connected with a narrow passage. This problem can be solved by adding additional nodes using the *bridge test*, where three nearby random points on some straight line is selected, as shown in Fig. 4.21. If the outermost points are in collision with the obstacle and the middle one is not, then the latter is included as a node in the map. We try to connect this node with neighbor

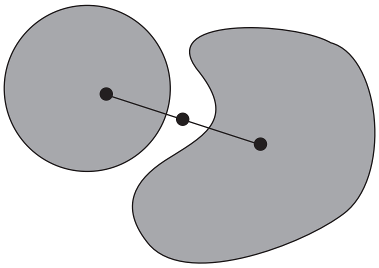

Fig. 4.21 In the bridge test two random nearby points that define a line are selected. The middle point is added as a node to the roadmap if it is in the free space and the outer points are inside obstacles.

nodes in a similar way as other connections and add these connections to the map. By combining a bridge test with uniform sampling into some hybrid sampling strategy [8], smaller roadmaps that cover free space efficiently and still have good connectively in narrow passages can be obtained.

EXAMPLE 4.8

Implement a PRM algorithm for 2D free space of size 10 by 10 squared meters. Try to extend the obtained algorithm for an environment with obstacles.

Solution

A solution is given in Listing 4.7.

Listing 4.7 PRM algorithm

```
D = 1; % Distance parameter
maxIter = 200;
M = []; % Map

j = 1;
while j <= maxIter
    xRand = 10*rand(1,2); % Random configuration
    M = [M; xRand];
    con = []; % Connections
    for i = 1:size(M,1) % Search connections to the neighbor nodes
        d = norm(M(i,:)-xRand);
        if d<D && d>eps % Add connections from xRand to neighbor
            con = [con; xRand, M(i,:)];
        end
    end
    j = j + 1;

    line(xRand(1), xRand(2), 'Color', 'r', 'Marker', '.');
    for i = 1:size(con,1)
        line(con(i,[1,3]), con(i,[2,4]), 'Color', 'b');
    end
end
```

4.3 SIMPLE PATH PLANNING ALGORITHMS: BUG ALGORITHMS

Bug algorithms are the simplest path planning algorithms that assume only local knowledge and do not need a map of the environment. Therefore, they are appropriate in situations where an environment map is unknown or it is changing rapidly and also when the mobile platform has very limited computational power. These algorithms use local information obtained from their sensors (e.g., distance sensor) and global goal information. Their operation consists of two simple behaviors: motion in a straight line toward the goal and obstacle boundary following.

Mobile robots that use these algorithms can avoid obstacles and move toward the goal. These algorithms require low memory usage and the obtained path is usually far from being optimal. Bug algorithms were first implemented in [9] and several improvements followed such as in [10–12].

In the following three basic versions bug algorithms are described.

4.3.1 Bug0 Algorithm

A Bug0 algorithm has two basic behaviors:

1. Move toward the goal until an obstacle is detected or the goal is reached.
2. If an obstacle is detected, then turn left (or right, but always in the same direction) and follow the contour of the obstacle until motion in a straight line toward the goal is again possible.

A example of Bug0 algorithm performance is given in Fig. 4.22.

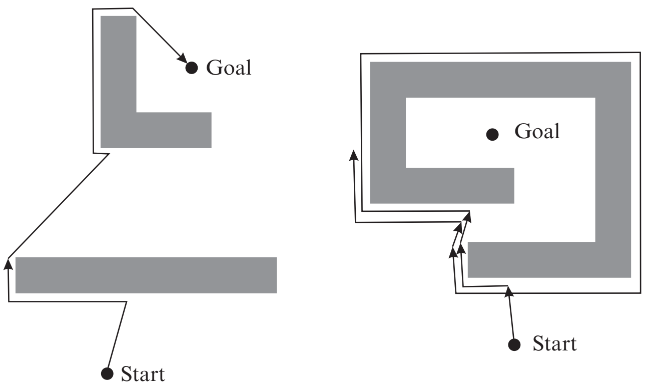

Fig. 4.22 Bug0 algorithm successfully finds a path to the goal in the environment on the *left* while it is unsuccessful in the environment on the *right*.

4.3.2 Bug1 Algorithm

A Bug1 algorithm in comparison to Bug0 uses more memory and requires several more computations. In each iteration it needs to calculate Euclidean distance to the goal and remember the closest point on the obstacle circumference to the goal. Its operation is described by the following:

1. Move on the straight line toward the goal until an obstacle is hit or the goal is reached.
2. If an obstacle is detected, then turn left and follow the entire contour of the obstacle and measure the Euclidean distance to the goal. When the point where the obstacle was initially detected is reached again, follow the contour of the obstacle in the direction that is shortest to the point on the contour that is closest to the goal. Then resume moving toward the goal in a straight line.

A example of a Bug1 algorithm operation is given in Fig. 4.23.

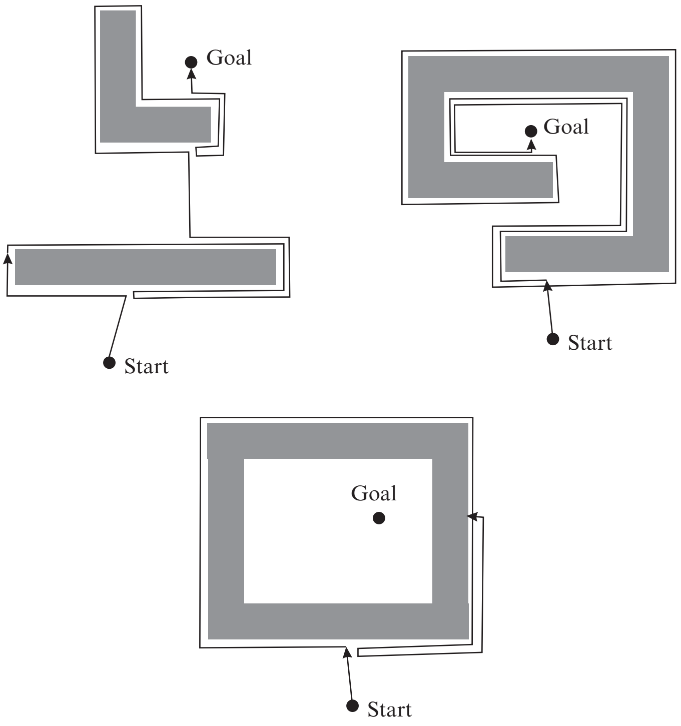

Fig. 4.23 Bug1 algorithm finds the path to the goal if it exists as shown in the *upper two figures*. In the worst case its path is for $\frac{3}{2}$ of the length of all obstacle contours longer than the Euclidean distance from the start to the goal configuration. The algorithm can identify unreachable goal.

The obtained path is not optimal and is in the worst case for $\frac{3}{2}$ of the length of all obstacle contours longer than the Euclidean distance from the start to the goal configuration. For each obstacle that is detected on its path from the start to the goal, the algorithm finds only one entry point and one leaving point from the obstacle contour. So it never detects the obstacle more than once and therefore never circles among the same obstacles. When the algorithm detects the same obstacle more than once it knows that either the start or the goal point is captured inside the obstacle and the path searching can be terminated, since no feasible path to the goal exists (bottom situation in Fig. 4.23).

4.3.3 Bug2 Algorithm

The Bug2 algorithm always tries to move on the main line that is defined as a straight line connecting the start point and the goal point. It operates by repeating the following steps:

1. Move on the main line until an obstacle is hit or the goal point is reached. In the latter case the path searching is finished.
2. If an obstacle is detected, follow the obstacle contour until the main line is reached where the Euclidean distance to the goal point is shorter than the Euclidean distance from the point where the obstacle was first detected.

Although the Bug2 algorithm in general seems much more effective than Bug1 (see left part of Fig. 4.24), it does not guarantee that the robot detects certain obstacles only once. At some obstacle configurations the robot with Bug2 could make unnecessary repeating obstacle encirclements

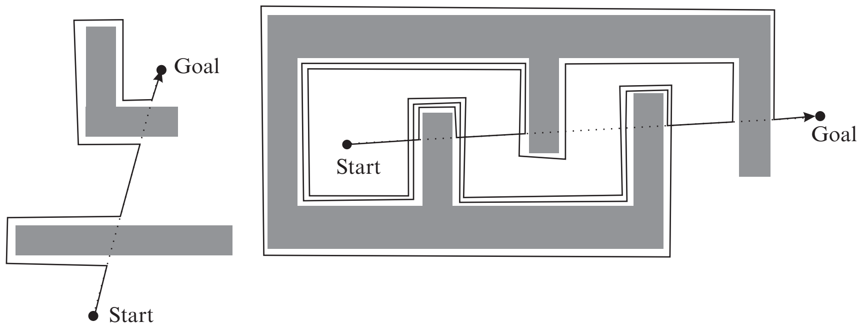

Fig. 4.24 The Bug2 algorithm follows the main line to the goal point. It can encircle the same obstacle more than once; therefore, unnecessary circling can occur. The Bug2 algorithm can identify an unreachable goal.

until it reaches the goal point as illustrated in the right part of Fig. 4.24. The algorithm can identify that the goal is unreachable if it comes across (detects) the same obstacle in the same point more than once.

From comparison of Bug1 and Bug2 algorithms the following can be concluded:

- Bug1 is the more thorough search algorithm, because it evaluates all the possibilities before making a decision.
- Bug2 is the greedy algorithm, because it selects the first option that looks promising.
- In most cases the Bug2 algorithm is more efficient than Bug1.
- However, the operation of the Bug1 algorithm is easier to predict.

EXAMPLE 4.9

Implement path planning using Bug0 algorithm for a mobile robot with differential drive. Assume that the environment map is not available and the robot only knows its current pose, goal point, and current distance to the goal (measurement from sensor).

According to the Bug0 algorithm the robot should drive toward the goal if it is far enough from any obstacle (e.g., more than 0.2 m), and it should follow the obstacle if it is close to the obstacle. Code your implementation of the algorithm in the simulation framework given in Listing 4.8, which already enables simulation of robot motion and sensor measurements. The environment and an example of an obtained robot path are shown in Fig. 4.25.

Listing 4.8 Simulation framework for Bug0 algorithm

```
Ts = 0.03; % Sampling time
t = 0:Ts:30; % Simulation1 time
q = [0; 0; 0];% Initial pose
goal = [4; 4]; % Goal location
% Obstacles
obstacles{1} = flipud([-1 -1; 7 -1; 7 5; -1 5]);
obstacles{2} = [0.5 1; 4 1];
obstacles{3} = [3 3.5; 3 2.5; 5 2.5; 3 2.5];
obst = [];
for i = 1:length(obstacles)
    obst = [obst; obstacles{i}([1:end,1],:); nan(1,2)];
end

for k = 1:length(t)
    % Distance to the nearest obstacle and orientation of the segment
    [dObst, ~, z] = nearestSegment(q(1:2).', obst);
    phiObst = atan2(obst(z+1,2)-obst(z,2), obst(z+1,1)-obst(z,1));

    % Control algorithm goes here ...

    % Robot motion simulation
    dq = [v*cos(q(3)); v*sin(q(3)); w];
```

EXAMPLE 4.9—cont'd

```
    noise = 0.00; % Set to experiment with noise (e.g. 0.001)
    q = q + Ts*dq + randn(3,1)*noise; % Euler integration
    q(3) = wrapToPi(q(3)); % Map orientation angle to [-pi, pi]
end
```

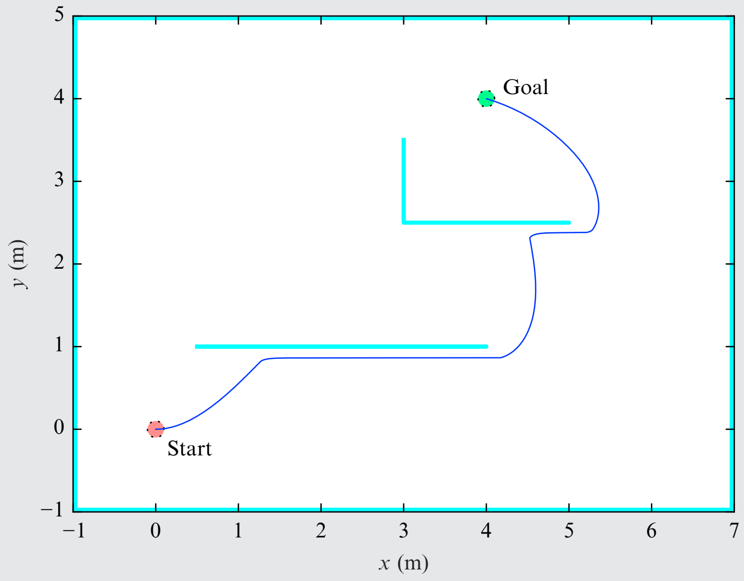

Fig. 4.25 Path planning and control example using the Bug0 algorithm to drive the robot to the goal while avoiding obstacles.

Solution

A possible implementation of the solution that produces the path shown in Fig. 4.25 is given in Listing 4.9; the code needs to be inserted into the marked line in Listing 4.8.

Listing 4.9 Bug0 algorithm

```
% Control based on distance to the obstacle
if dObst>0.2 % Drive toward the goal
    phiRef = atan2(goal(2)-q(2), goal(1)-q(1));
    ePhi = wrapToPi(phiRef - q(3));
    dGoal = sqrt(sum((goal-q(1:2)).^2));
    g = [dGoal/2, 1]; % Control gains
else % Drive right arround the obstacle
    phiRef = wrapToPi(phiObst + pi*0);  % Add pi to go always left
    ePhi = wrapToPi(phiRef-q(3));
    g = [0.4, 5]; % Control gains
end
% Simple control of differential drive
v = g(1)*abs(cos(ePhi));
w = g(2)*ePhi;
v = min([v, 0.5]);
```

EXAMPLE 4.10

Also implement Bug1 and Bug2 algorithms by extending Example 4.9.

Solution

To implement the Bug1 algorithm you can adapt Listing 4.9 in Example 4.25 where the main behavior consists of two parts. The first behavior (driving toward the goal) remains unchanged, while the second needs to be changed as follows. Store the start position where the robot first detects the obstacle. Drive around the obstacle and measure the distance to the goal and remember the closest point. Perform this until the robot arrives in the stored start position or sufficiently close. Return to the remembered closest point.

Similarly you can adapt Listing 4.9 in Example 4.25 to the Bug2 algorithm as follows. Store the start position where the robot first detects the obstacle. Do the obstacle encircling until the main straight line is crossed. If this cross point is closer to the goal than the starting point then drive toward the goal; otherwise, continue encircling the obstacle.

4.4 GRAPH-BASED PATH PLANNING METHODS

If the environment with obstacles is sufficiently presented by a graph, one can use path searching algorithms to find the path from the start to goal configuration. In the following, a few well-known path graph-based searching algorithms are presented.

In general, path searching algorithms proceed as follows. The starting node is first checked to determine if it is also the goal node. Normally this is not the case; therefore, the search is extended to other nodes in the neighborhood of the current node. One of the neighbor nodes is selected (how the nodes are selected depends on the used algorithm and its cost function), and if it is not the goal node, then the search is also extended to the neighbor nodes of this new node. This procedure is continued until the solution is found or until all graph nodes have been investigated.

When performing a search in a graph, a list of nodes that have already been visited is made with the main purpose to prevent visiting the same node more than once. Nodes that can still extend the search area to other neighbor nodes (also known as alive nodes) are kept in an *open list*. Nodes that have no successors or have been checked already (also known as dead nodes) are kept in a *closed list*.

The evolution of path searching depends on the strategy of selecting new nodes for extending the search area. A list of open nodes is sorted according to some criteria and when the search is extended the first node from that sorted list is taken; this node best fits the used sorting criteria (has the smallest value of the criteria).

In the beginning, the open list Q only contains the start node. Nodes in the neighborhood of the start node are calculated and put in the open list, and the starting node is put in the closed list. The search is then extended with the first node in the open list and successors (of the chosen node from the open list) are calculated. The open list then contains the remaining previous successors (except the first one that has been already extended) and currently calculated successors of the chosen node. This procedure is shown in Fig. 4.26, where nodes in the open list are colored with light gray, nodes in the closed list are colored black and unvisited nodes are white. In Fig. 4.26 it can be seen why the nodes from the open list are called leaf nodes.

Graph search algorithms can be classified as *informed* or *noninformed*. Noninformed algorithms use no additional information beside the problem definition. These approaches search the graph systematically and do not distinguish more promising nodes from the less promising ones. Informed search, also known as heuristic search, contains additional information about the nodes. Therefore, it is possible to select more promising nodes and in that way searching for the final solution can be more efficient.

4.4.1 Breadth-First Search

Breadth-first search belongs to a class of noninformed graph search algorithms. It first explores the shallow nodes, that is, the nodes that are closest to the starting node. All nodes that can be accessed in k steps are visited before any of the nodes that require $k + 1$ steps (Fig. 4.26).

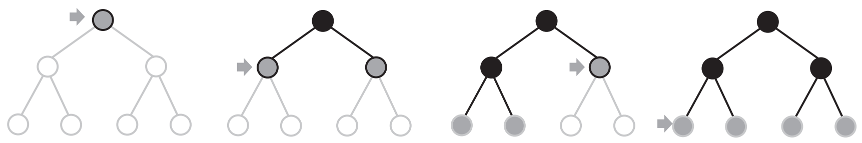

Fig. 4.26 Breadth-first search algorithm first explores the closest nodes. The currently explored node is marked with an *arrow*, nodes in open list are *colored gray*, nodes in the closed list are *black*, and unvisited nodes are *white*.

In the open list Q, the nodes are sorted using a FIFO approach (first-in first-out). The newly opened nodes are added to the end of the list Q and the nodes for extending the search are taken from the beginning of the list Q.

The algorithm is *complete* because it finds solution if it exists. If there exist several solutions, it finds the one that has the least steps from the starting node. In general, the found solution is not optimal because it is not necessary that all transitions between nodes have equal cost.

The algorithm generally requires high memory usage and long computing times. Computational time and memory consumption both increase exponentially as graph branching progresses.

4.4.2 Depth-First Search

Depth-first search is a noninformed graph search algorithm where the search is first extended in the depth. The node that is the farthest away from the starting node is used to extend the search. The search continues in the depth until the current node has no further successors. The search is then continued with the next deepest node whose successors have not been explored yet, as shown in Fig. 4.27.

The list of open nodes Q is a stack sorted by the LIFO (last-in first-out) method. The newly opened nodes are added to the beginning of the list Q and from the same end; also, the nodes for extending the search area are taken.

The depth-first search is not complete. In the case of an infinite graph (with infinite branch that does not end), it can get trapped in one branch of the graph; or in the case of a loop branch (loop in finite graph depth), it can get stuck in cycles. To avoid this problem the search can be limited to a certain depth only, but then the solution can have a higher depth than the maximum depth limitation. The algorithm is also not optimal because the found path is not necessarily the shortest one also.

This method has low memory usage as it only stores the path from the start node to the current node and intermediate nodes that have not been explored yet. When some nodes and all of their successors are explored, this node no longer needs to be stored in the memory.

4.4.3 Iterative Deepening Depth-First Search

This algorithm combines advantages of the breadth-first search and depth-first search algorithms. It iteratively increases the search depth limit and explores nodes using the depth-first search algorithm until the solution is

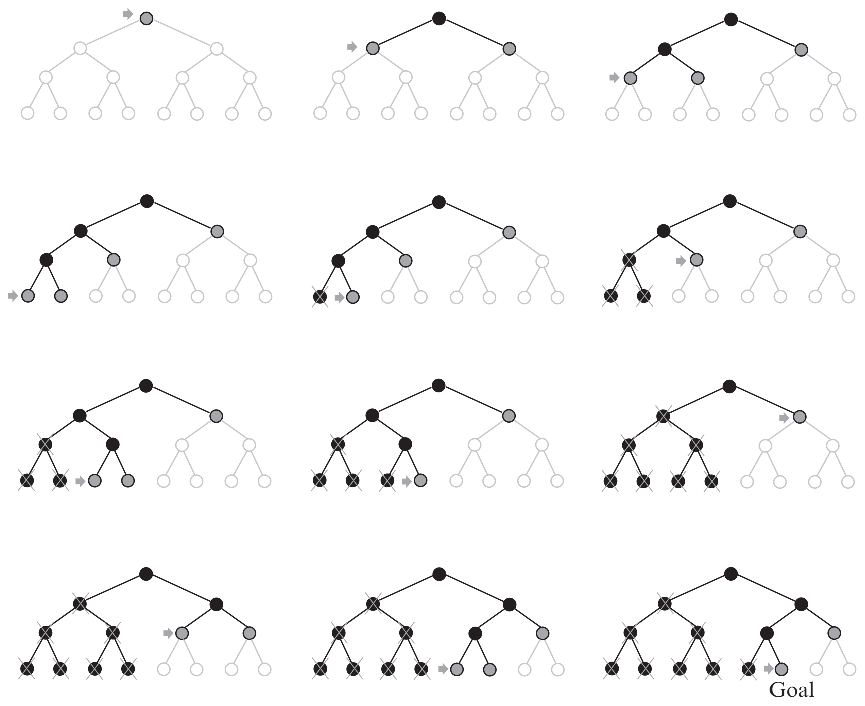

Fig. 4.27 Depth-first search has low memory usage because it stores only leaf nodes (*light gray*) and extended nodes (*gray*). Nodes removed from the memory are marked with *crosses*.

found (Fig. 4.28). First, the depth-first search is performed for the nodes that are zero steps away from the starting node. If the solution is not found, the search in the depth is repeated for nodes that are up to one step away from the starting one and so on.

This algorithm is complete (the solution is found if it exists), and it has small memory usage and is optimal if the cost of all transitions are equal or if transition costs increase with the node depth. If all the nodes have approximately the same rate of branching, then the repeated calculation of nodes is also not a big computational burden because the majority of the nodes are in the bottom of the tree, and those nodes are visited only once or a few times.

4.4.4 Dijkstra's Algorithm

Dijkstra's algorithm is a noninformed algorithm that finds the shortest paths from one node (source node) to all the other nodes in the graph [13].

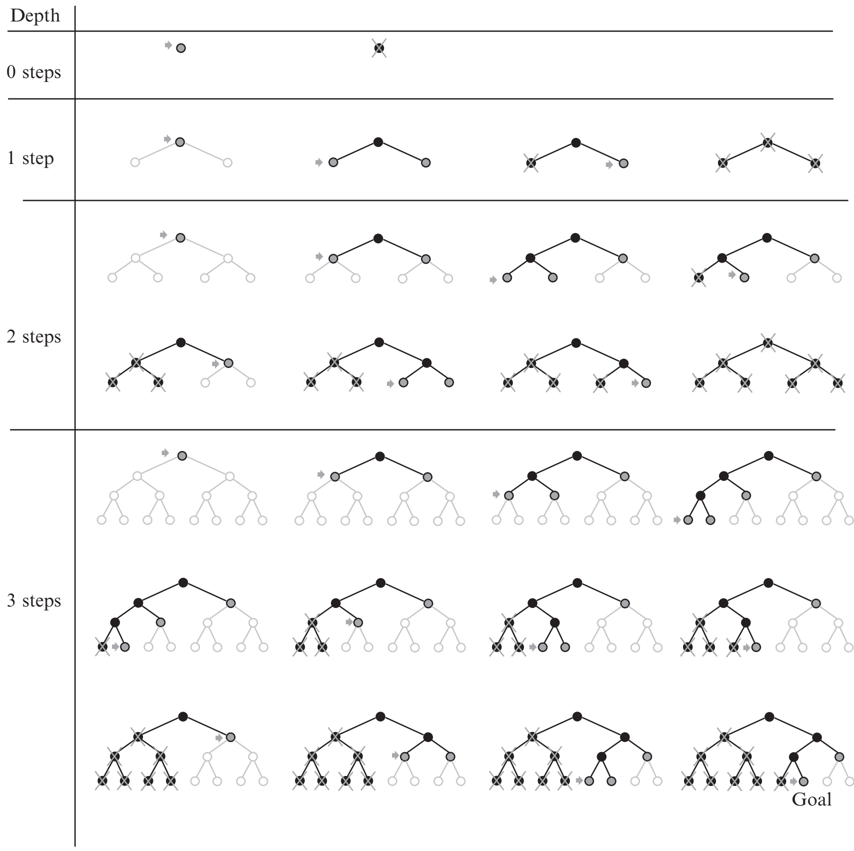

Fig. 4.28 Demonstration of iterative deepening depth-first search algorithm, where search is iterated to the depth of tree steps. Nodes removed from the memory are marked with *crosses*. The algorithm terminates when the goal node in the third step is found.

The result is therefore the shortest path tree from the source node to any other node. Originally it was proposed by Dijkstra [14] and later extended with many modifications. In problems where the shortest path between one starting node and one goal node must be found Dijkstra's algorithm may seem inefficient due to calculation of optimum paths to all the other nodes. In this case the algorithm can be terminated as soon the shortest desired path is calculated.

The algorithm finds the shortest path between the starting (source) node and the goal node because it calculates the cost function for the path between the start node to the current node, which we name cost-to-here.

The cost of the path to the current node (cost-to-here) is calculated as the sum of the path cost to the previous node (from which we came to the current node) and the cost of the transition between the previous node and the current node. In the case of several shortest paths (paths with the same cost) the algorithm returns only one, and it is not important which one.

To run the algorithm the transitions between the nodes and their costs need to be defined. For each visited node the algorithm stores the cost of currently shortest path to it (cost-to-here) and the connection from where we arrived to the current node. During the search, the lists of open nodes and closed nodes are adapted.

The operation of the algorithm is as follows. At the beginning, the open list contains only the start node, which has zero cost-to-here and it is without a connection to a previous node. The list of closed nodes is empty. Then the following steps (also shown in Fig. 4.29) are performed.

1. From the open list take the first node; call it the current node. The open list is sorted according to the cost-to-here in increasing order, where the first node is the one with the smallest cost-to-here.
2. For all nodes that can be reached from the current node calculate the cost-to-here as a sum of the current node's cost-to-here and transition cost.
3. Calculate and store cost-to-here and appropriate connection to the current node for all the nodes that do not already have these information stored.
4. If in the previous step some nodes already have cost-to-here and appropriate connection from some previous iteration, then compare both costs-to-here and keep in storage the lower cost-to-here and its connection.
5. Nodes are added to the open list and sorted in increasing order of the costs-to-here. Such a list is called a priority queue and enables that the node with the lowest price-to-here is found faster than in an unsorted list. The current node is moved to the closed list.

Originally, Dijkstra's algorithm terminates when the open nodes list is empty and the result are the shortest paths from the starting node to all the other nodes. If only the shortest path from the starting node and some goal node is needed then the algorithm terminates when the goal node is added to the closed list.

The resulting path can be obtained by back-tracking the connections that brought us to the goal node. At the goal node the connection to

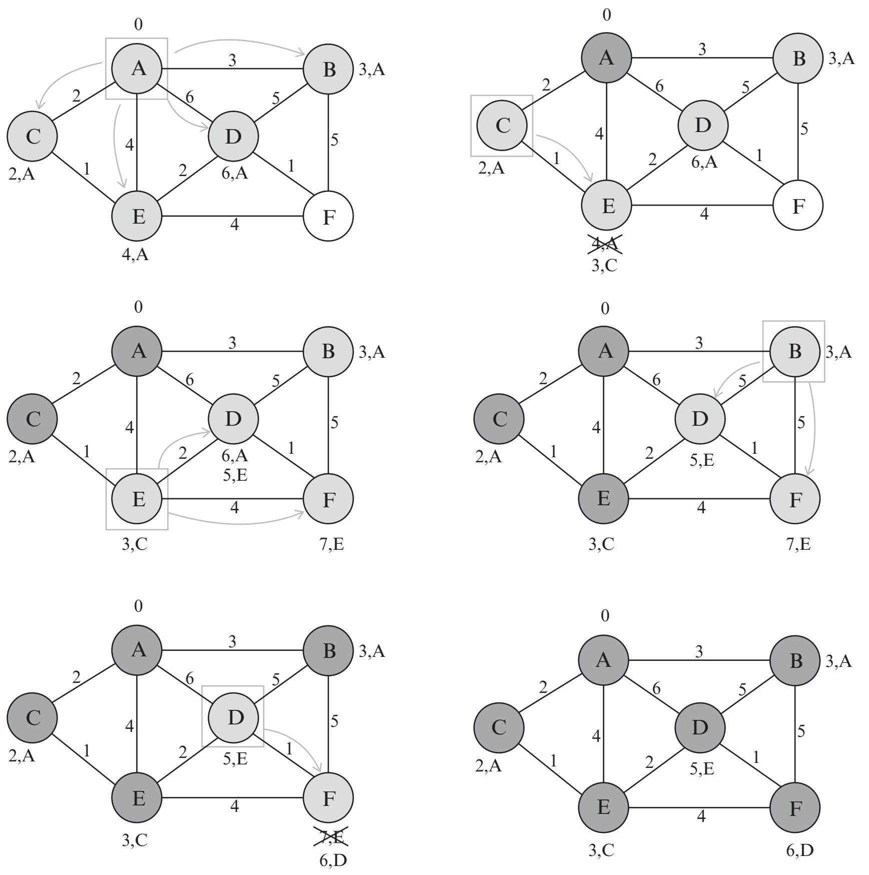

Fig. 4.29 Dijkstra's algorithm for searching the shortest paths from the starting node A to all the other nodes. The current node is surrounded with a *gray square* and its successors are marked with *arrows*. Costs for transitions between the nodes are marked at connections. Along nodes the cost-to-here and connection to the previous node are given. Nodes in the open list are colored with *light gray* while the nodes in the closed list are colored with *dark gray*. Example: If we need to obtain the shortest path between the nodes A and F its cost reads $Cost_{F-D-E-C-A} = 6$, while the path goes through nodes $A \rightarrow C \rightarrow E \rightarrow D \rightarrow F$.

the previous node is obtained then the connection to its previous node is read and so on until the start node is reached. Finally, the obtained list of connections is inverted.

Dijkstra's algorithm is complete and optimal algorithm if all the connections have costs higher than zero.

4.4.5 A* Algorithm

A* (A star) is an informed algorithm because it includes additional information or heuristic. Heuristic is the cost estimate of the path from the current node to the goal that is for the part of the graph tree that has not been explored yet. This enables the algorithm to distinguish between more or less promising nodes, and consequentially it can find the solution more efficiently. For each node the algorithm computes the heuristic function that is the cost estimate for the path from this node to the goal, and it is called cost-to-goal. This heuristic function can be Euclidean distance or Manhattan distance (sum of vertical and horizontal moves) from the current node to the goal node. The heuristic can be computed also by some other appropriate function.

During algorithm execution for each node the cost of the whole path (cost-of-the-whole-path) is calculated to consist of cost-to-here and cost-to-goal. During the path search, the open nodes list and closed nodes list are also used.

The algorithm operation is as follows. In the beginning, the open list contains only the starting node, which has zero cost-to-here and is without a connection to a previous node. Then the following steps (also illustrated in Fig. 4.30) are repeated.

1. From the open list the first node is taken; it is called the current node. The open list is sorted according to the cost-of-the-whole-path in increasing order, where the first node is the one with the smallest cost-of-the-whole-path.
2. For all nodes that can be reached from the current node we calculate the following:
   - cost-to-goal,
   - cost-to-here as the sum of the cost-to-here of the current node and the transition cost, and
   - cost-of-the-whole-path as the sum of cost-to-here and cost-to-goal.
3. For each of those nodes that do not already have stored cost-to-here, the values of the connection from the current node, cost-to-goal, and cost-of-the-whole-path are stored.
4. If in the previous step some node already has stored the cost values from some previous iteration, then both costs-to-here (current and the previous one) are compared and the one with a lower value is stored, alongside with the corresponding connection and cost-of-the-whole-path.

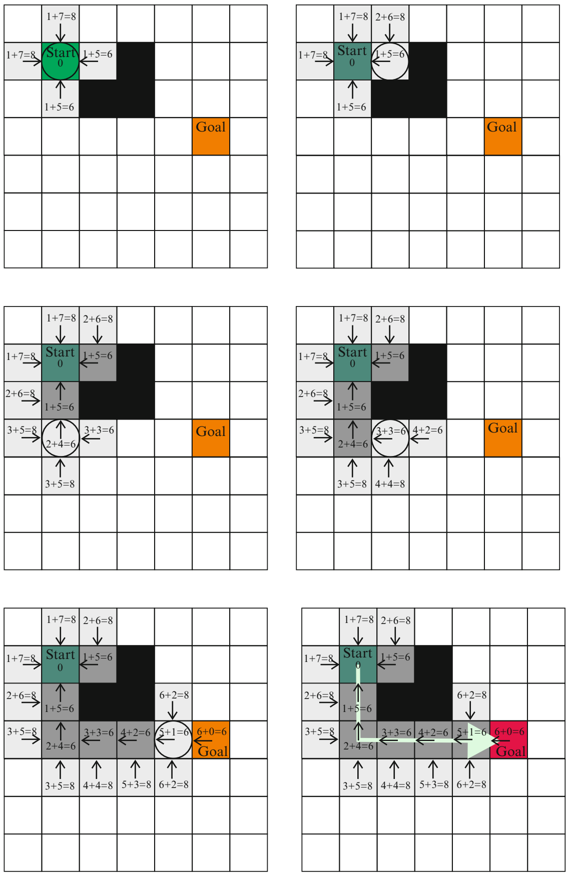

Fig. 4.30 The start and goal nodes are labeled. The current node is marked with a *circle*, nodes in the open list are *light gray*, and nodes in the closed list are *dark gray*, while obstacles are *black*. Transitions are possible in four directions (left, right, up, and down). In each visited cell (node) the direction to the parent node is denoted with an *arrow*. Each visited cell contains the cost of the path, which is the sum of cost-to-here and cost-to-goal. For the cost function the Manhattan distance is used. The found path can be tracked by following the connections marked with *arrows*.

5. Nodes whose cost values have been calculated for the first time are added to the open list. The nodes that have already been in the open-list and were updated are kept in the open list. The nodes that have been updated and were in the closed list are moved to the open-list. The open list is sorted according to the cost-of-the-whole-path in increasing order. The current node is transferred to the closed list.

In the first subfigure (in Fig. 4.30) the current node is the start node and its successor nodes (accessed in four directions: left, right, up, and down from the current node) are selected. For all the successor nodes, the cost-to-here is one as they are only one step away from the star node and cost-to-goal is the Manhattan distance (heuristic) from the successor node to the goal node, which can also be measured through the obstacle. The sum of both costs is the cost-of-the-whole path. The successors nodes also have a connection marked by an arrow (direction) to its parent node (current node marked by a circle). The open list now contains these four successor nodes and closed list contains the starting node.

In the second subfigure (in Fig. 4.30) the node with the lowest cost-of-the-whole path (with cost 6) is selected from the open list as the current node. The current node has only one successor (other cells are blocked by the obstacle or in the closed list). Cost-to-here for this successor node is 2 as it is two steps away (Manhattan distance) from the start node and cost-to-goal is 8. The current node is put to the closed list and the successor node to the open list. In the third subfigure (in Fig. 4.30) the current node becomes the node from the open list with the lowest cost-of-the-whole path, which is 6. Then the algorithm continues similarly as in the first two steps.

The A* algorithm is guaranteed to find the optimal path in graph if the heuristic for calculation of the cost-to-goal is admissible (or optimistic), which means that the estimated cost-to-goal is smaller or equal to true cost-to-goal. The algorithm finishes when the goal node is added to the closed list.

The A* algorithm is a complete algorithm because it finds the path if it exists, and as already mentioned, it is optimal if the heuristic is admissible (optimistic). Its drawback is large memory usage. If all costs-to-goal are set to zero the A* operation equals Dijkstra's algorithm. In Fig. 4.31 a comparison of Dijkstra's and A* algorithm performance is shown.

4.4.6 Greedy Best-First Search

Greedy best-first search is the informed algorithm. The open list is sorted in the increasing cost-to-goal. Therefore, the search in each iteration is extended to the open node that is closest to the goal (has the smallest cost-to-goal) assuming it will reach the goal quickly. The found path is not guaranteed to be optimal as shown in Fig. 4.32. The algorithm only considers cost from the current node to the goal and ignores the cost required to come to the current node; therefore, the overall path may become longer than the optimum one. Because the algorithm is not optimal it is also not necessary that the heuristic is admissible as it is important in A*.

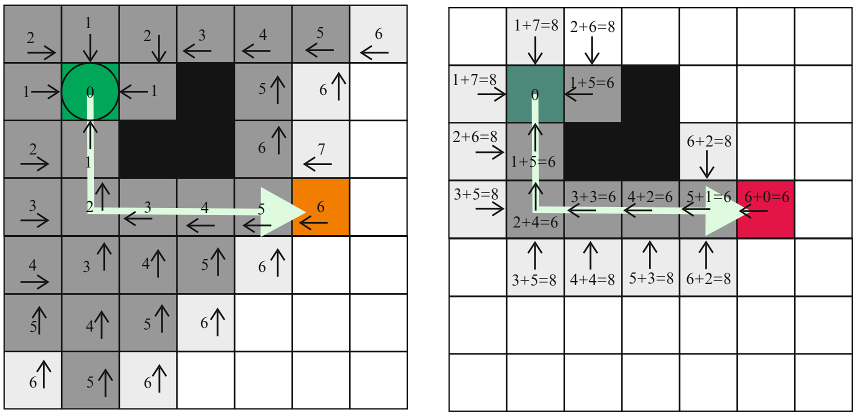

Fig. 4.31 Comparison of Dijkstra's (*left*) and A* (*right*) algorithm performance. Both algorithms find the shortest path to the goal; however, A* algorithm requires less iterations due to the used heuristics when searching the graph.

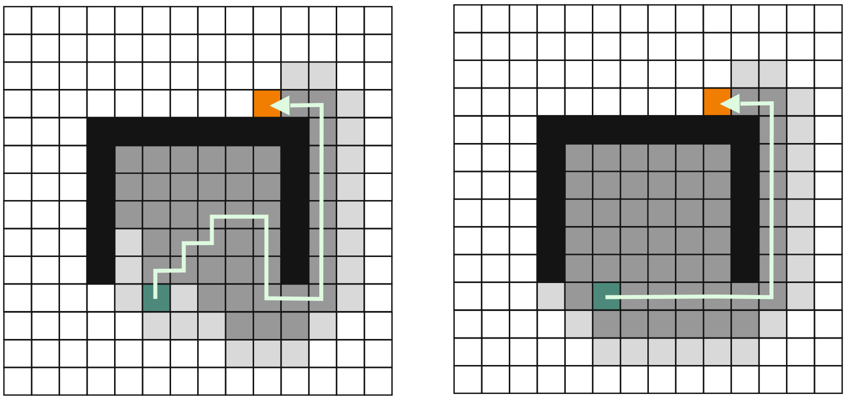

Fig. 4.32 Comparison of greedy best-first search (*left*) and A* (*right*) algorithm. The path found by the greedy best-first search algorithm is not optimal. In most cases it can find the solution faster but it is not always optimal. *Dark gray* nodes are the ones from the closed list, while the *light gray* nodes are from the open list.

EXAMPLE 4.11

Implement path planning using A* algorithm for the environment with cell decomposition in Fig. 4.32. Compare the obtained results with the results in Fig. 4.32. For calculation of the distance use Manhattan distance, while for the heuristic (cost-to-goal) use Manhattan distance or Euclidean distance, and also compare results obtained by both distances.

EXAMPLE 4.11—cont'd

Modify obtained code to also implement the greedy best-first search algorithm.

Solution

Although the reader is encouraged to implement the A* algorithm, a possible implementation of the algorithm is shown in Listing 4.10, where Manhattan distance is used to calculate the cost-to-here and Euclidean distance is used to calculate the heuristic.

Listing 4.10 Implementation of A* algorithm

```
classdef AStarBase < handle
properties
    map = []; % Map: 0 - free space, 1 - obstacles
    open = []; closed = []; start = []; goal = []; act = []; path = [];
end

methods
    function path = find(obj, start, goal) % start=[is; js], goal=[ig; jg]
        obj.start = start; obj.goal = goal; obj.path = [];
        obj.closed = []; % Empty closed list
        obj.open = struct('id', start, 'src', [0; 0], 'g', 0, ...
                          'h', obj.heuristic(start)); % Initial open list

        if obj.map(start(1), start(2))~=0 || obj.map(goal(1), goal(2))~=0
            path = []; return; % Path not feasible!
        end

        while true % Search loop
            if isempty(obj.open), break; end % No path found :(

            obj.act = obj.open(1); % Get node from the ordered open list,
            obj.closed = [obj.closed, obj.act]; % add it to the closed list
            obj.open = obj.open(2:end); % and remove it from the open list.

            if obj.act.id(1)==obj.goal(1) && obj.act.id(2)==obj.goal(2)
                % Path found :) Get the path from the closed list ...
                p = obj.act.id; obj.path = [p]; ids = [obj.closed.id];
                while sum(abs(p-start))~=0 % Follow src nodes to the start
                    p = obj.closed(ids(1,:)==p(1) & ids(2,:)==p(2)).src;
                    obj.path = [p, obj.path];
                end
                break;
            end

            neighbours = obj.getNodeNeighbours(obj.act.id);
            for i = 1:size(neighbours, 2)
                n = neighbours(:,i);
                % Add neighbour to the open list if it is not in the closed
                % list and it is not an obstacle.
                ids = [obj.closed.id]; z = ids(1,:)==n(1) & ids(2,:)==n(2);
                if isempty(find(z, 1)) && ~obj.map(n(1), n(2))
                    obj.addNodeToOpenList(n);
                end
            end
        end
```

(Continued)

EXAMPLE 4.11—cont'd

```
        path = obj.path;
    end
    function addNodeToOpenList(obj, i)
        g = obj.act.g + obj.cost(i); % Path cost
        % Check if the node is in already the open list
        ids = [obj.open.id]; s = [];
        if ~isempty(ids)
            s = sum(abs(ids-repmat(i, 1, size(ids, 2))))==0;
        end
        if isempty(find(s, 1)) % Add new node to the open list
            node = struct('id', i, 'src', obj.act.id, ...
                          'g', g, 'h', obj.heuristic(i));
            obj.open = [obj.open, node];
        else % Update the node in the open list if it has better score
            if g<obj.open(s).g
                obj.open(s).g = g;
                obj.open(s).src = obj.act.id;
            end
        end
        % Sort open list
        [~,i] = sortrows([[obj.open.g]+[obj.open.h]; obj.open.h].', [1,2]);
        obj.open = obj.open(i);
    end

    function n = getNodeNeighbours(obj, a)
        n = [a(1)-1, a(1), a(1), a(1)+1; a(2), a(2)-1, a(2)+1, a(2)];
        [h, w] = size(obj.map);
        n = n(:, n(1,:)>=1 & n(1,:)<=h & n(2,:)>=1 & n(2,:)<=w); % Bounds
    end

    function g = cost(obj, a)
        g = sum(abs(a-obj.act.id)); % Manhattan distance
    end

    function h = heuristic(obj, a)
        h = sqrt(sum((a-obj.goal).^2)); % Euclidean distance
    end
end
end
```

The usage of the A* algorithm given in Listing 4.10 is shown in Listing 4.11. The columns of output variable `path` represent an ordered list of cells that lead from the start to goal. A minimum modification of the algorithm is needed in order to use different heuristic (e.g., Manhattan distance as it was used to obtain the path shown in Fig. 4.32).

Listing 4.11 Usage of A* algorithm

```
map = zeros(14, 14); % Map
map(5:10,[4 11]) = 1; map(5,4:11) = 1; % Obstacles

astar = AStarBase();
astar.map = map;
path = astar.find([11; 6], [4; 10])
```

EXAMPLE 4.11—cont'd

```
path =
  Columns 1 through 13
    11     11     11     11     11     11     11     10      9      8      7      6      5
     6      7      8      9     10     11     12     12     12     12     12     12     12
  Columns 14 through 16
     4      4      4
    12     11     10
```

Also the A* algorithm can be converted to the greedy best-first search algorithm if the cost-to-here is set to zero.

EXAMPLE 4.12

Extend Example 4.11 also with the quadtree environment decomposition and optimal path solution using the A* algorithm.

Solution

In order to extend the A* algorithm in Listing 4.10 with the quadtree map, only some minor modifications are required. In this case all the cells are not of the same size, therefore, the distance between the cells varies, and also the number of neighbors varies for each cell. Hence, the way neighboring cells are determined needs to be modified. Implementation of a quadtree-decomposition algorithm in Listing 4.2 already creates a visibility map; that is, it finds all the neighboring cells for each cell in the quadtree. In order to determine the cost-to-here, the distance between the cells can be defined as the Euclidean distance between the cell centers (see Fig. 4.10). The reader is encouraged to implement these modifications.

REFERENCES

[1] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, S. Thrun, Principles of Robot Motion: Theory, Algorithms, and Implementations, MIT Press, Boston, MA, 2005.

[2] S.M. LaValle, Planning Algorithms, Cambridge University Press, Cambridge, NY, 2006.

[3] B. Siciliano, O. Khatib, Springer Handbook of Robotics, Springer, Berlin, 2008.

[4] S.M. LaValle, Rapidly-exploring random trees: a new tool for path planning, Technical Report TR 98-11, Computer Science Department, Iowa State University, Iowa, 2013.

[5] S.M. LaValle, J.J. Kuffner, Randomized kinodynamic planning, Int. J. Robot. Res. 20 (5) (2001) 278–400.

[6] J.J. Kuffner, S.M. LaValle, RRT-connect: an efficient approach to single-query path planning, in: IEEE International Conference on Robotics and Automation (ICRA 2000), IEEE, 2000, pp. 1–7.

[7] L.E. Kavraki, P. Svestka, J.-C. Latombe, M.H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Trans. Robot. Autom. 12 (4) (1996) 566–580.
[8] Z. Sun, D. Hsu, T. Jiang, H. Kurniawati, J.H. Reif, Narrow passage sampling for probabilistic roadmap planning, IEEE Trans. Robot. 21 (6) (2005) 1105–1115.
[9] V. Lumelsky, P. Stepanov, Dynamic path planning for a mobile automaton with limited information on the environment, IEEE Trans. Autom. Control 31 (11) (1986) 1058–1063.
[10] A. Sankaranarayanan, M. Vidyasagar, A new path planning algorithm for moving a point object amidst unknown obstacles in a plane, in: IEEE Conference on Robotics and Automation, IEEE, 1990, pp. 1930–1936.
[11] I. Kamon, E. Rivlin, Sensory-based motion planning with global proofs, IEEE Trans. Robot. 13 (6) (1997) 814–821.
[12] S. Laubach, J. Burdick, An autonomous sensor-based path-planner for planetary microrovers, in: IEEE Conference on Robotics and Automation, IEEE, 1999, pp. 347–354.
[13] K. Mehlhorn, P. Sanders, Algorithms and Data Structures: The Basic Toolbox, Springer, Berlin, 2008.
[14] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.