# 6 *Cell Decompositions*

NEXT, WE consider a different type of representation of the free space called an *exact cell decomposition.* These structures represent the free space by the union of simple regions called *cells.* The shared boundaries of cells often have a physical meaning such as a change in the closest obstacle or a change in line of sight to surrounding obstacles. Two cells are *adjacent* if they share a common boundary. An *adjacency graph,* as its name suggests, encodes the adjacency relationships of the cells, where a node corresponds to a cell and an edge connects nodes of adjacent cells.

Assuming the decomposition is computed, path planning with a cell decomposition is usually done in two steps: first, the planner determines the cells that contain the start and goal, respectively, and then the planner searches for a path within the adjacency graph. Note that the adjacency graph could serve as a roadmap of the free space as well. Therefore, mapping can be achieved by incrementally constructing the adjacency graph.

Cell decompositions, however, distinguish themselves from other methods in that they can be used to achieve coverage. A coverage path planner determines a path that passes an effector (e.g., a robot, a detector, etc.) over all points in a free space. Since each cell has a simple structure, each cell can be covered with simple motions such as back-and-forth farming maneuvers; once the robot visits each cell, coverage is achieved. In other words, coverage can be reduced to finding an exhaustive walk through the adjacency graph. Sensor-based coverage is achieved by simultaneously covering an unknown space and constructing its adjacency graph.

The most popular cell decomposition is the *trapezoidal decomposition* [356]. This decomposition relies heavily on the polygonal representation of the planar

configuration space. A more general class of decompositions, which are termed *Morse Decompositions* [12], allow for representations of nonpolygonal and nonplanar spaces. Morse decompositions are based on ideas from Canny's roadmap work. We then consider a broader class of decompositions which includes those based on visibility constraints. One such decomposition serves as a basis for the pursuit/evasion problem which is introduced section 6.3.

## 6.1 Trapezoidal Decomposition

The trapezoidal decomposition comprises two-dimensional cells that are shaped like trapezoids. Some cells can be shaped like triangles, which can be viewed as degenerate trapezoids where one of the parallel sides has a zero-length edge. Assume a simple $(x, y)$ coordinate system for the planar configuration space, the free space is bounded by a polygon and that all of the obstacles are polygonal. For the sake of explanation, assume that each vertex $v_i$ on all of the polygons has a unique $x$ coordinate, i.e., for all $i \neq j$, $v_{i_x} \neq v_{j_x}$. This assumption is equivalent to saying that the polygons lie in general position (see figure 6.1).

To form the decomposition, at each vertex $v$ draw two segments, one called an upper vertical extension and the other called a lower vertical extension. Here, "up" and "above" correspond to increasing the $y$ coordinate, and likewise "down" and "below" mean decreasing it. The upper and lower vertical extensions start at the vertex and

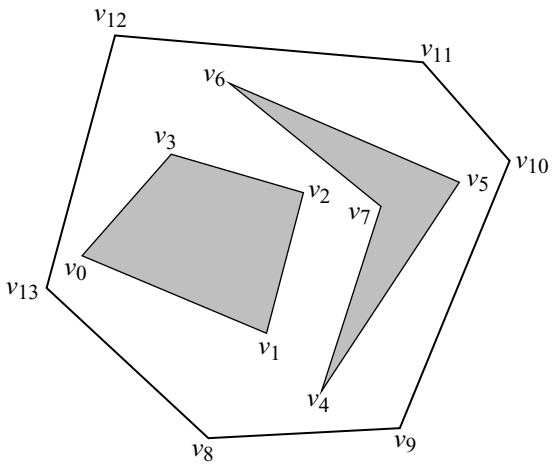



**Figure 6.1** Sample polygonal configuration space.

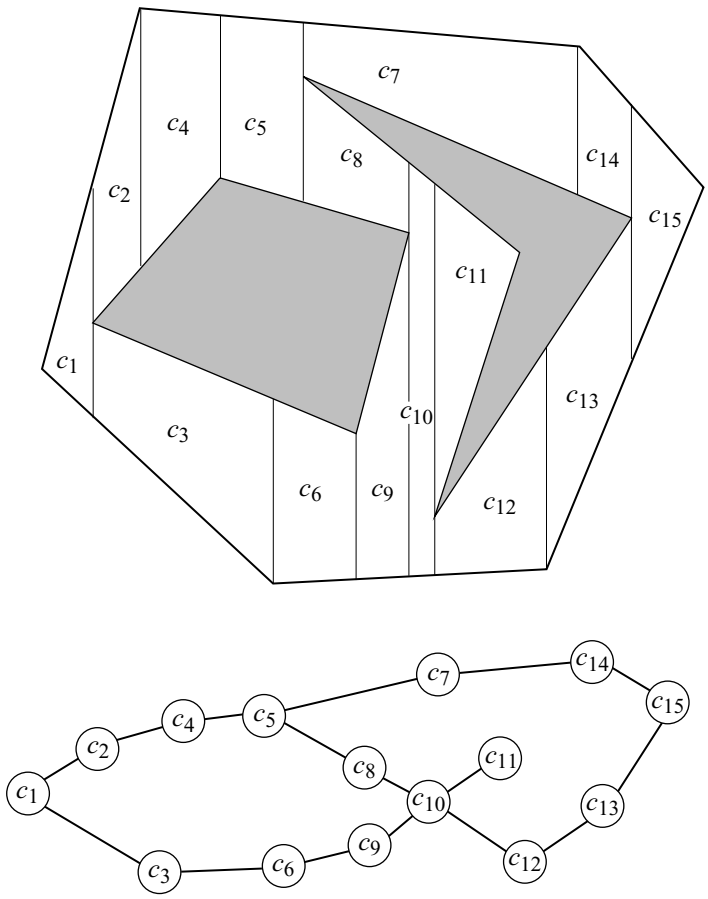



**Figure 6.2** Trapezoidal decomposition for the configuration space in figure 6.1.

terminate when they first intersect an edge of the polygon that lies immediately above and below $v$, respectively. Note that many vertices will have either just an upper or a lower vertical extension. Figure 6.2 contains the trapezoidal decomposition and its adjacency graph for the workspace in figure 6.1. Recall that two cells are adjacent if they share a common boundary, i.e., a common vertical extension.

Once the cells that contain the start and goal are determined, the planner searches the adjacency graph to determine the path. However, the result of the graph search is just a sequence of nodes, not a sequence of points embedded in the free space, and so the next step is to determine the explicit path. Since a trapezoid is a convex set,

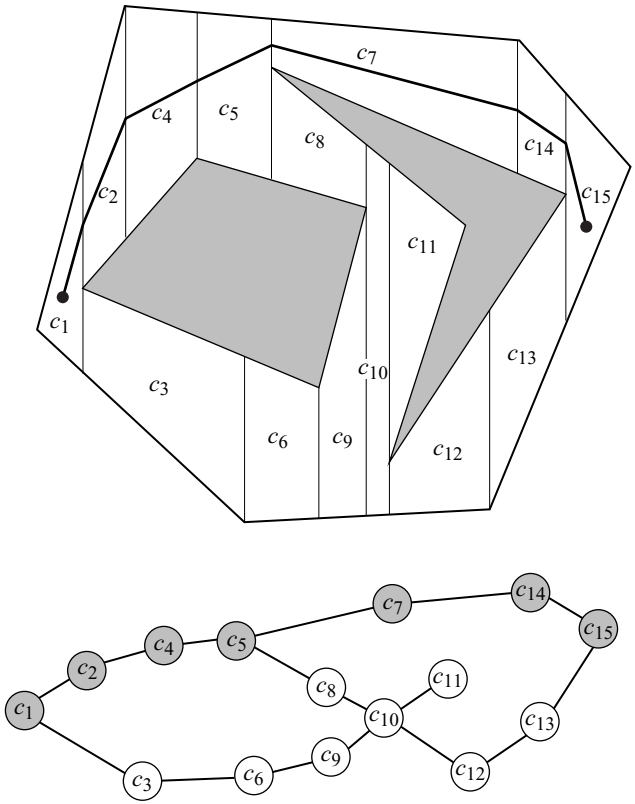



**Figure 6.3** The resulting paths in the adjacency graph and free space.

any two points on the boundary of a trapezoidal cell can be connected by a straight-line segment that does not intersect any obstacle. The planner constructs the path, one trapezoid at a time, by connecting the midpoints of the vertical extensions to the centroids of each trapezoid. This yields a connected collision-free path through the free space that is derived from the adjacency graph. To connect the start and goal points, simply draw a straight line to the vertical extensions' midpoints of the appropriate trapezoids (figure 6.3).

The next issue centers on constructing the decomposition itself. The input to the algorithm is a list of polygons, each represented by a list of vertices. The first step is to sort the vertices based on the $x$-coordinate of each vertex. This takes $O(n \log n)$ time and $O(n)$ storage where $n$ is the number of edges (or vertices) in all of the polygons.

The next step is to determine the vertical extensions. For each vertex $v_i$, a naive algorithm can intersect a line through $v_i$ with each edge $e_j$ for all $j$. This will require $O(n)$ time resulting in a $O(n^2)$ time to construct the trapezoidal decomposition. We can do better: the extensions can be determined by sweeping a sweep line (similar to the sweep line in chapter 5, section 5.1 or the slice in Canny's roadmap algorithm in section 5.5.1) through the free space stopping at the vertices, which are sometimes termed *events.* While passing the sweep line, the planner can maintain a list $L$ that contains the "current" edges which the slice intersects.

With the list $L$, determining the vertical extensions at each event requires $O(n)$ time with a simple search, but if the list is stored in an "efficient" data structure like a balanced tree, then the search requires $O(\log n)$ time. It is easy to determine the $y$-coordinates of the intersection of the line that passes through $v_i$ and each edge $e_i$. The trick is to find the appropriate edge or edges for the vertical extensions, i.e., the two edges that $v$ lies between. Let these two edges be called $e_{\text{LOWER}}$ and $e_{\text{UPPER}}$.

So as long as the "current" list requires $O(\log n)$ insertions and deletions, as balanced trees do, then keeping track of all the edges that intersect the sweep line, i.e., maintaining $L$, requires $O(n \log n)$ time. Let $e_{\text{lower}}$ and $e_{\text{upper}}$ be the two edges that contain $v$ (these are *not* $e_{\text{LOWER}}$ and $e_{\text{UPPER}}$). The "other" vertex of $e_{\text{lower}}$ has a $y$-coordinate lower than the "other" vertex of $e_{\text{upper}}$. Now, there are four types of events (figure 6.4) that can occur and the type of event determines the appropriate action to take on the list. These events and actions are

- $e_{\text{lower}}$ and $e_{\text{upper}}$ are both to the left of the sweep line
  - delete $e_{\text{lower}}$ and $e_{\text{upper}}$ from the list
  - $(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots) \rightarrow (\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots)$
- $e_{\text{lower}}$ and $e_{\text{upper}}$ are both to the right of the sweep line
  - insert $e_{\text{lower}}$ and $e_{\text{upper}}$ into the list
  - $(\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots) \rightarrow (\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

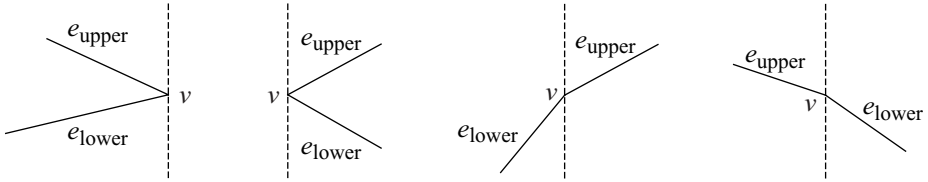



**Figure 6.4** The kinds of events.

- $e_{\text{lower}}$ is to the left and $e_{\text{upper}}$ is to the right of the sweep line
  - delete $e_{\text{lower}}$ from the list and insert $e_{\text{upper}}$
  - $(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{UPPER}}, \ldots) \rightarrow (\ldots, e_{\text{LOWER}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

- $e_{\text{lower}}$ is to the right and $e_{\text{upper}}$ is to the left of the sweep line
  - delete $e_{\text{upper}}$ from the list and insert $e_{\text{lower}}$
  - $(\ldots, e_{\text{LOWER}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots) \rightarrow (\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{UPPER}}, \ldots)$

Figures 6.5 through 6.8 contain examples of a sweep line being swept through a polygonal free space of figure 6.1 with the corresponding list updates at each event.

Finally, we need to determine which cells contain the start and goal. First, we will seemingly construct a finer cell decomposition which will have cells that are subsets of the trapezoid and then infer from there which trapezoids contain the start and goal. Draw a vertical line through all of the events forming "slabs" of the free space. Let $w$ be a point in the free space. Determining which slab contains $w$ requires $O(\log n)$ time. From here, it is easy to determine which edge of the polygonal workspace intersects the slab and thus it requires a second $O(\log n)$ search to determine the ceiling and floor edges that contain $w$. With the ceiling and floor determined, it is trivial to determine which trapezoid contains $w$. See [124] for more efficient algorithms.

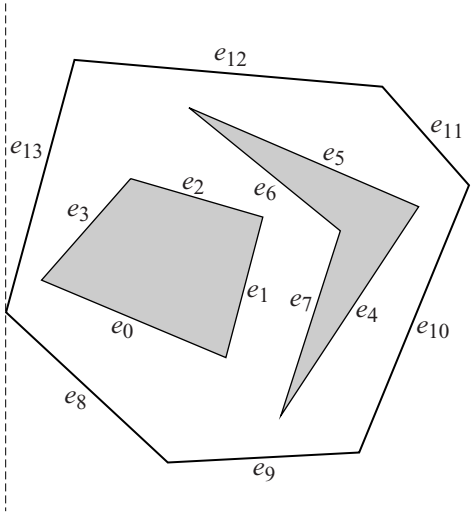



**Figure 6.5** $L : \emptyset \rightarrow \{e_8, e_{13}\}$.

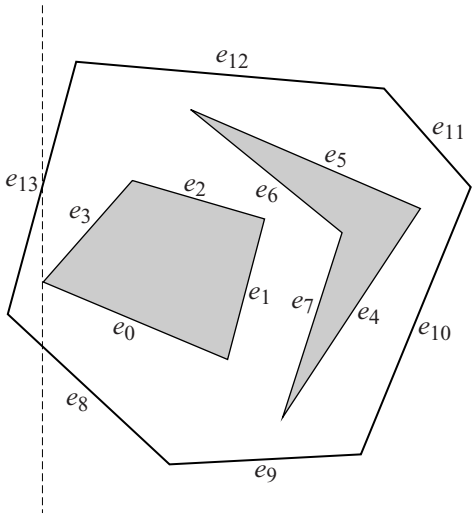



**Figure 6.6** $L : \{e_8, e_{13}\} \rightarrow \{e_8, e_0, e_3, e_{13}\}$.

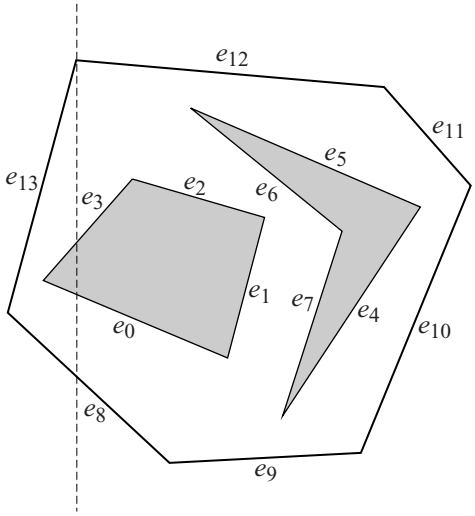



**Figure 6.7** $L : \{e_8, e_0, e_3, e_{13}\} \rightarrow \{e_8, e_0, e_3, e_{12}\}$.

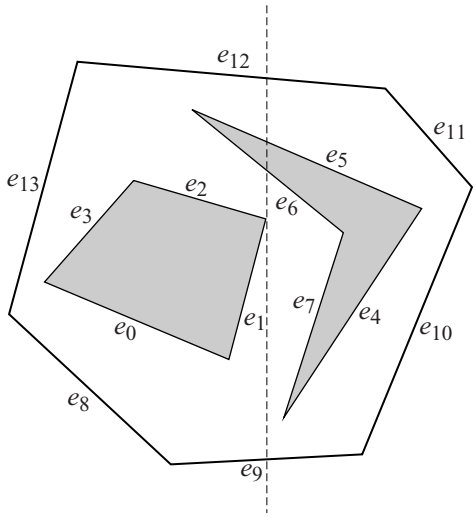



**Figure 6.8** $L : \{e_9, e_1, e_2, e_6, e_5, e_{12}\} \rightarrow \{e_9, e_6, e_5, e_{12}\}$.

## 6.2 Morse Cell Decompositions

Conventional motion planning approaches determine paths between start and goal configurations, such as those described in chapters 2, 4, 5, and elsewhere. However, applications such as robotic demining and floor cleaning require a robot to pass over all points in its free space, i.e., follow a path to *cover* the space. A planner can use an exact cell decomposition to cover an unknown space by simply covering each cell and then using the adjacency graph to ensure each cell is visited and hence covered. This approach requires that each cell can indeed be covered. Naturally, cells with simple structure can easily be covered; e.g., the cells of the trapezoidal decomposition can be covered with simple back-and-forth motions.

Unfortunately, the trapezoidal decomposition may not produce efficient paths for coverage. Here, we measure efficiency in terms of area covered vs. path length traversed. Observe that cells in the trapezoidal decomposition can be "clumped" together to form more efficient coverage paths. Perhaps a bigger drawback to the trapezoidal method is that it fundamentally requires a polygonal workspace, which is not a realistic assumption for many applications.

In this section, we use Morse functions (chapter 5, section 5.5) to define cells that have simple structure and can be defined in nonpolygonal spaces. Recall that a Morse function is one whose critical points are nondegenerate; from a practical perspective,

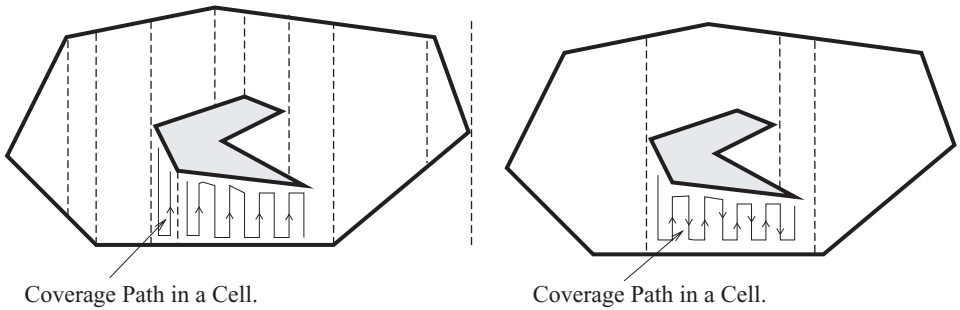



**Figure 6.9** (left) Trapezoidal decomposition and (right) boustrophedon decomposition for the same space. Each cell, in both decompositions, can be covered with simple back and forth motions. However, note that the coverage path in the boustrophedon decomposition is a little bit shorter. The region below the polygon obstacle requires an extra pass because the planner has to "start over" each time the robot enters a new cell. Since there are fewer cells under the polygonal obstacle in the boustrophedon decomposition, the coverage path is shorter.

this means that critical points are isolated. In this section, we evolve the trapezoidal decomposition to a new decomposition called the *boustrophedon decomposition* and then show that the boustrophedon decomposition is a *Morse decomposition.* Next, we generalize the boustrophedon decomposition to form other Morse decompositions.

### 6.2.1 Boustrophedon Decomposition

From a coverage perspective, a minor shortcoming of the trapezoidal decomposition is that many small cells are formed that can seemingly be aggregated with neighboring cells. Reorganizing the cells can result in a shorter and more efficient path to cover the same area.

To address this issue, the boustrophedon[1] cell decomposition approach was introduced [110]. The boustrophedon decomposition is formed by considering the vertices at which a vertical line can be extended both up and down in the free space (figure 6.9). We call such vertices *critical points,* and we will show that these correspond to the same critical points in Canny's roadmap described in chapter 5, section 5.5.1.

With the decomposition in hand, the planner determines a coverage path in two steps. First, the planner determines an exhaustive walk through the adjacency graph (figure 6.10). This list can be computed by using a depth-first search algorithm. Once

1. The Greek word *boustrophedon* literally means "ox turning" [8]. Typically, when an ox drags a plow in a field, it crosses the full length of the field in a straight-line, turns around, and then traces a new straight line path adjacent to the previous one.

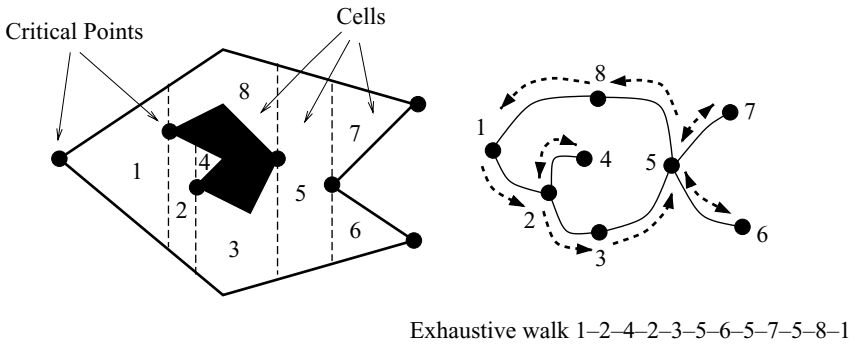



**Figure 6.10** The boustrophedon decomposition of a space and its adjacency graph. The nodes represent the cells and the edges indicate the adjacent cells. An exhaustive walk on the graph is generated.

the ordered list of cells is determined, the planner then computes the explicit robot motions within each cell. The path in each cell consists of a repeated sequence of straight-line segments separated by one robot width and short segments connecting the straight line-segments. Typically, these short segments follow the boundary of the environment.

### 6.2.2 Morse Decomposition Definition

We generalize the boustrophedon decomposition beyond polygons by borrowing ideas from Canny's work [91,93] which first applied a "slicing method" to motion planning, as described in chapter 5. Recall that a *slice* is a is a codimension one manifold denoted by $\mathcal{Q}_\lambda$. The slices are parameterized by $\lambda$ (varying $\lambda$ sweeps a slice through the space). The portion of the slice in the free configuration space, $\mathcal{Q}_{\text{free}}$, is denoted by $\mathcal{Q}_{\text{free}\lambda}$, i.e., $\mathcal{Q}_{\text{free}\lambda} = \mathcal{Q}_\lambda \bigcap \mathcal{Q}_{\text{free}}$. Recall from chapter 5 that connectivity changes of $\mathcal{Q}_{\text{free}\lambda}$ were used to ensure the connectivity of the roadmap. Now, we are going to use the connectivity changes to define cells in a cell decomposition.

Recall from section 5.5 in chapter 5 that the slice can be defined in terms of the preimage of the projection operator $\pi_1 : \mathcal{Q} \rightarrow \mathbb{R}$. In the plane $\pi_1(x, y) = x$ and the slice $\mathcal{Q}_\lambda = \pi_1^{-1}(\lambda)$ corresponds to a vertical slice. Increasing the value of $\lambda$ sweeps the slice to the right through the plane. As the slice is swept through the target region, obstacles intersect (or stop intersecting) the slice in the free space, severing it into smaller pieces as the slice first encounters an obstacle (or merging smaller pieces into larger pieces as the slice immediately departs an obstacle). The connectivity

changes occur at points termed *critical points.* Note that critical points are analogous to vertices which have vertical extensions that can be drawn both up and down.

Slices that contain critical points are termed *critical slices.* It should be emphasized, however, that the slice $\mathcal{Q}_\lambda$ itself does not change connectivity, but rather the slice in the free space $\mathcal{Q}_{\text{free}\lambda}$ changes connectivity at critical points. Naturally, $\mathcal{Q}_{\text{free}\lambda}$ contains one or more connected components, which are termed *slice intervals* and are denoted $\mathcal{Q}_{\text{free}\lambda}^{j}$ for the $j$th open connected slice interval. So, $\mathcal{Q}_{\text{free}\lambda} = \bigcup_j \mathcal{Q}_{\text{free}\lambda}^{j}$. Denote the set of slice intervals that contain a critical point by $I^*$. Note that a critical point cannot be in the interior of a slice interval; it can only lie at the endpoints of a slice interval. With this, we can define a Morse decomposition

DEFINITION 6.2.1 (Morse Decomposition [12]) *A Morse decomposition is an exact cell decomposition whose cells are the connected components of* $\mathcal{Q}_{\text{free}} \setminus I^*$.

In figure 6.11, the dashed lines are the slice intervals lying in the free space and have end points lying on obstacle boundaries. Each of these slice intervals has at least one critical point on an obstacle boundary as well. When the slice intervals are removed from the free space, the remaining free space is still two-dimensional but is no longer connected. Each connected component is a cell.

One can see that within a cell, the slice interval remains connected and only extends or contracts. Morse theory assures us that between critical slices, "merging" and "severing" of slices do not occur, i.e., the topology of the slice remains constant.

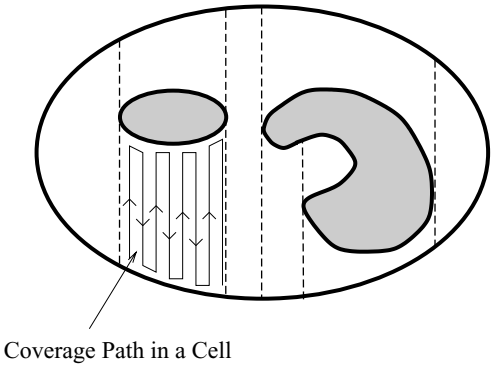



**Figure 6.11** The boustrophedon decomposition of a nonpolygonal environment. As we sweep a straight-line slice from left to right, its connectivity in the free space changes first from one to two, then two to one and so forth. At the points where these connectivity changes occur, we locate the cell boundaries in the free space.

This is useful for tasks such as coverage because the robot can trivially perform simple motions between critical points and guarantee complete coverage of a cell (figure 6.11). A coverage path within a cell contains two parts: motion along the slice and motion along the boundary of the obstacles. A bulk of the coverage operation occurs with motions along the slices, sometimes called *laps,* and this motion terminates when the robot encounters an obstacle. Motion along the boundary of the obstacle directs the robot to move "one width[2] over," i.e., increase its slice function value by one robot width while following the cell boundary along an obstacle. We call the distabce between subsequent laps as the *inter-lap distance*.

Cao, Huang, and Hall [96] implicitly use a Morse decomposition to achieve coverage but they assume all obstacles are convex. The Morse decompositions, defined above, assume that the critical points are not degenerate, but Butler et al. [86] present a coverage algorithm that uses decompositions of rectilinear spaces where all critical points are degenerate.

### 6.2.3 Examples of Morse Decomposition: Variable Slice

The definition of a Morse decomposition is not specific to a particular slice. In the previous section, the slice was defined by the preimage of a real-valued function, which happened to be $\pi_1$. We can use this function to define the boustrophedon decomposition which induces back-and-forth coverage pattern. Now we will vary the function that defines the shape of the slice, resulting in different decompositions and hence different patterns by which the free space is covered. Now, we rewrite the definition of the slice as the preimage of a general real-valued function $h : \mathcal{Q} \rightarrow \mathbb{R}$. For the boustrophedon decomposition in the plane, this function is $h(x, y) = x$.

#### Spiral, Spike, and Squarel Patterns

We can use the function $h(x, y) = \sqrt{x^2 + y^2}$ to produce a pattern of concentric circles in the plane. Critical points occur at points where a circle changes connectivity; this happens when it is tangent to an obstacle. Critical points are then used to form annular or arc-shaped cells and the adjacency graph (figure 6.12, left). As before, a planner determines a coverage path in two steps: first it finds an exhaustive walk through the adjacency graph and then it plans the explicit coverage path in each uncovered cell. The coverage pattern within a cell has three parts: motion along a slice, motion orthogonal to the slice, and motion along the boundary of the cell. The slice here, however, is not a straight-line segment, but rather a circle or subset of a circle. Therefore, in

2. Here, width is determined by the size of the detector or end-effector that is being used.

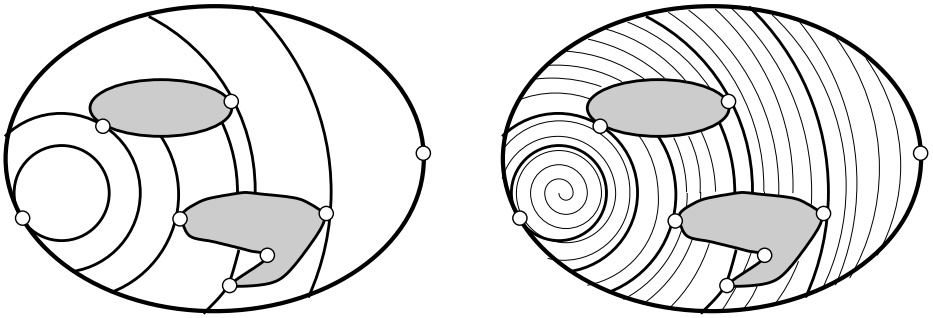


**Figure 6.12** (Left) Cell decomposition for $h(x, y) = \sqrt{x^2 + y^2}$ and (right) its associated spiral coverage pattern. The slices are the circles that are the preimages of $h$. At the critical points, labeled with little open circles (not to be confused with the slices), the circle-shaped slices become tangent to the obstacles. Rather than moving along circular paths and stepping outwardly, the robot follows a spiral pattern.

the plane, a planner initially directs the robot to circumnavigate a circle, move the interlap distance along the radius of the circle, and then circumnavigate a circle of a larger radius. If the robot encounters an obstacle while circumnavigating a circle, the planner simply directs the robot to follow the obstacle boundary until the robot has moved an interlap distance and then follows the circle of a larger radius.

Note that instead of following a circle and stepping outward, the robot can follow a spiral pattern until it encounters critical points (figure 6.12, right). The spiral pattern bypasses the need to step along the radial direction. This yields a path that maximizes the area covered per unit distance traveled in regions sparsely populated with obstacles.

The function $h(x, y) = \tan(\frac{y}{x})$ induces a pattern that is orthogonal to the set of concentric circles (figure 6.13). Using this pattern to perform coverage has the effect of covering the region closest to the center of the pattern more densely. This is useful if the likelihood of finding a desired object is highest at the center of the pattern and the robot's detector experiences false negatives (something is under the detector but the detector does not sense it).

The function $h(x, y) = |x| + |y|$ can be used to produce cells that look like rotated squares or diamonds (figure 6.14). For coverage, instead of driving in concentric squares, we can direct the robot to "spiral" out while looking for critical points, hence the term *squarel*. The resulting pattern is shown in figure 6.14. The squarel pattern serves as an approximation to the spiral pattern that is easier to implement on differential drive robots.

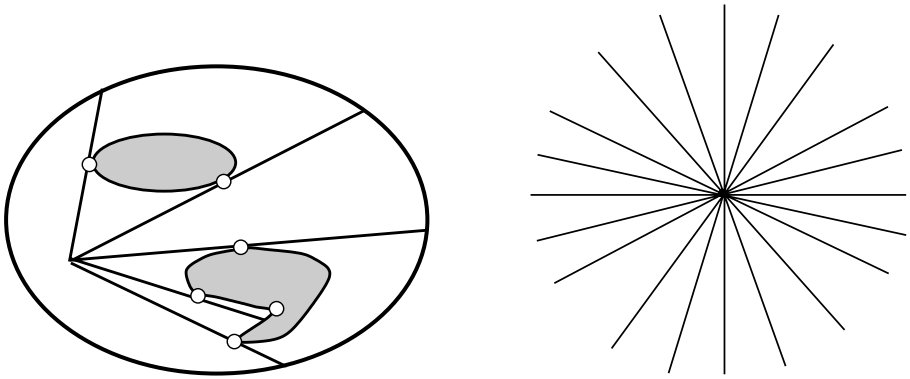


**Figure 6.13** Decomposition for $h(x, y) = \tan(\frac{y}{x})$ and a spiked pattern. The free space is sliced like a pie. At the critical points, the slices are tangent to obstacles. The robot can use this pattern to cover more densely the region closest to the center of the pattern.

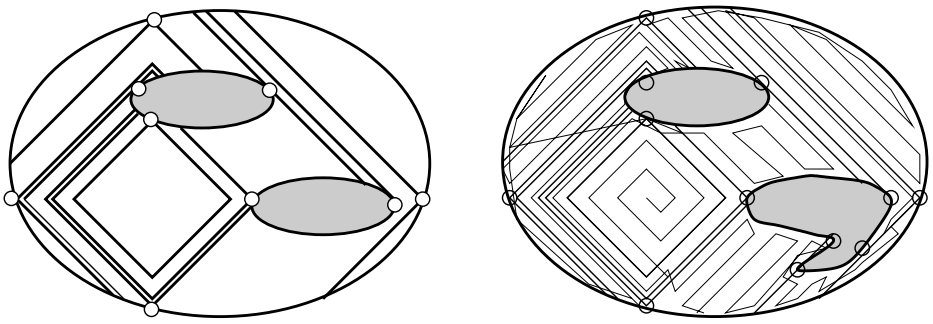


**Figure 6.14** Decomposition for $h(x, y) = |x| + |y|$ and a coverage pattern. Squares are the slices and at the critical points the corner of a square touches an obstacle or the side of it becomes tangent to an obstacle. Since it is easier for the robot to move along straight lines rather than circles, this pattern can be used to approximate the spiral pattern.

Note that $h(x, y) = |x| + |y|$ is not smooth so we have to use the formulation of the generalized gradient given in chapter 5, section 5.5.2. The squarel pattern has two parts: a straight line segment and a 90 degree turn, as can be seen in figure 6.14. Note that critical points occur when the flat portions of $\{(x, y) : |x|+|y| = \lambda\}$ become tangent to an obstacle and at some of the 90 degree turn points. At these points, the obstacle surface normal lies in the convex hull of the two flat portions that meet at the 90 degree turn point.

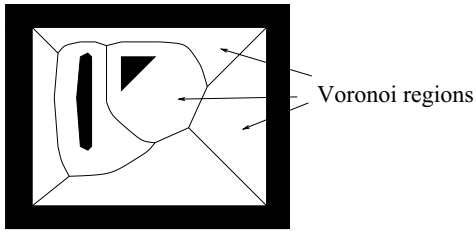



**Figure 6.15** GVD of an environment.

### Brushfire Decomposition

The brushfire algorithm [262] is a popular technique to construct the GVD (figure 6.15). In chapter 4, section 4.3.2, we described the brushfire algorithm on a grid; here we describe it on a continuous space.

The brushfire algorithm is so named because in implementation, imaginary wave fronts emanate from each obstacle and collide at points on the GVD. By noting the location of the collision points, the algorithm constructs the GVD. The algorithm, however, induces a decomposition that is *not* the Voronoi regions of the GVD. Instead, the decomposition models the topology of the wave fronts as they initially collide with each other and form or destroy new wave fronts. Compare figure 6.15 and figure 6.16.

The distance function $D$, which measures the distance between the point $x$ and the nearest point $c$ on the closest obstacle $\mathcal{QO}_i$, admits a decomposition termed the *brushfire decomposition.* Each slice of $D$ is a wave front where each point on the front has propagated a distance $\lambda$ from the closest obstacle. As $\lambda$ increases, the wave fronts progress. Cells of the brushfire decomposition are formed when these wave fronts initially collide. Figure 6.16 contains a decomposition induced by $D$ where regions of the same color represent a cell. Whereas for the boustrophedon decomposition we are essentially "pushing" a line segment through the cell, here we are "growing" a wave front that originates on the boundary of the environment, which in figure 6.16 has three obstacles: the exterior, the vertical barlike obstacle, and a triangle. These three wave fronts progress until they initially collide with each other, which occurs at critical points. The light gray regions adjacent to the obstacles represent the three newly formed cells. The type of critical points that define the gray regions in figure 6.16 are saddle points. In fact, all of the cells are defined by saddle points of $D$. Note that since $D$ is nonsmooth, its generalized gradient must be used as well.

To determine the coverage path, the planner again first derives the decomposition, finds an exhaustive walk through the adjacency graph, and then plans the coverage path within each cell. The coverage path within a cell consists of three parts: motion

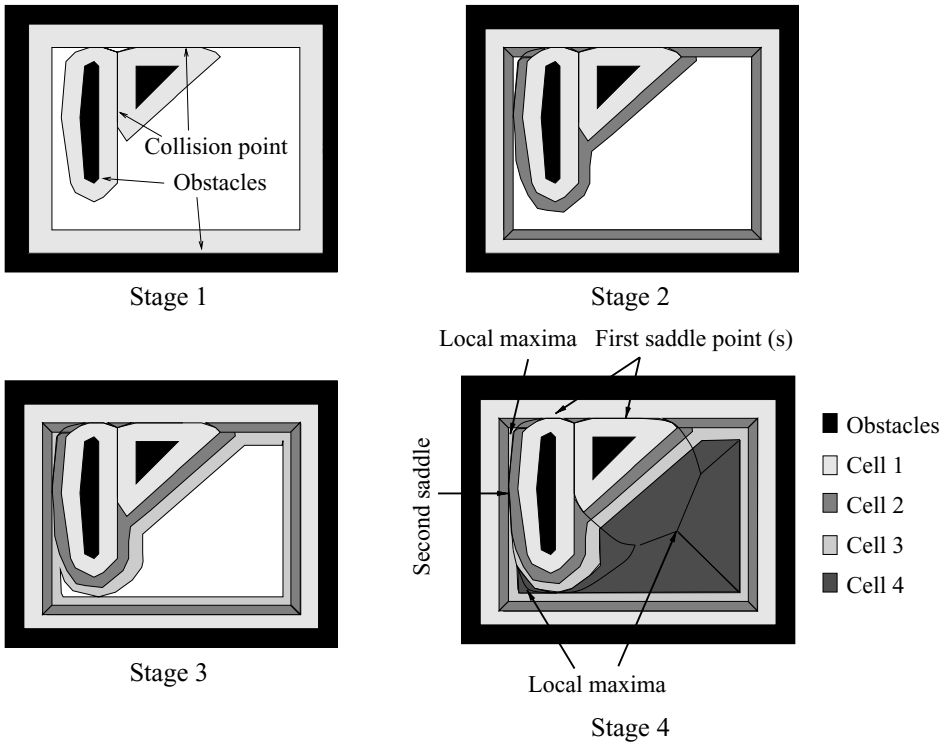



**Figure 6.16** Incremental construction of the cells of the brushfire decomposition. The wave fronts collide with each other at the points located on the GVD.

along the slice, motion orthogonal to the slice, and motion along the boundary of the cell (figure 6.17). For motion along the slice, the robot follows a path at a fixed distance from the nearest obstacle. The robot follows an obstacle boundary at a fixed distance until it returns to its starting point or a point where the distance to two obstacles becomes the same. When the robot returns to its starting point, it simply moves away from the closest obstacle by one width and repeats following the obstacle boundary at the new fixed distance. When the robot becomes doubly equidistant, it is on the boundary of the cell, at which point it follows the GVD. The robot follows the GVD until it reaches a point where the distance to the nearest obstacle is an integer multiple of the robot's diameter, at which point the robot then resumes obstacle boundary-following at this fixed distance. Here, the boundary-following motion is much different from before because cell boundaries lie exclusively in the free space with the exception of the first slice $\lambda = 0$.

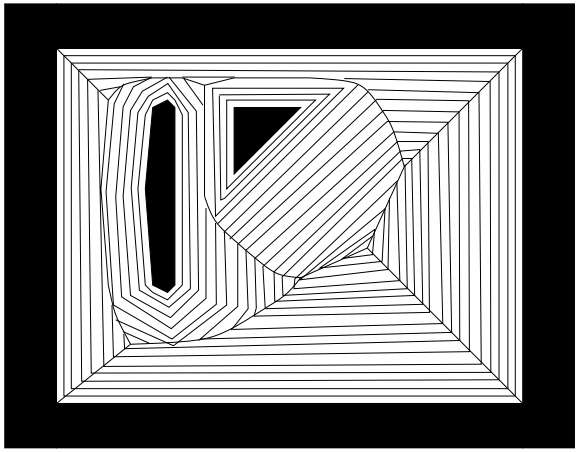


**Figure 6.17** Coverage pattern for the brushfire decomposition. To generate this pattern, the robot follows the boundaries of obstacles and thus it has a continuous robust reference for localization. Therefore this pattern is suitable for the robots that are prone to accrue dead-reckoning error. However, the robot relies heavily on long-range sensors.

The pattern induced by the brushfire algorithm is ideally suited for coverage with mobile robots experiencing dead-reckoning error but have a large sensing range. A mobile robot can follow this pattern servoing off of the boundaries of the obstacles by moving forward and maintaining a fixed distance from the boundary (figure 6.17). Since the robot is servoing off of range readings to the obstacle boundary, this method is insensitive to dead-reckoning error. This benefit, however, requires that the robot can indeed measure distance to the obstacle, which could be far away from the robot. This is in contrast to the boustrophedon decomposition approach which requires only very limited sensing range but which is sensitive to dead-reckoning error.

### Wave-Front Decomposition

Let $h(x, y)$ be the length of the shortest path between a point $(x, y)$ and a fixed location. The level sets $h^{-1}(\lambda)$ foliate the free space where for a given $\lambda$, the set of points in $h^{-1}(\lambda)$ are a distance $\lambda$ away from the fixed point in the free space. This particular function is sometimes called the wave-front potential, which was described for a discrete space in chapter 4, section 4.5. Imagine a wave front starting at $q_{\text{start}}$ and expanding into the free space. The value $\lambda$ parameterizes each wave front (or level set of $h$). Once the wave front crosses $q_{\text{goal}}$, the planner can backtrack a path from $q_{\text{goal}}$ to $q_{\text{start}}$ [208].

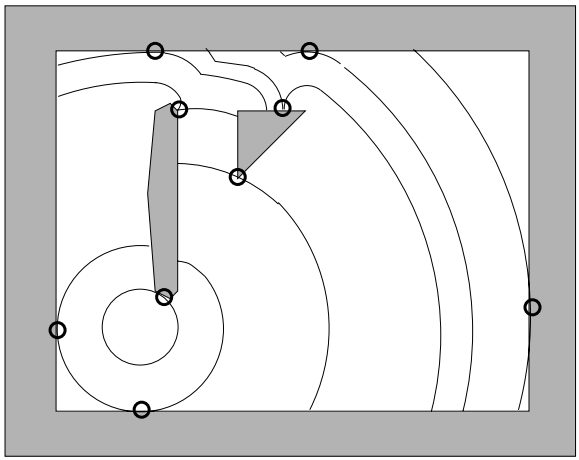


**Figure 6.18** Wave-front decomposition defined on a continuous domain. The wave front emanates from a point in the lower-left portion of the figure. Cusp points on the wave fronts originate from the critical points, e.g., the cusp point on the upper boundary of the obstacle located on the left.

The shortest path-length function induces a cell decomposition, as well. Critical points of this function occur both when wave fronts becomes tangent to obstacles and when wave fronts collide (figure 6.18). Note how once the waves collide, they propagate as one wave with a nonsmooth point that originated at the critical point. In fact, this nonsmooth point traces the set of points of equal pathlength to the goal for two classes of paths, one to the right of the obstacle and one to the left. This decomposition is especially useful for coverage by a tethered robot where the robot's tether is incrementally fed and the robot sweeps out curves each at constant tether length.

### 6.2.4 Sensor-Based Coverage

Now, let's place the robot in an unknown environment, but assume it has the standard range sensor ring as depicted in chapter 2, figures 2.5 and 2.16. The task is to simultaneously cover and explore the unknown space. This can be reduced to concurrently and incrementally covering each cell while constructing the adjacency graph. For sensor-based coverage, however, we incrementally construct a "dual" graph called a *Reeb graph* [154]. This graph is dual in the sense that the nodes of the Reeb graph are the critical points and the edges connect neighboring critical points, i.e., correspond to cells. For the sake of explanation, we limit discussion to Morse decompositions

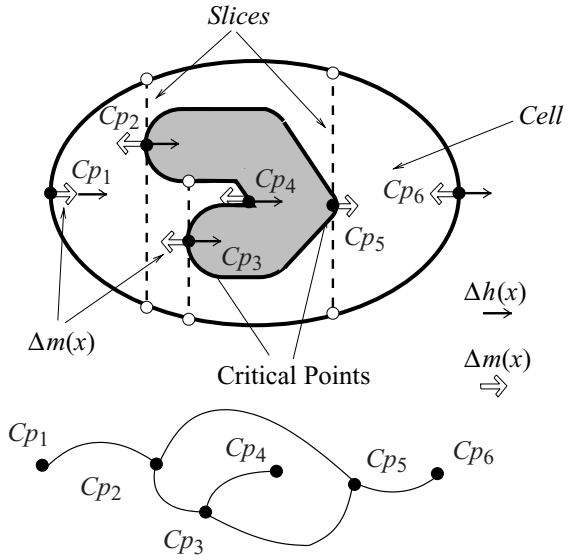



**Figure 6.19** A boustrophedon decomposition and its Reeb graph. At the critical points, the surface normals and sweep direction are parallel.

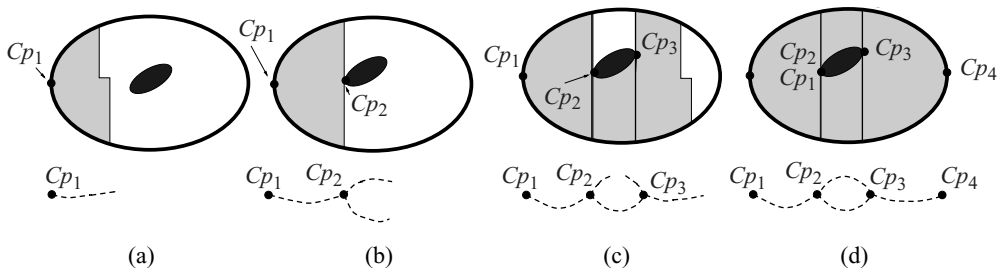
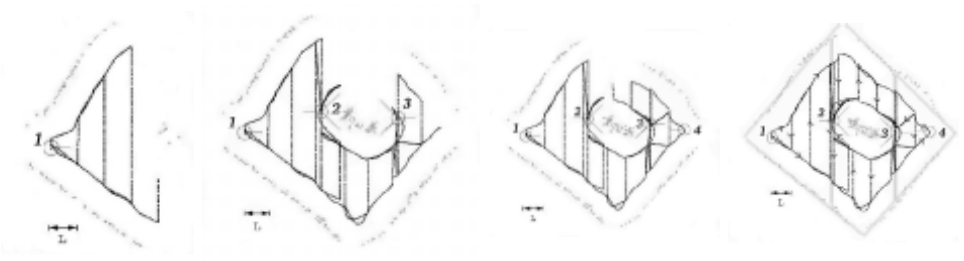



**Figure 6.20** Incremental construction of the graph while the robot is covering the space.

defined by $h(x, y) = x$, i.e., the boustrophedon decomposition. See figure 6.19 for an example of the boustrophedon decomposition and its corresponding Reeb graph.

The procedure of concurrently covering the cells and constructing the Reeb graph is depicted in figure 6.20. In figure 6.20(a), the robot starts to cover the space at the critical point $Cp_1$ and the planner instantiates an edge with only one node. When the robot is done covering the cell between $Cp_1$ and $Cp_2$, the planner joins their corresponding nodes with an edge in the graph representation (figure 6.20b). Now the



(a) (b) (c) (d)

**Figure 6.21** Four stages of coverage in an unknown environment with a robot-size detector on a Nomad Scout named RT Snookums. The coverage path followed by RT Snookums is shown by dotted black lines. We depict the critical points as light gray circles with lines emanating from them. The lines represent the directions of the corresponding adjacent cells. The robot incrementally constructs the graph representation by sensing the critical points 1, 2, 3, 4, 3, 2 (in the order of appearance) while covering the space. In the final stage (d), since all the critical points have explored edges, the robot concludes that it has completely covered the space. For the sake of discussion, we outlined the boundaries of the obstacles and cells in (d). The length scale $L = 0.53$ meters.

robot has two new uncovered cells. Since the space is *a priori* unknown, the planner arbitrarily chooses the lower cell to cover. When the robot reaches $Cp_3$, nodes of $Cp_2$ and $Cp_3$ become connected with an edge and the lower cell is completed (figure 6.20c). At $Cp_3$, the planner directs the robot to cover the cell to the right of $Cp_3$. When the robot senses $Cp_4$, it goes back to $Cp_3$ and starts to cover the upper cell. When the robot returns to $Cp_2$, the planner determines that all of the edges of all of the nodes (critical points) have been explored (figure 6.20d). Thus the planner concludes that the robot has completely covered the space. Figure 6.21 shows different stages of this incremental construction in an *a priori* unknown 2.75 meter $\times$ 3.65 meter room with a Nomad mobile robot that has a sonar ring.

Two details remain: How does the robot sense a critical point when it encounters one and how does the robot find all of the critical points? Critical point sensing is rather straightforward: the robot looks for points where the surface normals are parallel to the sweep direction. This is a direct consequence of lemma 5.5.2 in chapter 5, section 5.5.1. Here, we are looking for extrema of $h$ on the boundaries of the obstacles. Let $m$ implicitly represent a function whose preimage is the surface boundary. The matrix $D(h, m)(x)$ (chapter 5, section 5.5.1) then loses rank when $\nabla h(x)$ is parallel to $\nabla m(x)$, i.e., the slice normal is parallel to the surface normal (figure 6.19). This can easily be detected by looking at the global minimum of the range sensors in a range sensor ring.

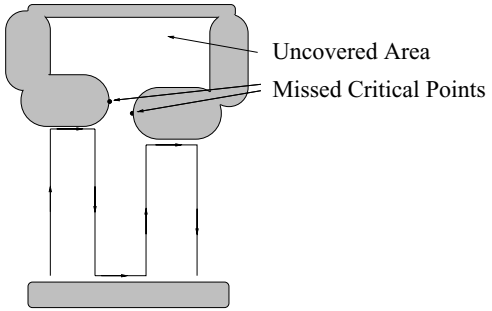



**Figure 6.22** Critical points in the ceiling are missed with conventional coverage algorithms.

The final challenge is to ensure that the robot encounters all critical points. Assume the robot starts to cover a cell at one of its defining critical points. While covering the cell, the robot looks for the other critical point that indicates complete coverage of the cell and the next node in the Reeb graph. We term this critical point the *closing critical point*. Since the Reeb graph is connected, the main challenge is to guarantee that the robot finds the closing critical point of each cell.

Most conventional coverage algorithms (e.g., [187, 190, 300]) miss the closing critical point because they perform the bulk of their coverage using a raster scan type of motion: move along a slice or lap to an obstacle, follow the obstacle boundary for a lateral distance equal to interlap spacing, and repeat. This alternates boundary-following between the "ceiling" and "floor" of the cell, as shown in figure 6.22. Unfortunately, this raster scan approach can miss the closing critical point of a cell. In figure 6.22, since the robot did not follow the boundary of the ceiling, it cannot sense the critical points in the ceiling using the critical point sensing method, described above. We may try to solve this problem by making the robot perform boundary-following along the ceiling in the reverse direction so that it will sense the critical points related to the ceiling. We call this motion *reverse boundary following*. However, reverse boundary following motion by itself is still not sufficient.

The robot must undergo additional motion to detect the closing critical point. We present an algorithm called the *cycle algorithm* [11] that ensures that the robot will find the closing critical point while performing coverage. For details, see [11]. Let $S_i$ be the start point of the cycle algorithm; $S_i$ is on (or near) the boundary of free space. From this point the robot looks for critical points via the following phases (figure 6.23):

1. *Forward phase*: The robot follows a slice, i.e., laps, until it encounters an obstacle. Then the robot follows the boundary of the obstacle in the forward sweep direction until either the robot moves laterally one lap width or until the robot encounters a critical point in the floor.

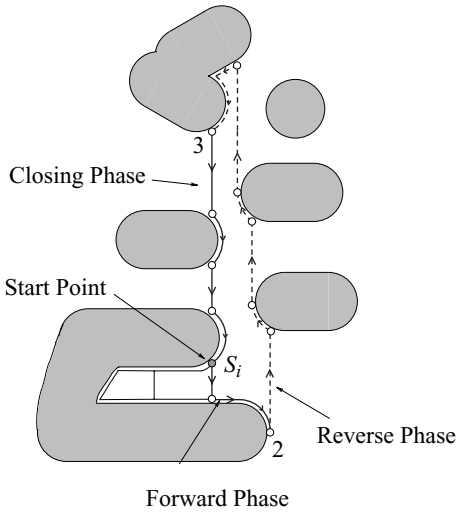



**Figure 6.23** In this particular arrangement of the obstacles, the robot executes every step of the cycle algorithm. The robot first follows the path between points $S_i$ and 2 during the forward phase. Then it follows the path between points 2 and 3 during the reverse phase. Finally, in the closing phase the robot follows the path between points 3 and $S_i$.

2. *Reverse phase*: The robot executes one or more laps in the reverse direction, intermixed with reverse boundary-following. Each reverse boundary-following operation terminates when the robot finds a critical point or when the aggregate lateral motion in the reverse direction is one lap width.

3. *Closing phase*: The robot executes one or more laps along the slice, possibly intermixed with boundary-following. Each boundary-following operation terminates when the robot encounters $S_i$ or the slice in which $S_i$ lies.

This algorithm is the most important part of the incremental construction. It guarantees encountering the closing critical point of a cell if it exists between subsequent laps.

### 6.2.5 Complexity of Coverage

We define complexity of coverage in two ways: first, we establish a relationship among the number of critical points, cells, and obstacles and second we determine an upper-bound on path length given the perimeter the obstacles and the diameter $\Delta$ smallest disk that circumscribes the space (figure 6.24). We limit our discussion to coverage with the boustrophedon decomposition. First, we establish a relationship between

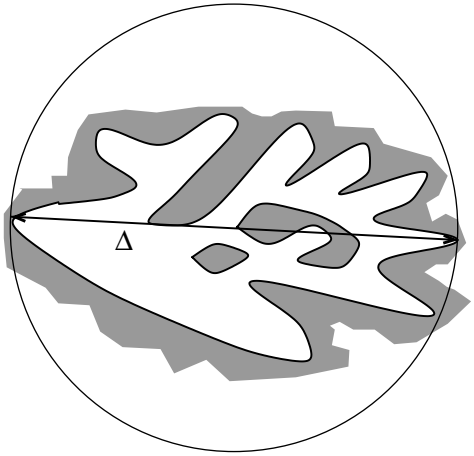



**Figure 6.24** To determine the complexity of the algorithm in terms of the environment size, we use the diameter $\Delta$ of the "minimal" disk that fully contains the space.

the number of cells, critical points and obstacles. The Reeb graph encodes the cells, critical points, and "obstacles." Note that obstacles (including the outer boundary) are represented with "faces"[3] in the graph. Graph theory uses *Euler's formula* to relate the number of nodes $v$, edges $e$ and faces $f$ of a planar connected graph [57] by

$$v - e + f = 2.$$

The nodes of the Reeb graph correspond to critical points, its edges represent the cells and its faces depict the obstacles. Moreover, the Reeb graph is connected and planar. Therefore we can use Euler's formula with one modification. Since the outer boundary of the space is, in general, not termed an obstacle, we subtract one from the number of faces to get the number of obstacles. Let $N_{cp}$ be the number of critical points, $N_{ce}$ be the number of cells and $N_{ob}$ be the number of obstacles (figure 6.25). Then

$$N_{ce} = N_{cp} + N_{ob} - 1.$$

This formula tells us that the number of cells increases *linearly* as the robot discovers new critical points.

Next, we calculate an upper bound on the total coverage path length. To simplify the calculation, we analyze lapping, boundary following and backtracking motions

3. A plane graph partitions the space into connected regions. Closures of these regions are called faces [57].

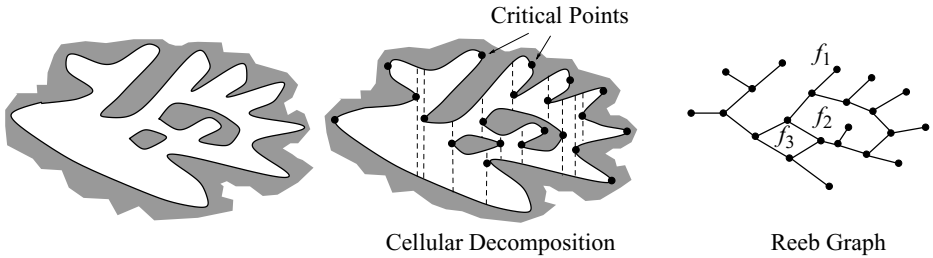



**Figure 6.25** In this decomposition example, there are twenty-one critical points (nodes in the graph), $N_{cp} = 21$, and two obstacles (faces $f_2$, $f_3$ in the graph; $f_1$ is the outer boundary), $N_{ob} = 2$. Using the modified Euler's formula $N_{ce} = N_{cp} + N_{ob} - 1$, there must be twenty-two cells (edges in the graph), $N_{ce} = 22$.

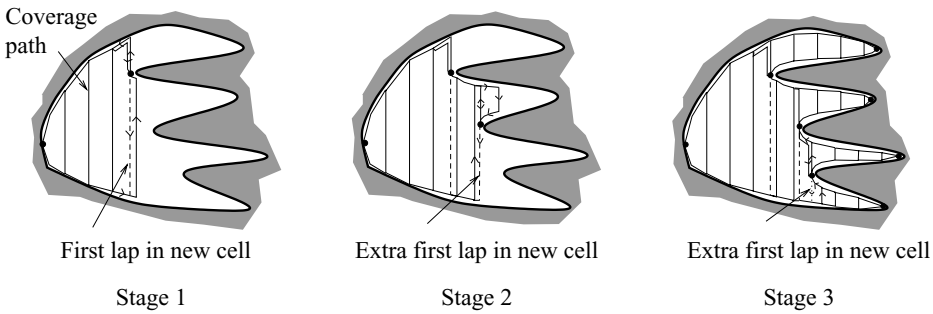



**Figure 6.26** When the robot starts to cover a new cell, it performs an "extra" lap starting from a critical point on one of its boundaries.

separately. Since the space is fully contained within a $\Delta$ diameter disk, the length of each lapping path can be at most $\Delta$. There must be at least $\lceil \frac{\Delta}{2r} \rceil$ lapping paths where $2r$ is the interlap spacing. However, often there is an additional lap associated with starting the coverage operation within a cell (figure 6.26). Hence, the maximum number of lapping paths is $\lceil \frac{\Delta}{2r} \rceil + N_{ce}$. Since the length of each lapping path is bounded above by $\Delta$, the total path length of the lapping motions is bounded above by $\Delta \lceil \frac{\Delta}{2r} \rceil + \Delta N_{ce}$.

Now we analyze the length of boundary-following paths. Let $P_{\text{cell}}$ be the length of the floor and ceiling of a cell. The coverage algorithm guarantees that the robot follows the entire floor and ceiling of a cell along the obstacle boundaries. Therefore, the length of boundary-following paths in a cell is at least $P_{\text{cell}}$. However, the robot, for each cycle, performs an undo-reverse boundary-following motion to get to the

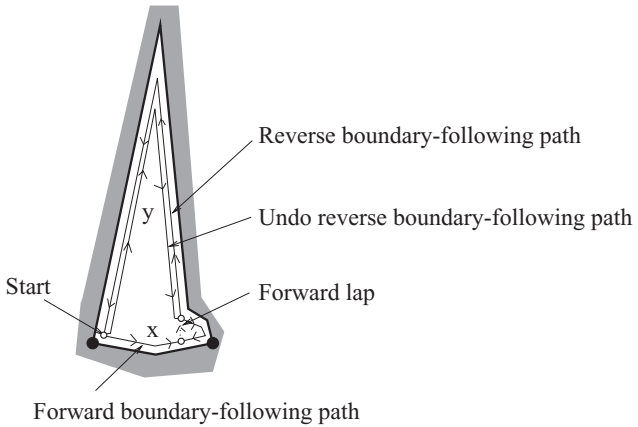



**Figure 6.27** The total perimeter of the cell is equal to $x + y$ where $y$ is the length of the floor and $x$ is the length of the ceiling and $x \gg y$. The total path length traveled along the boundary of the cell is bounded above by $2x + y$. In the worst case, as $x$ gets much larger than $y$, this value is equal to $2(x + y)$.

start point. Hence, the lower bound is $1.5P_{\text{cell}}$. In the worst case, the upper bound becomes $2P_{\text{cell}}$ (figure 6.27). Then, the total length of the boundary-following paths is less than $2P_{\text{total}}$ where $P_{\text{total}}$ is the length of the perimeter of all of the obstacles and the outer boundary.

After discovering the closing critical point of a cell, the robot backtracks to the closing critical point of a cell with uncovered cells associated with it by boundary-following and (if necessary) lapping (figure 6.28). In the worst case, the length of this backtracking path is $P_{\text{cell}} + \Delta$ (where the robot follows every boundary and the longest slice). When we consider all the backtracking paths, the upper bound becomes $P_{\text{total}} + \Delta N_{ce}$.

The robot starts to cover an uncovered cell from one of its defining critical points. While discovering this critical point by performing the cycle algorithm, the robot covers a small portion of the uncovered cell. The extra boundary-following path followed by the robot to discover the critical point is bounded above by $P_{\text{cell}}$. Hence, the total extra boundary-following path length is bounded above by $P_{\text{total}}$.

When the robot finishes covering a cell, it performs a depth-first search on the Reeb graph to choose an uncovered cell (if any are left) (figure 6.29). The robot reaches the uncovered cell by traversing the covered cells. To traverse a covered cell, the robot performs boundary-following and lapping motions as we explained in section 6.2.1. Within each covered cell, the total path length traveled is bounded above

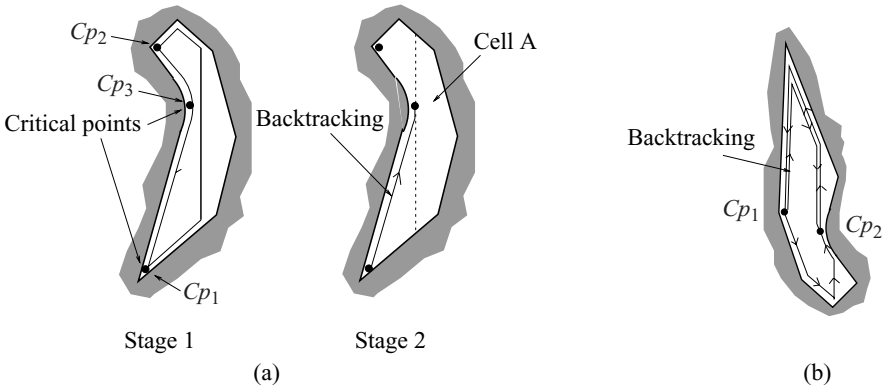



**Figure 6.28** (a) The robot starts to cover the space from $Cp_1$. Along the first cycle path, it discovers the critical points $Cp_1$, $Cp_2$, and $Cp_3$. The robot moves back to $Cp_3$ by following the boundary of the obstacle to start to cover cell A. (b) The robot travels back to $Cp_2$ from $Cp_1$ by boundary-following and lapping. Therefore, the length of the backtracking path is bounded above by $P_{\text{cell}} + D$ for each cell.

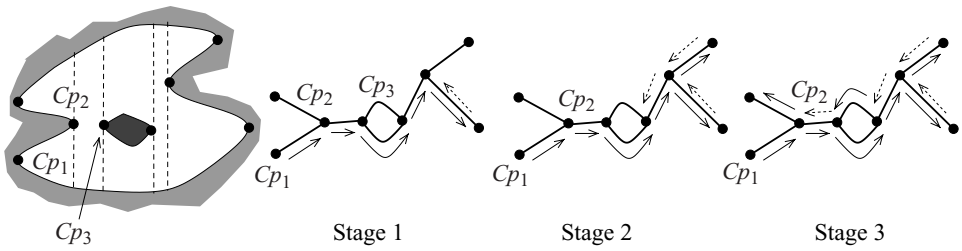



**Figure 6.29** The robot starts to cover the space from $Cp_1$. Whenever the robot finishes covering a cell, a depth-first search is performed on the graph to choose a new cell to cover. On the graph, solid arrows depict the coverage directions and dashed arrows correspond to backtracking directions. The depth-first search on the graph requires a maximum of $N_{ce}$ (number of edges) backtracking.

by $P_{\text{cell}} + 2\Delta$ (figure 6.30). Since we perform a depth-first search on the graph, each cell is traversed at most once [118], and therefore the backtracking path length is bounded by $\sum_{i=1}^{N_{ce}} P_{\text{cell}_i} + 2N_{ce}\Delta$ or $P_{\text{total}} + 2N_{ce}\Delta$.

Combining the above upper bounds, the length of the coverage path is less than

$$\frac{\Delta^2}{2r} + 4\Delta N_{ce} + 5P_{\text{total}},$$

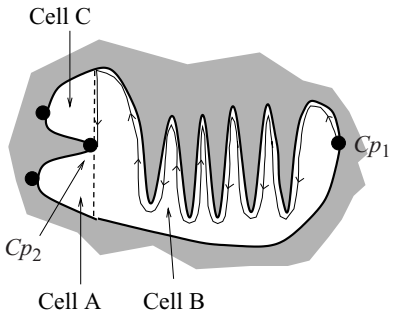



**Figure 6.30** After finishing covering cells A and B, the robot needs to travel from $Cp_1$ to $Cp_2$ to start to cover cell C. The robot simply follows the boundary of the obstacle either along the ceiling or floor of the cell. In the worst case, the boundary-following path length is bounded above by the length of the perimeter of the obstacles that form the boundary of the cell.

or using the modified Euler's formula,

$$\frac{\Delta^2}{2r} + 4\Delta(N_{cp} + N_{ob}) + 5P_{\text{total}} - 4\Delta.$$

Therefore, the total coverage path length is bounded linearly by the area of the space, the number of critical points, and the length of the perimeter of the obstacles and the outer boundary.

## 6.3 Visibility-Based Decompositions for Pursuit/Evasion

In the previous section, we used *connectivity changes* (i.e., critical points) to decompose the space into cells. The benefit of using the critical points is that the cells have a structure that is "easy" to cover. In this section, we use changes in line-of-sight related information to define cells. Such cells form *visibility-based decompositions.* Moving from one cell to another corresponds to a change in visibility, e.g., obstacle or target appears or disappears. We can use a visibility-based cell decomposition to address the *pursuit/evasion problem.* This problem, first introduced by Suzuki and Yamashita [403], considers one or more multiple agents called *pursuers* who are searching a bounded free space (usually polygonal) for a single agent called an *evader.* This evader can be a bad guy who is escaping the police or a trapped survivor wandering around a disaster site in need of help in searching rescue for workers. Lavalle, Guibas, and coworkers [172, 273] use a cell decomposition approach to address the pursuit/evasion problem. This decomposition lies in the *workspace,* not the configuration space, and the agents are points in the plane.

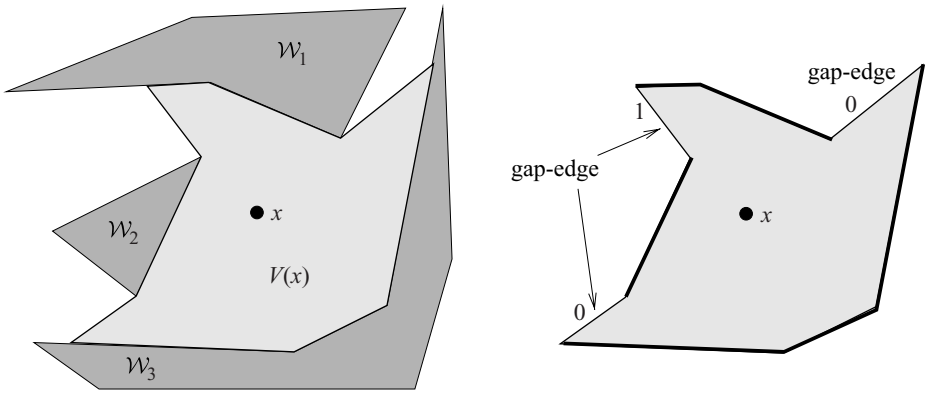



**Figure 6.31** The visibility polygon is shaded inside a polygonal world populated by obstacles. The gap-edges and their labels for a generic visibility polygon $V(x)$ are labeled. A clear edge has a 0 label, a contaminated edge has a 1 label. Here, $B(x) = \{010\}$.

In this description, we borrow terminology and notation from [172, 273]. The evader $e$ is caught when any one of the pursuers $\gamma^i$ becomes within line of sight with it, i.e., there exists an $i$ such that for all $\tau \in [0, 1]$, $\tau e + (1 - \tau)\gamma_i \in \mathcal{W}_{\text{free}}$. Let $e : [0, \infty) \rightarrow \mathcal{W}_{\text{free}}$ and $\gamma^i : [0, \infty) \rightarrow \mathcal{W}_{\text{free}}$ respectively be the paths that the evader and the $i$th pursuer follow. An evader is caught at the earliest time $t$ when there exists an $i$ such that for all $\tau \in [0, 1]$, $\tau e(t) + (1 - \tau)\gamma_i(t) \in \mathcal{W}_{\text{free}}$. Let us recast the capture condition once more: let $V(x) \subset \mathcal{W}_{\text{free}}$ be the star-shaped set of points that are within line of sight of $x$ (figure 6.31). An evader is caught if there exists an $i$ and $t$ such that $e(t) \in V(\gamma_i(t))$. A *motion strategy* is the collection of the pursuer paths $\gamma = \{\gamma_1, \ldots, \gamma_n\}$ and is termed a *solution strategy* if at least one pursuer catches the evader for all $e(t)$. Finally, let $H(\mathcal{W}_{\text{free}})$ be the minimum number of pursuers required to capture an evader in $\mathcal{W}_{\text{free}}$ in finite time.

We address the pursuit/evasion problem in two steps. Continuing to borrow terminology from [172, 273], we will define qualitatively important subsets of the free space and then use these subsets to define the decomposition. A region of $\mathcal{W}_{\text{free}}$ that *may* contain an evader is termed *contaminated*. If a region is not contaminated, then it is *clear.* However, a region that was contaminated, then cleared, and contaminated again is termed *recontaminated* (figure 6.32).

A visibility polygon now has two types of edges, those that lie on the boundary of obstacles and those that lie in the free space, which we call *gap-edges*. Gap-edges have a zero label if they bound a cleared region and a one if they bound a contaminated region (figure 6.31).

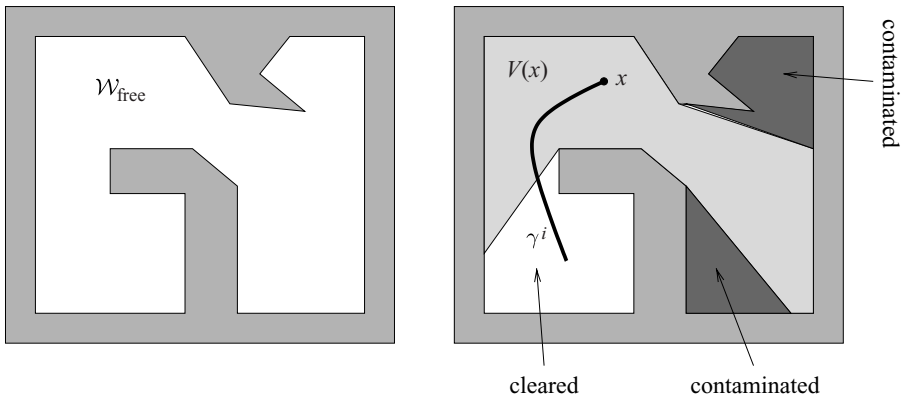



**Figure 6.32** A polygonal world with visibility polygon, cleared area, and contaminated area. A cleared area is a region we know the evader is not in. A contaminated region could contain the evader. The $i$th pursuer's path is also drawn.

Let $B(x)$ denote a binary vector of these gap-edge labels for a particular star-shaped set centered $x$. The pair $(x, B(x))$ denotes the *information state* and the set of all possible information states is the *information space.* A connected set $\nu \subset \mathcal{W}_{\text{free}}$ is *conservative* if for all $x \in \nu$, $B(x)$ remains fixed (figure 6.33). Finally, we can construct an adjacency graph for the conservative regions in a given environment. Hence, the conservative regions form an exact cell decomposition. (figure 6.34)

To construct the conservative regions for a polygonal environment, simply extend rays from each convex vertex of all of the obstacles until they intersect another obstacle. Also, if two vertices are within line of sight of each other, extend two rays, one from each vertex, but in the opposite directions. In other words, for $v_i \in \mathcal{WO}_i$ and $v_j \in \mathcal{WO}_j$, if $\lambda v_i + (1 - \lambda)v_j \in \mathcal{W}_{\text{free}}$ for all $\lambda \in (-\epsilon, 1 + \epsilon)$ for some $\epsilon > 0$, then extend a ray from $v_i$ away from $v_j$ and vice versa until they intersect an obstacle (figure 6.35).

This process forms a cell decomposition of the free space where each cell is a conservative region. This cell decomposition, however, is not sufficient to solve the pursuit/evasion problem. We have to form a cell decomposition in the information space [172,273]. To do this, first identify all of the "transitions" that can occur when an agent passes from one conservative cell to another in the free space decomposition (figure 6.36). If

1. a gap-edge disappears, do nothing;

2. a gap-edge appears, assign it a zero;

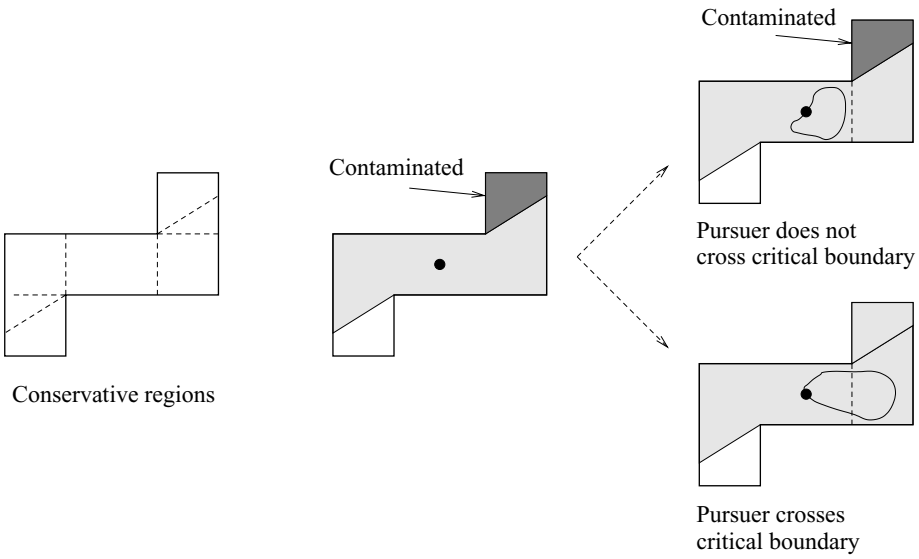



**Figure 6.33** The leftmost figure denotes all of the conservative regions separated by dashed lines. In the next figure, the pursuer starts off with a visibility polygon represented by a light gray area and a contaminated region by dark gray. If the pursuer crosses the critical boundary, then the contaminated region becomes cleared and the information state changes.

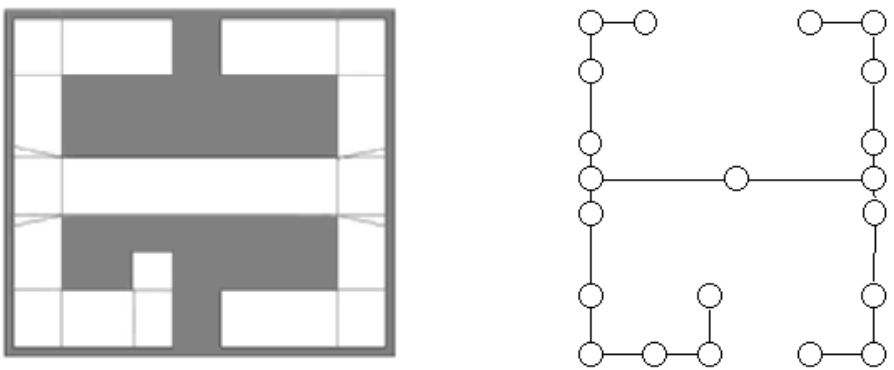


**Figure 6.34** Conservative regions and their corresponding adjacency graph.

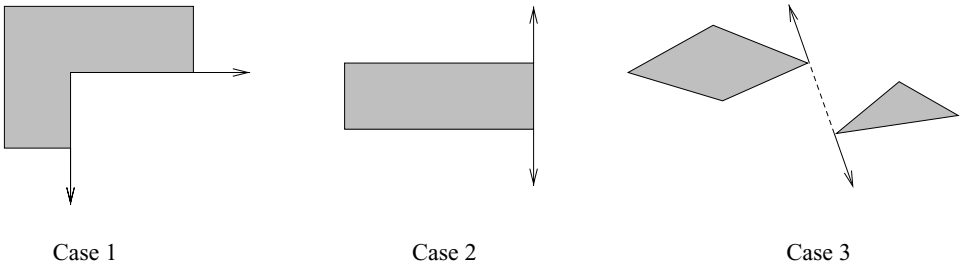



**Figure 6.35** The three conservative edge construction cases.

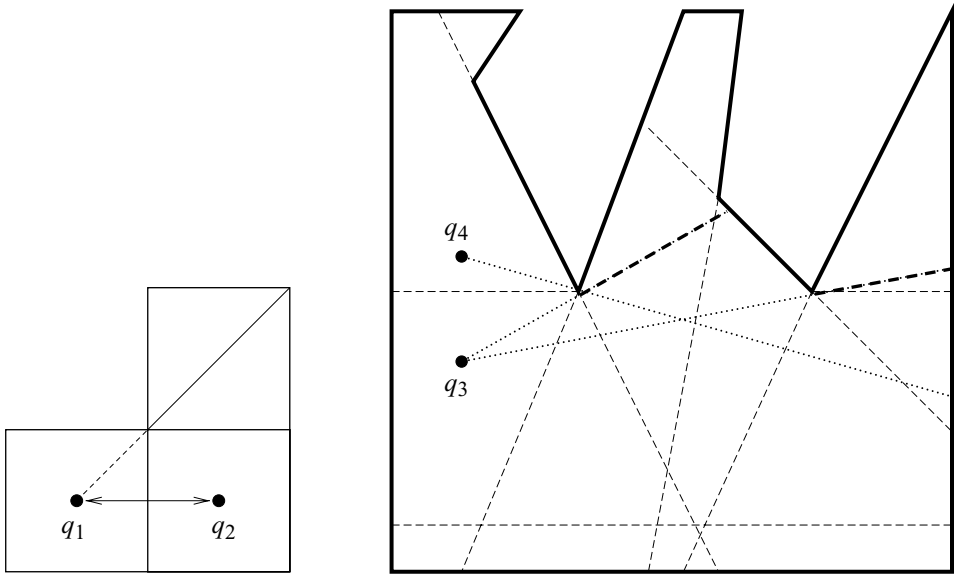



**Figure 6.36** (Left) A gap-edge appears or disappears (Right) Multiple gap-edges merge or a gap-edge divides. Moving from $q_3$ to $q_4$ causes two edges to merge into one (case 3). From $q_4$ to $q_3$, a single gap edge splits into two (case 4).

3. two or more gap-edges merge into one, if any of them had a one label, assign a one to the new edge;

4. a gap-edge divides into multiple gap-edges, assign the new edges the same label as the original;

This transition information serves as a basis for an adjacency graph for a new decomposition. This graph, called the information graph, can be used to solve the

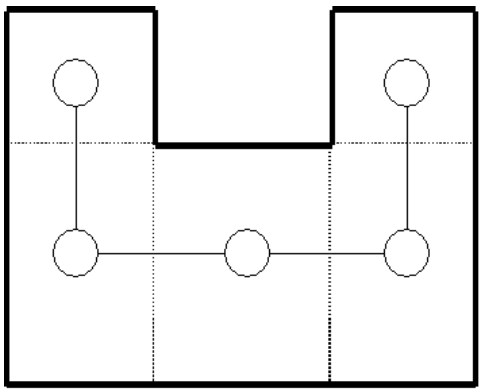


**Figure 6.37** A simple space with its corresponding adjacency graph overlaid on top.

pursuit/evasion problem. For each cell of the conservative region decomposition, we generate a sequence of nodes, each corresponding to a possible set of gap-edge labels. Figure 6.37 contains a simple free space with its adjacency graph overlaying on it. Consider the upper right conservative region. For all points in this region, it can only have one gap-edge which could have either a zero or a one label. Note that this gap-edge does not lie in the conservative region, i.e., it is *not* the horizontal line that separates the rightmost conservative regions.

Likewise, the conservative region in the lower-right cell has only one free edge. However, the transition from the upper-right cell to the lower-right cell is limited by the possible transition cases. In other words, if the upper-right cell has a $B(x) = 1$, then the lower-right cell must have $B(x) = 1$, and it cannot be zero. Therefore, the edges of the information graph represent the possible transitions from cell to cell (figure 6.38).

For a single pursuer and single evader in simply-connected spaces, a planner can start from any node in the information graph and then search for a node that has $B(x) = 0$ (a vector of zeros). This determines a path through the conservative regions that is guaranteed to catch the pursuer (figure 6.39).

Now, let's consider the number of pursuers required to find an evader. First, assume that $\mathcal{W}_{\text{free}}$ is a simply-connected polygon. Assume $\mathcal{W}_{\text{free}}$ is partitioned into $\mathcal{W}_{\text{free}1}$ and $\mathcal{W}_{\text{free}2}$ by connecting two vertices of the boundary of $\mathcal{W}_{\text{free}}$. Moreover, if $H(\mathcal{W}_{\text{free}1}) \leq k$, and $H(\mathcal{W}_{\text{free}2}) \leq k$, then $H(\mathcal{W}_{\text{free}}) \leq k + 1$, since $\mathcal{W}_{\text{free}}$ can be cleared by first clearing $\mathcal{W}_{\text{free}1}$ and $\mathcal{W}_{\text{free}2}$ successively using the same $k$ pursuers while keeping one pursuer, called the "static pursuer," at the common boundary between them.

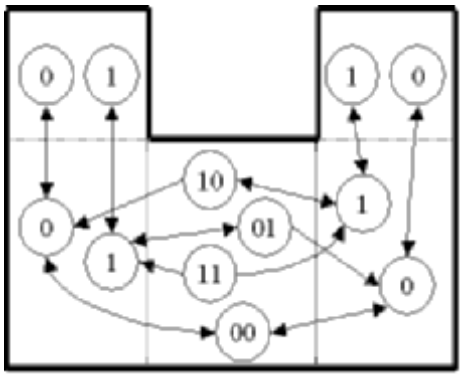



**Figure 6.38** The information graph $Gi$, with all possible routes through the graph shown. Each node in the information graph has all the possible gap-edge labels for the conservative region corresponding to each node.

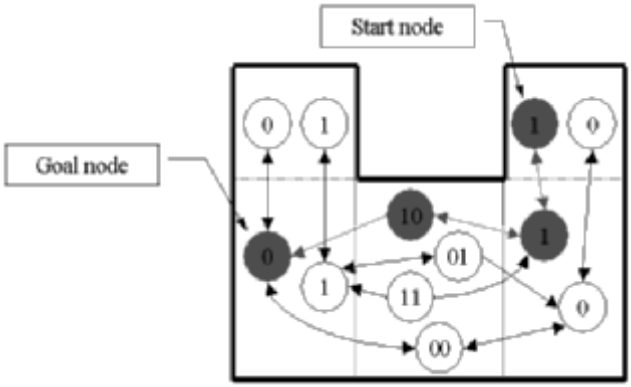



**Figure 6.39** The information graph $Gi$ from figure 6.38 with a solution path highlighted. Any node on the information graph with all zeros is a solution. Thus we use a graph search of our choice until we find a node of all zeros.

Since a simply-connected polygon can be partitioned into two pieces such that each component has at least one third of the edges of the original polygon, a simply-connected polygon can be triangulated by recursively connecting two vertices, and the "depth" of such a triangulation is at most $O(\log n)$. Therefore the original polygon can be cleared by clearing each triangle using one pursuer while keeping $O(\log n)$ static pursuers at each "level." Thus, at most $O(\log n)$ pursuers are required to clear a simply-connected polygon (figure 6.40).

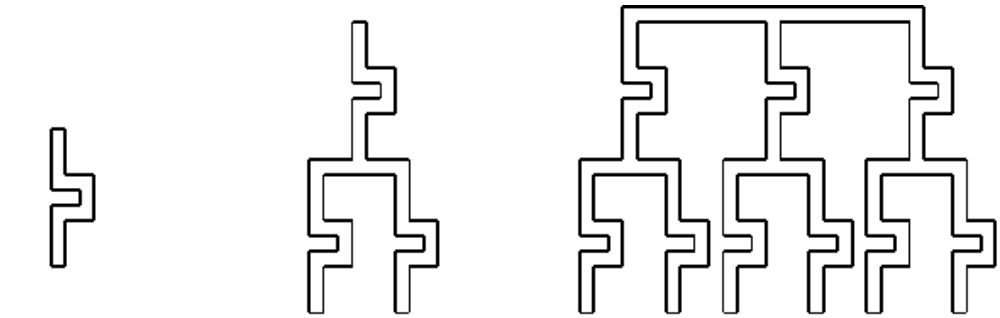
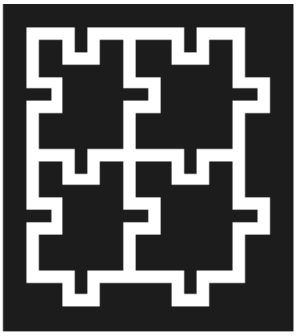



**Figure 6.40** Using a U-shaped space we can show how there are simply-connected free spaces that require order $(\log n)$ pursuers to explore. Each U-shape requires a single robot, and we can use another to divide the space into smaller and smaller sections.

Now consider a space with holes. Let $h$ be the number of the holes of a free space $\mathcal{W}_{\text{free}}$, and assume that $\mathcal{W}_{\text{free}}$ is triangulated. Let a *trichromatic triangle* be the triangle that touches three distinct connected components of $\mathcal{W}_{\text{free}}$. If all of the trichromatic triangles were removed, then $\mathcal{W}_{\text{free}}$ would be divided into (disconnected) simply-connected regions. The number of the trichromatic triangles can be determined by forming a graph that has the following properties. The vertices of the graph correspond to the holes of the space, and two of the vertices are connected if there is a trichromatic triangle that touches the boundary of the two holes corresponding to these vertices. Since this graph is planar, it can be shown that the number of edges of this graph and therefore the number of the trichromatic triangles is $O(h)$.

Now consider the dual graph of the triangulation of $\mathcal{W}_{\text{free}}$, but actually consider only the vertices corresponding to the trichromatic triangles. Note that each edge of this graph corresponds to a simply-connected region of $\mathcal{W}_{\text{free}}$, which can be cleared using $O(\log n)$ pursuers using the result above. This graph can be partitioned using $O(\sqrt{h})$ edges into two components so that each component has at least one third of the edges. The $O(\sqrt{h})$ "static" pursuers are placed on the edges that partition the graph. Recursively applying the planar graph separator theorem, and placing static pursuers accordingly (thus total number of static pursuer is $O(\sqrt{h} + \sqrt{2/3h} + \sqrt{4/9h}) = O(\sqrt{h})$, a simply-connected region (i.e., an edge of the dual graph) can be isolated. Since a simply-connected region can be cleared using $O(\log n)$ pursuers, the complete region can be cleared using $O(\sqrt{h} + \log n)$ pursuers (figure 6.41).



4 holes, 111 edges, 4 pursuers

**Figure 6.41** An example of a space that requires $O(\sqrt{h} + \log n)$ pursuers, where $n$ is the number of edges and $h$ is the number of holes. This example requires four pursuers. Three pursuers are used to divide the space into simply-connected regions, while the other robot searches.

## Problems

1. At the end of section 6.1, we describe a method for locating a single query. Why is this important if it takes less time to locate a single query than to construct the search graph?
2. Write a program that determines a path for a planar convex translation-only robot from a start to final configuration using the trapezoidal decomposition. Input from a file a robot and from a separate file a set of obstacles in a known workspace. Input a start and goal configuration from the keyboard. Use the configuration space generator from chapter 3. Hand in meaningful output.
3. Describe a generalization of the trapezoidal decomposition in three dimensions. What do the cells look like?
4. Consider the trapezoidal decomposition and adjacency graph in figure 6.2. Using the method described in section 6.1, determine a path when $q_{\text{start}}$ is in $c_5$ and $q_{\text{goal}}$ is in $c_6$.
5. Describe a generalization of the boustrophedon decomposition in three dimensions. What do the cells look like?
6. How does the solution to the pursuit/evasion problem change if there are multiple evaders?
7. The solution for the pursuit/evasion problem is described in the workspace, not the configuration space. Why?

8. Figure 6.35 shows the three cases that are used to construct the cell decomposition of conservative cells. Show that the solution to the pursuit/evasion problem, described above, works correctly without case 2. Hint: Show that there are no critical changes in gaps for case 2, except where a gap is "anchored." Note that removing case 2 does, however, lead to concave cells. Show the new decomposition for the workspace in figure 6.34.

9. Adapt the pursuit/evasion problem to the case where the pursuer has a limited field-of-view sensor, then has a limited range sensor, and then a limited range limited field-of-view sensor.