Chapter 6

Combinatorial Motion Planning

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to approximations. Due to this property, they are alternatively referred to as *exact* algorithms. This is in contrast to the sampling-based motion planning algorithms from Chapter 5.

6.1 Introduction

All of the algorithms presented in this chapter are *complete*, which means that for any problem instance (over the space of problems for which the algorithm is designed), the algorithm will either find a solution or will correctly report that no solution exists. By contrast, in the case of sampling-based planning algorithms, weaker notions of completeness were tolerated: resolution completeness and probabilistic completeness.

Representation is important When studying combinatorial motion planning algorithms, it is important to carefully consider the definition of the input. What is the representation used for the robot and obstacles? What set of transformations may be applied to the robot? What is the dimension of the world? Are the robot and obstacles convex? Are they piecewise linear? The specification of possible inputs defines a set of problem instances on which the algorithm will operate. If the instances have certain convenient properties (e.g., low dimensionality, convex models), then a combinatorial algorithm may provide an elegant, practical solution. If the set of instances is too broad, then a requirement of both completeness and practical solutions may be unreasonable. Many general formulations of general motion planning problems are PSPACE-hard[1]; therefore, such a hope appears unattainable. Nevertheless, there exist general, complete motion planning algorithms. Note that focusing on the representation is the opposite philosophy from sampling-based planning, which hides these issues in the collision detection module.

[1]This implies NP-hard. An overview of such complexity statements appears in Section 6.5.1.

Reasons to study combinatorial methods There are generally two good reasons to study combinatorial approaches to motion planning:

1. In many applications, one may only be interested in a special class of planning problems. For example, the world might be 2D, and the robot might only be capable of translation. For many special classes, elegant and efficient algorithms can be developed. These algorithms are complete, do not depend on approximation, and can offer much better performance than sampling-based planning methods, such as those in Chapter 5.

2. It is both interesting and satisfying to know that there are complete algorithms for an extremely broad class of motion planning problems. Thus, even if the class of interest does not have some special limiting assumptions, there still exist general-purpose tools and algorithms that can solve it. These algorithms also provide theoretical upper bounds on the time needed to solve motion planning problems.

Warning: Some methods are impractical Be careful not to make the wrong assumptions when studying the algorithms of this chapter. A few of them are efficient and easy to implement, but many might be neither. Even if an algorithm has an amazing asymptotic running time, it might be close to impossible to implement. For example, one of the most famous algorithms from computational geometry can split a simple[2] polygon into triangles in $O(n)$ time for a polygon with n edges [190]. This is so amazing that it was covered in the *New York Times*, but the algorithm is so complicated that it is doubtful that anyone will ever implement it. Sometimes it is preferable to use an algorithm that has worse theoretical running time but is much easier to understand and implement. In general, though, it is valuable to understand both kinds of methods and decide on the trade-offs for yourself. It is also an interesting intellectual pursuit to try to determine how efficiently a problem can be solved, even if the result is mainly of theoretical interest. This might motivate others to look for simpler algorithms that have the same or similar asymptotic running times.

Roadmaps Virtually all combinatorial motion planning approaches construct a *roadmap* along the way to solving queries. This notion was introduced in Section 5.6, but in this chapter stricter requirements are imposed in the roadmap definition because any algorithm that constructs one needs to be complete. Some of the algorithms in this chapter first construct a cell decomposition of $\mathcal{C}_{free}$ from which the roadmap is consequently derived. Other methods directly construct a roadmap without the consideration of cells.

Let $\mathcal{G}$ be a topological graph (defined in Example 4.6) that maps into $\mathcal{C}_{free}$. Furthermore, let $S \subset \mathcal{C}_{free}$ be the swath, which is set of all points reached by $\mathcal{G}$, as defined in (5.40). The graph $\mathcal{G}$ is called a *roadmap* if it satisfies two important conditions:

[2] A polygonal region that has no holes.

1. **Accessibility:** From any $q \in \mathcal{C}_{free}$, it is simple and efficient to compute a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q$ and $\tau(1) = s$, in which s may be any point in S. Usually, s is the closest point to q, assuming $\mathcal{C}$ is a metric space.

2. **Connectivity-preserving:** Using the first condition, it is always possible to connect some q_I and q_G to some s_1 and s_2, respectively, in S. The second condition requires that if there exists a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_I$ and $\tau(1) = q_G$, then there also exists a path $\tau' : [0,1] \rightarrow S$, such that $\tau'(0) = s_1$ and $\tau'(1) = s_2$. Thus, solutions are not missed because $\mathcal{G}$ fails to capture the connectivity of $\mathcal{C}_{free}$. This ensures that complete algorithms are developed.

By satisfying these properties, a roadmap provides a discrete representation of the continuous motion planning problem without losing any of the original connectivity information needed to solve it. A query, (q_I, q_G), is solved by connecting each query point to the roadmap and then performing a discrete graph search on $\mathcal{G}$. To maintain completeness, the first condition ensures that any query can be connected to $\mathcal{G}$, and the second condition ensures that the search always succeeds if a solution exists.

6.2 Polygonal Obstacle Regions

Rather than diving into the most general forms of combinatorial motion planning, it is helpful to first see several methods explained for a case that is easy to visualize. Several elegant, straightforward algorithms exist for the case in which $\mathcal{C} = \mathbb{R}^2$ and $\mathcal{C}_{obs}$ is polygonal. Most of these cannot be directly extended to higher dimensions; however, some of the general principles remain the same. Therefore, it is very instructive to see how combinatorial motion planning approaches work in two dimensions. There are also applications where these algorithms may directly apply. One example is planning for a small mobile robot that may be modeled as a point moving in a building that can be modeled with a 2D polygonal floor plan.

After covering representations in Section 6.2.1, Sections 6.2.2–6.2.4 present three different algorithms to solve the same problem. The one in Section 6.2.2 first performs *cell decomposition* on the way to building the roadmap, and the ones in Sections 6.2.3 and 6.2.4 directly produce a roadmap. The algorithm in Section 6.2.3 computes maximum clearance paths, and the one in Section 6.2.4 computes shortest paths (which consequently have no clearance).

6.2.1 Representation

Assume that $\mathcal{W} = \mathbb{R}^2$; the obstacles, $\mathcal{O}$, are polygonal; and the robot, $\mathcal{A}$, is a polygonal body that is only capable of translation. Under these assumptions, $\mathcal{C}_{obs}$ will be polygonal. For the special case in which $\mathcal{A}$ is a point in $\mathcal{W}$, $\mathcal{O}$ maps directly

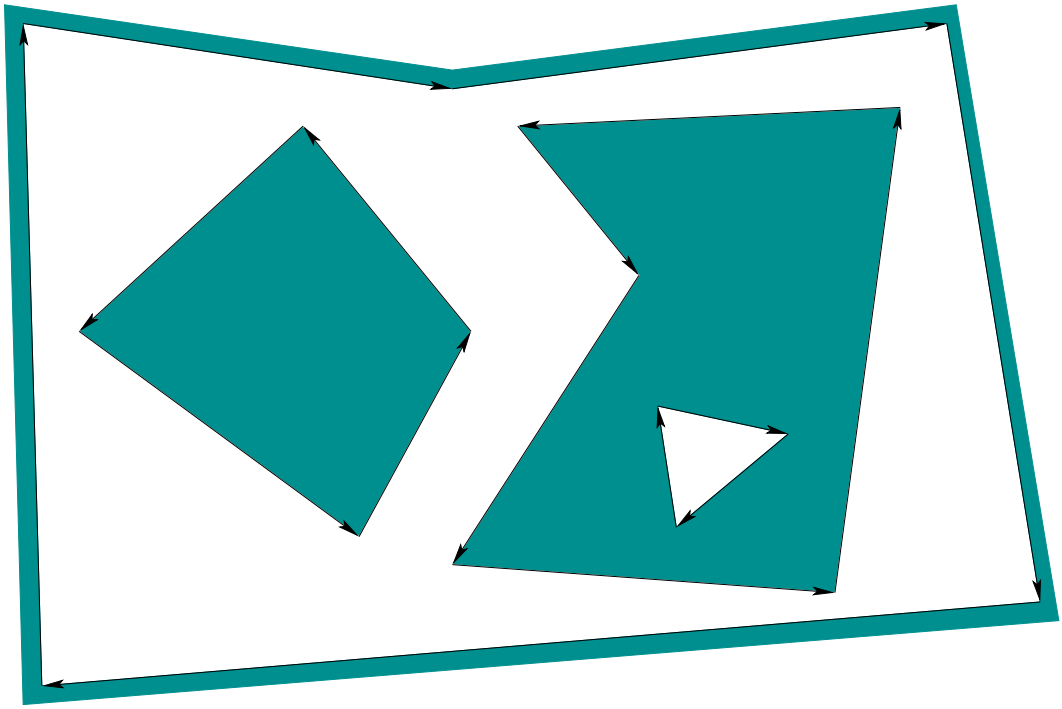


Figure 6.1: A polygonal model specified by four oriented simple polygons.

to $\mathcal{C}_{obs}$ without any distortion. Thus, the problems considered in this section may also be considered as planning for a *point robot*. If $\mathcal{A}$ is not a point robot, then the Minkowski difference, (4.37), of $\mathcal{O}$ and $\mathcal{A}$ must be computed. For the case in which both $\mathcal{A}$ and each component of $\mathcal{O}$ are convex, the algorithm in Section 4.3.2 can be applied to compute each component of $\mathcal{C}_{obs}$. In general, both $\mathcal{A}$ and $\mathcal{O}$ may be nonconvex. They may even contain holes, which results in a $\mathcal{C}_{obs}$ model such as that shown in Figure 6.1. In this case, $\mathcal{A}$ and $\mathcal{O}$ may be decomposed into convex components, and the Minkowski difference can be computed for each pair of components. The decompositions into convex components can actually be performed by adapting the cell decomposition algorithm that will be presented in Section 6.2.2. Once the Minkowski differences have been computed, they need to be merged to obtain a representation that can be specified in terms of simple polygons, such as those in Figure 6.1. An efficient algorithm to perform this merging is given in Section 2.4 of [264]. It can also be based on many of the same principles as the planning algorithm in Section 6.2.2.

To implement the algorithms described in this section, it will be helpful to have a data structure that allows convenient access to the information contained in a model such as Figure 6.1. How is the outer boundary represented? How are holes inside of obstacles represented? How do we know which holes are inside of which obstacles? These questions can be efficiently answered by using the doubly connected edge list data structure, which was described in Section 3.1.3 for consistent labeling of polyhedral faces. We will need to represent models, such

as the one in Figure 6.1, and any other information that planning algorithms need to maintain during execution. There are three different records:

Vertices: Every vertex v contains a pointer to a point $(x, y) \in \mathcal{C} = \mathbb{R}^2$ and a pointer to some half-edge that has v as its origin.

Faces: Every face has one pointer to a half-edge on the boundary that surrounds the face; the pointer value is NIL if the face is the outermost boundary. The face also contains a list of pointers for each connected component (i.e., hole) that is contained inside of that face. Each pointer in the list points to a half-edge of the component's boundary.

Half-edges: Each half-edge is directed so that the obstacle portion is always to its left. It contains five different pointers. There is a pointer to its *origin vertex*. There is a *twin* half-edge pointer, which may point to a half-edge that runs in the opposite direction (see Section 3.1.3). If the half-edge borders an obstacle, then this pointer is NIL. Half-edges are always arranged in circular chains to form the boundary of a face. Such chains are oriented so that the obstacle portion (or a twin half-edge) is always to its left. Each half-edge stores a pointer to its internal face. It also contains pointers to the next and previous half-edges in the circular chain of half-edges.

For the example in Figure 6.1, there are four circular chains of half-edges that each bound a different face. The face record of the small triangular hole points to the obstacle face that contains the hole. Each obstacle contains a pointer to the face represented by the outermost boundary. By consistently assigning orientations to the half-edges, circular chains that bound an obstacle always run counterclockwise, and chains that bound holes run clockwise. There are no twin half-edges because all half-edges bound part of $\mathcal{C}_{obs}$. The doubly connected edge list data structure is general enough to allow extra edges to be inserted that slice through $\mathcal{C}_{free}$. These edges will not be on the border of $\mathcal{C}_{obs}$, but they can be managed using twin half-edge pointers. This will be useful for the algorithm in Section 6.2.2.

6.2.2 Vertical Cell Decomposition

Cell decompositions will be defined formally in Section 6.3, but here we use the notion informally. Combinatorial methods must construct a finite data structure that exactly encodes the planning problem. Cell decomposition algorithms achieve this partitioning of $\mathcal{C}_{free}$ into a finite set of regions called *cells*. The term *k-cell* refers to a k-dimensional cell. The cell decomposition should satisfy three properties:

1. Computing a path from one point to another inside of a cell must be trivially easy. For example, if every cell is convex, then any pair of points in a cell can be connected by a line segment.

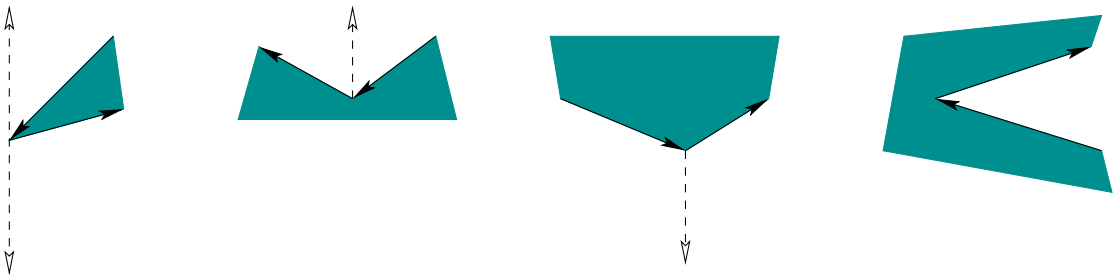


Figure 6.2: There are four general cases: 1) extending upward and downward, 2) upward only, 3) downward only, and 4) no possible extension.

2. Adjacency information for the cells can be easily extracted to build the roadmap.

3. For a given q_I and q_G, it should be efficient to determine which cells contain them.

If a cell decomposition satisfies these properties, then the motion planning problem is reduced to a graph search problem. Once again the algorithms of Section 2.2 may be applied; however, in the current setting, the entire graph, $\mathcal{G}$, is usually known in advance.[3] This was not assumed for discrete planning problems.

Defining the vertical decomposition We next present an algorithm that constructs a *vertical cell decomposition* [189], which partitions $\mathcal{C}_{free}$ into a finite collection of 2-cells and 1-cells. Each 2-cell is either a trapezoid that has vertical sides or a triangle (which is a degenerate trapezoid). For this reason, the method is sometimes called *trapezoidal decomposition*. The decomposition is defined as follows. Let P denote the set of vertices used to define $\mathcal{C}_{obs}$. At every $p \in P$, try to extend rays upward and downward through $\mathcal{C}_{free}$, until $\mathcal{C}_{obs}$ is hit. There are four possible cases, as shown in Figure 6.2, depending on whether or not it is possible to extend in each of the two directions. If $\mathcal{C}_{free}$ is partitioned according to these rays, then a vertical decomposition results. Extending these rays for the example in Figure 6.3a leads to the decomposition of $\mathcal{C}_{free}$ shown in Figure 6.3b. Note that only trapezoids and triangles are obtained for the 2-cells in $\mathcal{C}_{free}$.

Every 1-cell is a vertical segment that serves as the border between two 2-cells. We must ensure that the topology of $\mathcal{C}_{free}$ is correctly represented. Recall that $\mathcal{C}_{free}$ was defined to be an open set. Every 2-cell is actually defined to be an open set in $\mathbb{R}^2$; thus, it is the interior of a trapezoid or triangle. The 1-cells are the interiors of segments. It is tempting to make 0-cells, which correspond to the endpoints of segments, but these are not allowed because they lie in $\mathcal{C}_{obs}$.

[3]Exceptions to this are some algorithms mentioned in Section 6.5.3, which obtain greater efficiency by only maintaining one connected component of $\mathcal{C}_{obs}$.

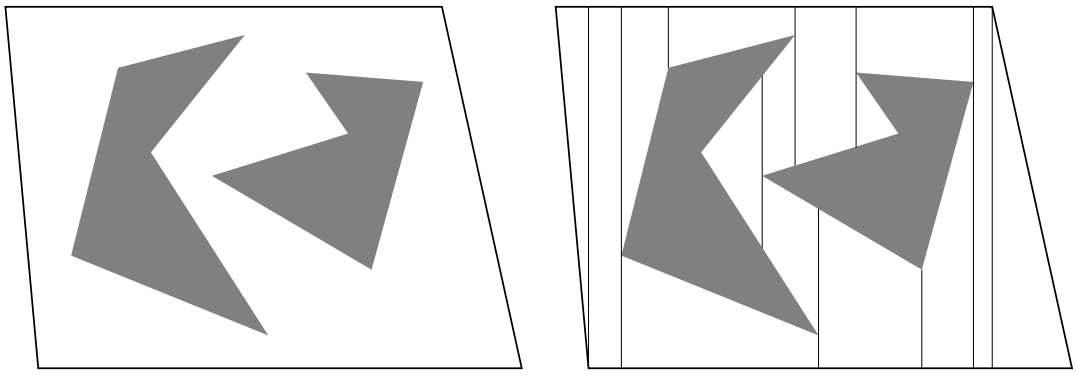


Figure 6.3: The vertical cell decomposition method uses the cells to construct a roadmap, which is searched to yield a solution to a query.

General position issues What if two points along $\mathcal{C}_{obs}$ lie on a vertical line that slices through $\mathcal{C}_{free}$? What happens when one of the edges of $\mathcal{C}_{obs}$ is vertical? These are special cases that have been ignored so far. Throughout much of combinatorial motion planning it is common to ignore such special cases and assume $\mathcal{C}_{obs}$ is in *general position*. This usually means that if all of the data points are perturbed by a small amount in some random direction, the probability that the special case remains is zero. Since a vertical edge is no longer vertical after being slightly perturbed, it is not in general position. The general position assumption is usually made because it greatly simplifies the presentation of an algorithm (and, in some cases, its asymptotic running time is even lower). In practice, however, this assumption can be very frustrating. Most of the implementation time is often devoted to correctly handling such special cases. Performing random perturbations may avoid this problem, but it tends to unnecessarily complicate the solutions. For the vertical decomposition, the problems are not too difficult to handle without resorting to perturbations; however, in general, it is important to be aware of this difficulty, which is not as easy to fix in most other settings.

Defining the roadmap To handle motion planning queries, a roadmap is constructed from the vertical cell decomposition. For each cell C_i, let q_i denote a designated *sample point* such that $q_i \in C_i$. The sample points can be selected as the cell centroids, but the particular choice is not too important. Let $\mathcal{G}(V, E)$ be a topological graph defined as follows. For every cell, C_i, define a vertex $q_i \in V$. There is a vertex for every 1-cell and every 2-cell. For each 2-cell, define an edge from its sample point to the sample point of every 1-cell that lies along its boundary. Each edge is a line-segment path between the sample points of the cells. The resulting graph is a roadmap, as depicted in Figure 6.4. The accessibility condition is satisfied because every sample point can be reached by a straight-line path thanks to the convexity of every cell. The connectivity condition is also satisfied

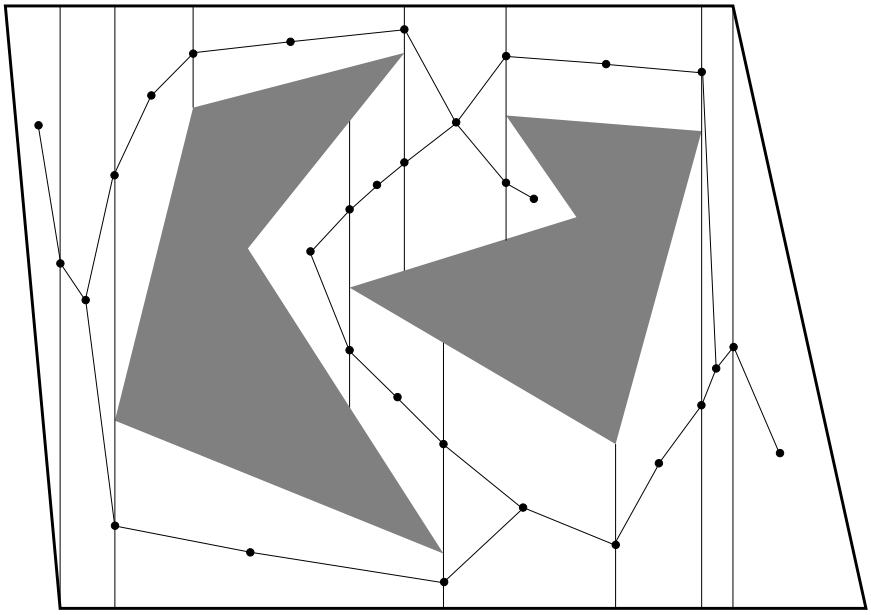


Figure 6.4: The roadmap derived from the vertical cell decomposition.

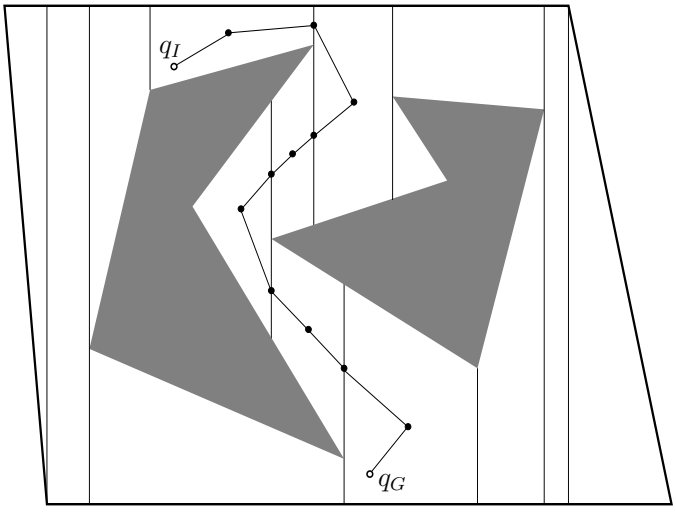


Figure 6.5: An example solution path.

because $\mathcal{G}$ is derived directly from the cell decomposition, which also preserves the connectivity of $\mathcal{C}_{free}$. Once the roadmap is constructed, the cell information is no longer needed for answering planning queries.

Solving a query Once the roadmap is obtained, it is straightforward to solve a motion planning query, (q_I, q_G). Let C_0 and C_k denote the cells that contain q_I and q_G, respectively. In the graph $\mathcal{G}$, search for a path that connects the sample point of C_0 to the sample point of C_k. If no such path exists, then the planning algorithm correctly declares that no solution exists. If one does exist, then let C_1, C_2, ..., C_{k-1} denote the sequence of 1-cells and 2-cells visited along the computed path in $\mathcal{G}$ from C_0 to C_k.

A solution path can be formed by simply "connecting the dots." Let q_0, q_1, q_2, ..., q_{k-1}, q_k, denote the sample points along the path in $\mathcal{G}$. There is one sample point for every cell that is crossed. The solution path, $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$, is formed by setting $\tau(0) = q_I$, $\tau(1) = q_G$, and visiting each of the points in the sequence from q_0 to q_k by traveling along the shortest path. For the example, this leads to the solution shown in Figure 6.5. In selecting the sample points, it was important to ensure that each path segment from the sample point of one cell to the sample point of its neighboring cell is collision-free.[4]

Computing the decomposition The problem of efficiently computing the decomposition has not yet been considered. Without concern for efficiency, the problem appears simple enough that all of the required steps can be computed by brute-force computations. If $\mathcal{C}_{obs}$ has n vertices, then this approach would take at least $O(n^2)$ time because intersection tests have to be made between each vertical ray and each segment. This even ignores the data structure issues involved in finding the cells that contain the query points and in building the roadmap that holds the connectivity information. By careful organization of the computation, it turns out that all of this can be nicely handled, and the resulting running time is only $O(n \lg n)$.

Plane-sweep principle The algorithm is based on the *plane-sweep* (or *line-sweep*) principle from computational geometry [129, 264, 302], which forms the basis of many combinatorial motion planning algorithms and many other algorithms in general. Much of computational geometry can be considered as the development of data structures and algorithms that generalize the sorting problem to multiple dimensions. In other words, the algorithms carefully "sort" geometric information.

The word "sweep" is used to refer to these algorithms because it can be imagined that a line (or plane, etc.) sweeps across the space, only to stop where some

[4]This is the reason why the approach is defined differently from Chapter 1 of [588]. In that case, sample points were not placed in the interiors of the 2-cells, and collision could result for some queries.

critical change occurs in the information. This gives the intuition, but the sweeping line is not explicitly represented by the algorithm. To construct the vertical decomposition, imagine that a vertical line sweeps from $x = -\infty$ to $x = \infty$, using (x, y) to denote a point in $\mathcal{C} = \mathbb{R}^2$.

From Section 6.2.1, note that the set P of $\mathcal{C}_{obs}$ vertices are the only data in $\mathbb{R}^2$ that appear in the problem input. It therefore seems reasonable that interesting things can only occur at these points. Sort the points in P in increasing order by their X coordinate. Assuming general position, no two points have the same X coordinate. The points in P will now be visited in order of increasing x value. Each visit to a point will be referred to as an *event*. Before, after, and in between every event, a list, L, of some $\mathcal{C}_{obs}$ edges will be maintained. This list must be maintained at all times in the order that the edges appear when stabbed by the vertical sweep line. The ordering is maintained from lower to higher.

Algorithm execution Figures 6.6 and 6.7 show how the algorithm proceeds. Initially, L is empty, and a doubly connected edge list is used to represent $\mathcal{C}_{free}$. Each connected component of $\mathcal{C}_{free}$ yields a single face in the data structure. Suppose inductively that after several events occur, L is correctly maintained. For each event, one of the four cases in Figure 6.2 occurs. By maintaining L in a balanced binary search tree [243], the edges above and below p can be determined in $O(\lg n)$ time. This is much better than $O(n)$ time, which would arise from checking every segment. Depending on which of the four cases from Figure 6.2 occurs, different updates to L are made. If the first case occurs, then two different edges are inserted, and the face of which p is on the border is split two times by vertical line segments. For each of the two vertical line segments, two half-edges are added, and all faces and half-edges must be updated correctly (this operation is local in that only records adjacent to where the change occurs need to be updated). The next two cases in Figure 6.2 are simpler; only a single face split is made. For the final case, no splitting occurs.

Once the face splitting operations have been performed, L needs to be updated. When the sweep line crosses p, two edges are always affected. For example, in the first and last cases of Figure 6.2, two edges must be inserted into L (the mirror images of these cases cause two edges to be deleted from L). If the middle two cases occur, then one edge is replaced by another in L. These insertion and deletion operations can be performed in $O(\lg n)$ time. Since there are n events, the running time for the construction algorithm is $O(n \lg n)$.

The roadmap $\mathcal{G}$ can be computed from the face pointers of the doubly connected edge list. A more elegant approach is to incrementally build $\mathcal{G}$ at each event. In fact, all of the pointer maintenance required to obtain a consistent doubly connected edge list can be ignored if desired, as long as $\mathcal{G}$ is correctly built and the sample point is obtained for each cell along the way. We can even go one step further, by forgetting about the cell decomposition and directly building a topological graph of line-segment paths between all sample points of adjacent cells.

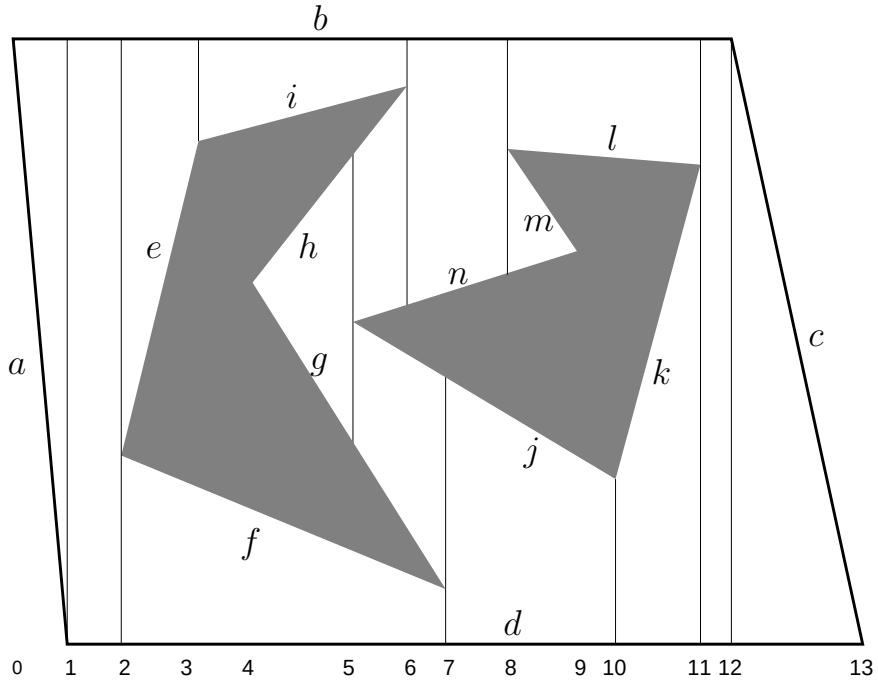


Figure 6.6: There are 14 events in this example.

Event	**Sorted Edges in L**	**Event**	**Sorted Edges in L**
0	$\{a, b\}$	7	$\{d, j, n, b\}$
1	$\{d, b\}$	8	$\{d, j, n, m, l, b\}$
2	$\{d, f, e, b\}$	9	$\{d, j, l, b\}$
3	$\{d, f, i, b\}$	10	$\{d, k, l, b\}$
4	$\{d, f, g, h, i, b\}$	11	$\{d, b\}$
5	$\{d, f, g, j, n, h, i, b\}$	12	$\{d, c\}$
6	$\{d, f, g, j, n, b\}$	13	$\{\}$

Figure 6.7: The status of L is shown after each of 14 events occurs. Before the first event, L is empty.

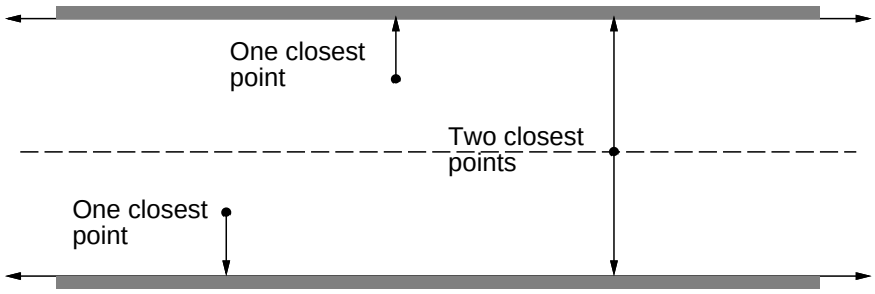


Figure 6.8: The maximum clearance roadmap keeps as far away from the $\mathcal{C}_{obs}$ as possible. This involves traveling along points that are equidistant from two or more points on the boundary of $\mathcal{C}_{obs}$.

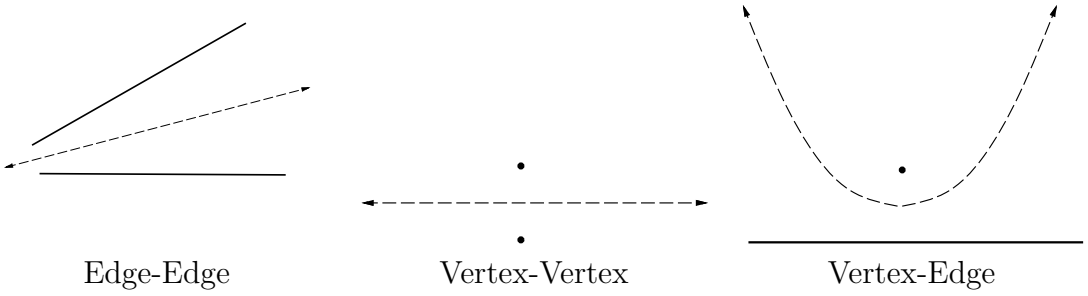


Figure 6.9: Voronoi roadmap pieces are generated in one of three possible cases. The third case leads to a quadratic curve.

6.2.3 Maximum-Clearance Roadmaps

A *maximum-clearance roadmap* tries to keep as far as possible from $\mathcal{C}_{obs}$, as shown for the corridor in Figure 6.8. The resulting solution paths are sometimes preferred in mobile robotics applications because it is difficult to measure and control the precise position of a mobile robot. Traveling along the maximum-clearance roadmap reduces the chances of collisions due to these uncertainties. Other names for this roadmap are *generalized Voronoi diagram* and *retraction method* [749]. It is considered as a generalization of the Voronoi diagram (recall from Section 5.2.2) from the case of points to the case of polygons. Each point along a roadmap edge is equidistant from two points on the boundary of $\mathcal{C}_{obs}$. Each roadmap vertex corresponds to the intersection of two or more roadmap edges and is therefore equidistant from three or more points along the boundary of $\mathcal{C}_{obs}$.

The retraction term comes from topology and provides a nice intuition about the method. A subspace S is a *deformation retract* of a topological space X if the following continuous homotopy, $h : X \times [0, 1] \rightarrow X$, can be defined as follows [451]:

1. $h(x, 0) = x$ for all $x \in X$.

2. $h(x, 1)$ is a continuous function that maps every element of X to some element of S.

3. For all $t \in [0,1]$, $h(s,t) = s$ for any $s \in S$.

The intuition is that $\mathcal{C}_{free}$ is gradually thinned through the homotopy process, until a skeleton, S, is obtained. An approximation to this shrinking process can be imagined by shaving off a thin layer around the whole boundary of $\mathcal{C}_{free}$. If this is repeated iteratively, the maximum-clearance roadmap is the only part that remains (assuming that the shaving always stops when thin "slivers" are obtained).

To construct the maximum-clearance roadmap, the concept of *features* from Section 5.3.3 is used again. Let the *feature set* refer to the set of all edges and vertices of $\mathcal{C}_{obs}$. Candidate paths for the roadmap are produced by every pair of features. This leads to a naive $O(n^4)$ time algorithm as follows. For every edge-edge feature pair, generate a line as shown in Figure 6.9a. For every vertex-vertex pair, generate a line as shown in Figure 6.9b. The maximum-clearance path between a point and a line is a parabola. Thus, for every edge-point pair, generate a parabolic curve as shown in Figure 6.9c. The portions of the paths that actually lie on the maximum-clearance roadmap are determined by intersecting the curves. Several algorithms exist that provide better asymptotic running times [616, 626], but they are considerably more difficult to implement. The best-known algorithm runs in $O(n \lg n)$ time in which n is the number of roadmap curves [865].

6.2.4 Shortest-Path Roadmaps

Instead of generating paths that maximize clearance, suppose that the goal is to find shortest paths. This leads to the *shortest-path roadmap*, which is also called the *reduced visibility graph* in [588]. The idea was first introduced in [742] and may perhaps be the first example of a motion planning algorithm. The shortest-path roadmap is in direct conflict with maximum clearance because shortest paths tend to graze the corners of $\mathcal{C}_{obs}$. In fact, the problem is ill posed because $\mathcal{C}_{free}$ is an open set. For any path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$, it is always possible to find a shorter one. For this reason, we must consider the problem of determining shortest paths in $\mathrm{cl}(\mathcal{C}_{free})$, the closure of $\mathcal{C}_{free}$. This means that the robot is allowed to "touch" or "graze" the obstacles, but it is not allowed to penetrate them. To actually use the computed paths as solutions to a motion planning problem, they need to be slightly adjusted so that they come very close to $\mathcal{C}_{obs}$ but do not make contact. This slightly increases the path length, but the additional cost can be made arbitrarily small as the path gets arbitrarily close to $\mathcal{C}_{obs}$.

The *shortest-path roadmap*, $\mathcal{G}$, is constructed as follows. Let a *reflex vertex* be a polygon vertex for which the interior angle (in $\mathcal{C}_{free}$) is greater than π. All vertices of a convex polygon (assuming that no three consecutive vertices are collinear) are reflex vertices. The vertices of $\mathcal{G}$ are the reflex vertices. Edges of $\mathcal{G}$ are formed from two different sources:

Consecutive reflex vertices: If two reflex vertices are the endpoints of an edge of $\mathcal{C}_{obs}$, then an edge between them is made in $\mathcal{G}$.

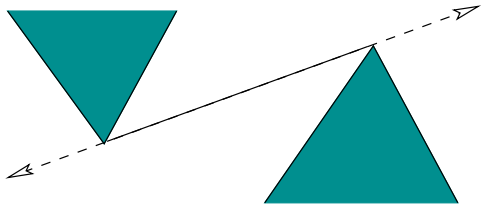


Figure 6.10: A bitangent edge must touch two reflex vertices that are mutually visible from each other, and the line must extend outward past each of them without poking into $\mathcal{C}_{obs}$.

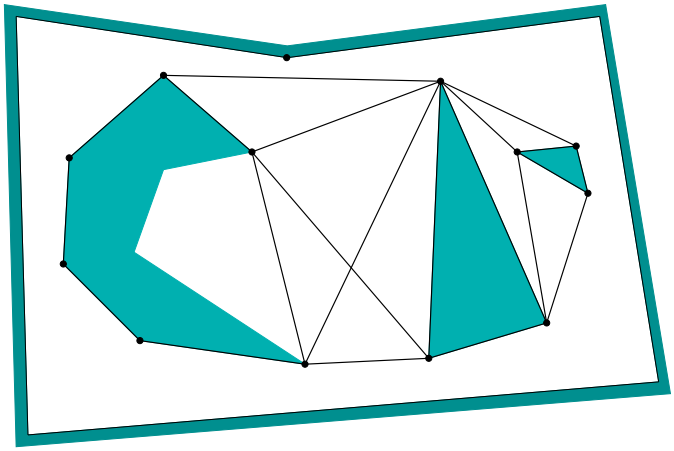


Figure 6.11: The shortest-path roadmap includes edges between consecutive reflex vertices on $\mathcal{C}_{obs}$ and also bitangent edges.

> **Bitangent edges:** If a *bitangent line* can be drawn through a pair of reflex vertices, then a corresponding edge is made in $\mathcal{G}$. A bitangent line, depicted in Figure 6.10, is a line that is incident to two reflex vertices and does not poke into the interior of $\mathcal{C}_{obs}$ at any of these vertices. Furthermore, these vertices must be mutually visible from each other.

An example of the resulting roadmap is shown in Figure 6.11. Note that the roadmap may have isolated vertices, such as the one at the top of the figure. To solve a query, q_I and q_G are connected to all roadmap vertices that are visible; this is shown in Figure 6.12. This makes an extended roadmap that is searched for a solution. If Dijkstra's algorithm is used, and if each edge is given a cost that corresponds to its path length, then the resulting solution path is the shortest path between q_I and q_G. The shortest path for the example in Figure 6.12 is shown in Figure 6.13.

If the bitangent tests are performed naively, then the resulting algorithm requires $O(n^3)$ time, in which n is the number of vertices of $\mathcal{C}_{obs}$. There are $O(n^2)$ pairs of reflex vertices that need to be checked, and each check requires $O(n)$ time to make certain that no other edges prevent their mutual visibility. The plane-sweep principle from Section 6.2.2 can be adapted to obtain a better algorithm,

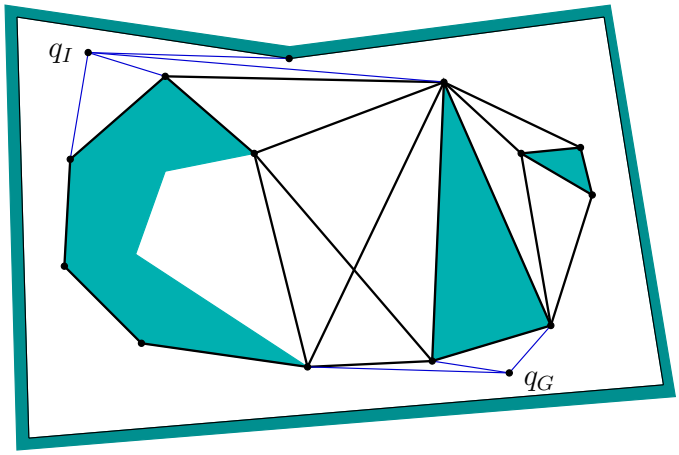


Figure 6.12: To solve a query, q_I and q_G are connected to all visible roadmap vertices, and graph search is performed.

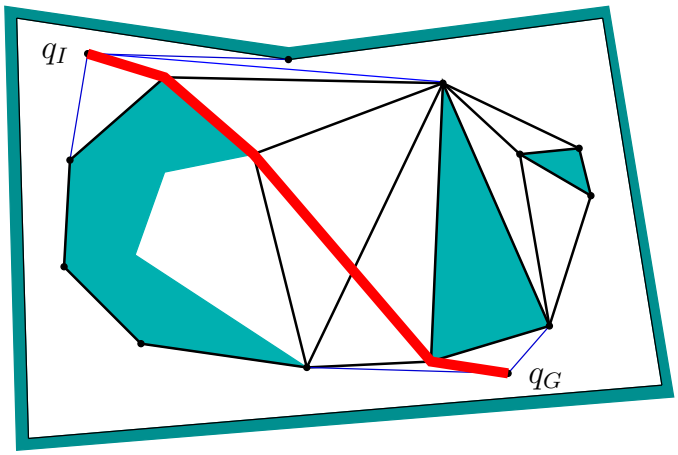


Figure 6.13: The shortest path in the extended roadmap is the shortest path between q_I and q_G.

which takes only $O(n^2 \lg n)$ time. The idea is to perform a *radial sweep* from each reflex vertex, v. A ray is started at $\theta = 0$, and events occur when the ray touches vertices. A set of bitangents through v can be computed in this way in $O(n \lg n)$ time. Since there are $O(n)$ reflex vertices, the total running time is $O(n^2 \lg n)$. See Chapter 15 of [264] for more details. There exists an algorithm that can compute the shortest-path roadmap in time $O(n \lg n + m)$, in which m is the total number of edges in the roadmap [384]. If the obstacle region is described by a simple polygon, the time complexity can be reduced to $O(n)$; see [709] for many shortest-path variations and references.

To improve numerical robustness, the shortest-path roadmap can be implemented without the use of trigonometric functions. For a sequence of three points, p_1, p_2, p_3, define the *left-turn predicate*, $f_l : \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \{\text{TRUE}, \text{FALSE}\}$, as

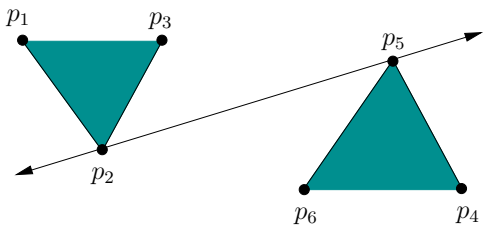


Figure 6.14: Potential bitangents can be identified by checking for left turns, which avoids the use of trigonometric functions and their associated numerical problems.

$f_l(p_1, p_2, p_3) =$ TRUE if and only if p_3 is to the left of the ray that starts at p_1 and pierces p_2. A point p_2 is a reflex vertex if and only if $f_l(p_1, p_2, p_3) =$ TRUE, in which p_1 and p_3 are the points before and after, respectively, along the boundary of $\mathcal{C}_{obs}$. The bitangent test can be performed by assigning points as shown in Figure 6.14. Assume that no three points are collinear and the segment that connects p_2 and p_5 is not in collision. The pair, p_2, p_5, of vertices should receive a bitangent edge if the following sentence is FALSE:

$$\big(f_l(p_1, p_2, p_5) \oplus f_l(p_3, p_2, p_5)\big) \vee \big(f_l(p_4, p_5, p_2) \oplus f_l(p_6, p_5, p_2)\big), \tag{6.1}$$

in which $\oplus$ denotes logical "exclusive or." The f_l predicate can be implemented without trigonometric functions by defining

$$M(p_1, p_2, p_3) = \begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{pmatrix}, \tag{6.2}$$

in which $p_i = (x_i, y_i)$. If $\det(M) > 0$, then $f_l(p_1, p_2, p_3) =$ TRUE; otherwise, $f_l(p_1, p_2, p_3) =$ FALSE.

6.3 Cell Decompositions

Section 6.2.2 introduced the vertical cell decomposition to solve the motion planning problem when $\mathcal{C}_{obs}$ is polygonal. It is important to understand, however, that this is just one choice among many for the decomposition. Some of these choices may not be preferable in 2D; however, they might generalize better to higher dimensions. Therefore, other cell decompositions are covered in this section, to provide a smoother transition from vertical cell decomposition to cylindrical algebraic decomposition in Section 6.4, which solves the motion planning problem in any dimension for any semi-algebraic model. Along the way, a cylindrical decomposition will appear in Section 6.3.4 for the special case of a line-segment robot in $\mathcal{W} = \mathbb{R}^2$.

6.3.1 General Definitions

In this section, the term *complex* refers to a collection of cells together with their boundaries. A partition into cells can be derived from a complex, but the complex contains additional information that describes how the cells must fit together. The term *cell decomposition* still refers to the partition of the space into cells, which is derived from a *complex*.

It is tempting to define complexes and cell decompositions in a very general manner. Imagine that any partition of $\mathcal{C}_{free}$ could be called a cell decomposition. A cell could be so complicated that the notion would be useless. Even $\mathcal{C}_{free}$ itself could be declared as one big cell. It is more useful to build decompositions out of simpler cells, such as ones that contain no holes. Formally, this requires that every k-dimensional cell is homeomorphic to $B^k \subset \mathbb{R}^k$, an open k-dimensional unit ball. From a motion planning perspective, this still yields cells that are quite complicated, and it will be up to the particular cell decomposition method to enforce further constraints to yield a complete planning algorithm.

Two different complexes will be introduced. The *simplicial complex* is explained because it is one of the easiest to understand. Although it is useful in many applications, it is not powerful enough to represent all of the complexes that arise in motion planning. Therefore, the *singular complex* is also introduced. Although it is more complicated to define, it encompasses all of the cell complexes that are of interest in this book. It also provides an elegant way to represent topological spaces. Another important cell complex, which is not covered here, is the *CW-complex* [439].

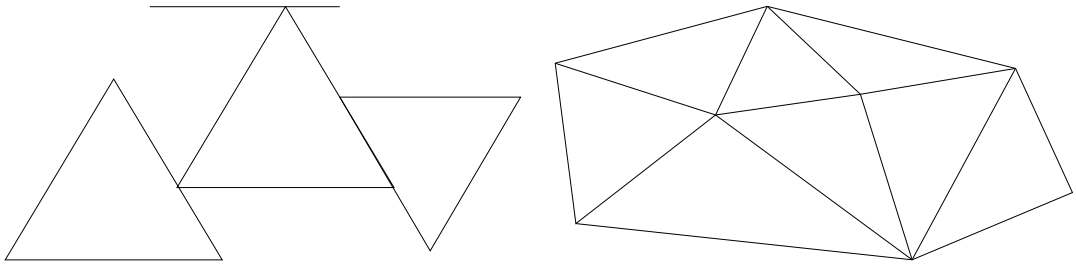
Simplicial Complex For this definition, it is assumed that $X = \mathbb{R}^n$. Let p_1, p_2, …, p_{k+1}, be $k+1$ linearly independent[5] points in $\mathbb{R}^n$. A k-simplex, $[p_1, \ldots, p_{k+1}]$, is formed from these points as

$$[p_1, \ldots, p_{k+1}] = \left\{ \sum_{i=1}^{k+1} \alpha_i p_i \in \mathbb{R}^n \;\middle|\; \alpha_i \geq 0 \text{ for all } i \text{ and } \sum_{i=1}^{k+1} \alpha_i = 1 \right\}, \tag{6.3}$$

in which $\alpha_i p_i$ is the scalar multiplication of α_i by each of the point coordinates. Another way to view (6.3) is as the convex hull of the $k+1$ points (i.e., all ways to linearly interpolate between them). If $k = 2$, a triangular region is obtained. For $k = 3$, a tetrahedron is produced.

For any k-simplex and any i such that $1 \leq i \leq k+1$, let $\alpha_i = 0$. This yields a $(k-1)$-dimensional simplex that is called a *face* of the original simplex. A 2-simplex has three faces, each of which is a 1-simplex that may be called an edge. Each 1-simplex (or edge) has two faces, which are 0-simplexes called *vertices*.

[5]Form k vectors by subtracting p_1 from the other k points for some positive integer k such that $k \leq n$. Arrange the vectors into a $k \times n$ matrix. For linear independence, there must be at least one $k \times k$ cofactor with a nonzero determinant. For example, if $k = 2$, then the three points cannot be collinear.



Not a simplicial complex A simplicial complex

Figure 6.15: To become a simplicial complex, the simplex faces must fit together nicely.

To form a complex, the simplexes must fit together in a nice way. This yields a high-dimensional notion of a *triangulation*, which in $\mathbb{R}^2$ is a tiling composed of triangular regions. A *simplicial complex*, $\mathcal{K}$, is a finite set of simplexes that satisfies the following:

1. Any face of a simplex in $\mathcal{K}$ is also in $\mathcal{K}$.

2. The intersection of any two simplexes in $\mathcal{K}$ is either a common face of both of them or the intersection is empty.

Figure 6.15 illustrates these requirements. For $k > 0$, a k-cell of $\mathcal{K}$ is defined to be interior, $\text{int}([p_1, \ldots, p_{k+1}])$, of any k-simplex. For $k = 0$, every 0-simplex is a 0-cell. The union of all of the cells forms a partition of the point set covered by $\mathcal{K}$. This therefore provides a *cell decomposition* in a sense that is consistent with Section 6.2.2.

Singular complex Simplicial complexes are useful in applications such as geometric modeling and computer graphics for computing the topology of models. Due to the complicated topological spaces, implicit, nonlinear models, and decomposition algorithms that arise in motion planning, they are insufficient for the most general problems. A *singular complex* is a generalization of the *simplicial complex*. Instead of being limited to $\mathbb{R}^n$, a singular complex can be defined on any manifold, X (it can even be defined on any Hausdorff topological space). The main difference is that, for a simplicial complex, each simplex is a subset of $\mathbb{R}^n$; however, for a singular complex, each *singular simplex* is actually a homeomorphism from a (simplicial) simplex in $\mathbb{R}^n$ to a subset of X.

To help understand the idea, first consider a 1D singular complex, which happens to be a topological graph (as introduced in Example 4.6). The interval $[0, 1]$ is a 1-simplex, and a continuous path $\tau : [0, 1] \rightarrow X$ is a *singular* 1*-simplex* because it is a homeomorphism of $[0, 1]$ to the image of τ in X. Suppose $\mathcal{G}(V, E)$ is a topological graph. The cells are subsets of X that are defined as follows. Each point $v \in V$ is a 0-cell in X. To follow the formalism, each is considered as the

image of a function $f : \{0\} \rightarrow X$, which makes it a *singular* 0*-simplex*, because $\{0\}$ is a 0-simplex. For each path $\tau \in E$, the corresponding 1-cell is

$$\{x \in X \mid \tau(s) = x \text{ for some } s \in (0,1)\}. \tag{6.4}$$

Expressed differently, it is $\tau((0,1))$, the image of the path τ, except that the endpoints are removed because they are already covered by the 0-cells (the cells must form a partition).

These principles will now be generalized to higher dimensions. Since all balls and simplexes of the same dimension are homeomorphic, balls can be used instead of a simplex in the definition of a singular simplex. Let $B^k \subset \mathbb{R}^k$ denote a closed, k-dimensional unit ball,

$$D^k = \{x \in \mathbb{R}^n \mid \|x\| \leq 1\}, \tag{6.5}$$

in which $\|\cdot\|$ is the Euclidean norm. A *singular* k*-simplex* is a continuous mapping $\sigma : D^k \rightarrow X$. Let $\text{int}(D^k)$ refer to the interior of D^k. For $k \geq 1$, the k*-cell*, C, corresponding to a singular k-simplex, σ, is the image $C = \sigma(\text{int}(D^k)) \subseteq X$. The 0-cells are obtained directly as the images of the 0 singular simplexes. Each singular 0-simplex maps to the 0-cell in X. If σ is restricted to $\text{int}(D^k)$, then it actually defines a homeomorphism between D^k and C. Note that both of these are open sets if $k > 0$.

A simplicial complex requires that the simplexes fit together nicely. The same concept is applied here, but topological concepts are used instead because they are more general. Let $\mathcal{K}$ be a set of singular simplexes of varying dimensions. Let S_k denote the union of the images of all singular i-simplexes for all $i \leq k$.

A collection of singular simplexes that map into a topological space X is called a *singular complex* if:

1. For each dimension k, the set $S_k \subseteq X$ must be closed. This means that the cells must all fit together nicely.

2. Each k-cell is an open set in the topological subspace S_k. Note that 0-cells are open in S_0, even though they are usually closed in X.

Example 6.1 (Vertical Decomposition) The vertical decomposition of Section 6.2.2 is a nice example of a singular complex that is not a simplicial complex because it contains trapezoids. The interior of each trapezoid and triangle forms a 2-cell, which is an open set. For every pair of adjacent 2-cells, there is a 1-cell on their common boundary. There are no 0-cells because the vertices lie in $\mathcal{C}_{obs}$, not in $\mathcal{C}_{free}$. The subspace S_2 is formed by taking the union of all 2-cells and 1-cells to yield $S_2 = \mathcal{C}_{free}$. This satisfies the closure requirement because the complex is built in $\mathcal{C}_{free}$ only; hence, the topological space is $\mathcal{C}_{free}$. The set $S_2 = \mathcal{C}_{free}$ is both open and closed. The set S_1 is the union of all 1-cells. This is also closed because the 1-cell endpoints all lie in $\mathcal{C}_{obs}$. Each 1-cell is also an open set.

One way to avoid some of these strange conclusions from the topology restricted to $\mathcal{C}_{free}$ is to build the vertical decomposition in $\text{cl}(\mathcal{C}_{free})$, the closure of $\mathcal{C}_{free}$. This

can be obtained by starting with the previously defined vertical decomposition and adding a new 1-cell for every edge of $\mathcal{C}_{obs}$ and a 0-cell for every vertex of $\mathcal{C}_{obs}$. Now $S_3 = \text{cl}(\mathcal{C}_{free})$, which is closed in $\mathbb{R}^2$. Likewise, S_2, S_1, and S_0, are closed in the usual way. Each of the individual k-dimensional cells, however, is open in the topological space S_k. The only strange case is that the 0-cells are considered open, but this is true in the discrete topological space S_0. ■

6.3.2 2D Decompositions

The vertical decomposition method of Section 6.2.2 is just one choice of many cell decomposition methods for solving the problem when $\mathcal{C}_{obs}$ is polygonal. It provides a nice balance between the number of cells, computational efficiency, and implementation ease. It is usually possible to decompose $\mathcal{C}_{obs}$ into far fewer convex cells. This would be preferable for multiple-query applications because the roadmap would be smaller. It is unfortunately quite difficult to optimize the number of cells. Determining the decomposition of a polygonal $\mathcal{C}_{obs}$ with holes that uses the smallest number of convex cells is NP-hard [519, 645]. Therefore, we are willing to tolerate nonoptimal decompositions.

Triangulation One alternative to the vertical decomposition is to perform a *triangulation*, which yields a simplicial complex over $\mathcal{C}_{free}$. Figure 6.16 shows an example. Since $\mathcal{C}_{free}$ is an open set, there are no 0-cells. Each 2-simplex (triangle) has either one, two, or three faces, depending on how much of its boundary is shared with $\mathcal{C}_{obs}$. A roadmap can be made by connecting the samples for 1-cells and 2-cells as shown in Figure 6.17. Note that there are many ways to triangulate $\mathcal{C}_{free}$ for a given problem. Finding good triangulations, which for example means trying to avoid thin triangles, is given considerable attention in computational geometry [129, 264, 302].

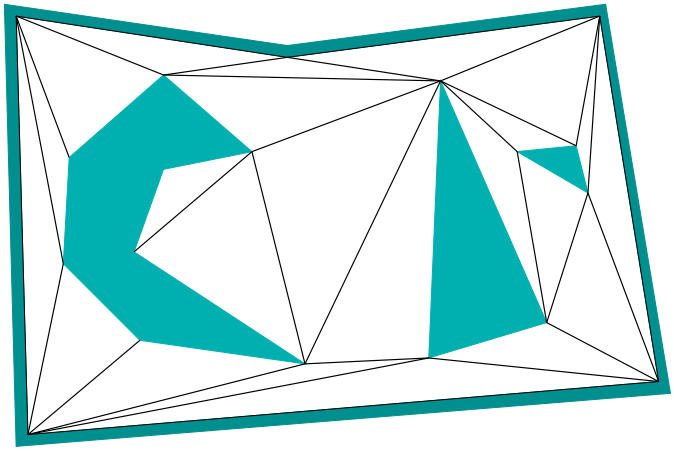


Figure 6.16: A triangulation of $\mathcal{C}_{free}$.

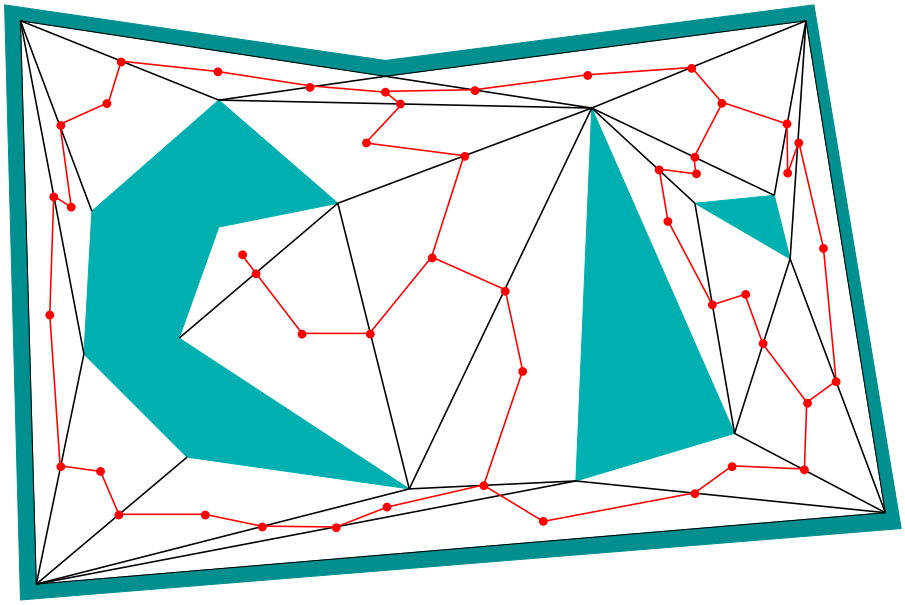


Figure 6.17: A roadmap obtained from the triangulation.

How can the triangulation be computed? It might seem tempting to run the vertical decomposition algorithm of Section 6.2.2 and split each trapezoid into two triangles. Even though this leads to triangular cells, it does not produce a simplicial complex (two triangles could abut the same side of a triangle edge). A naive approach is to incrementally split faces by attempting to connect two vertices of a face by a line segment. If this segment does not intersect other segments, then the split can be made. This process can be iteratively performed over all vertices of faces that have more than three vertices, until a triangulation is eventually obtained. Unfortunately, this results in an $O(n^3)$ time algorithm because $O(n^2)$ pairs must be checked in the worst case, and each check requires $O(n)$ time to determine whether an intersection occurs with other segments. This can be easily reduced to $O(n^2 \lg n)$ by performing radial sweeping. Chapter 3 of [264] presents an algorithm that runs in $O(n \lg n)$ time by first partitioning $\mathcal{C}_{free}$ into *monotone polygons*, and then efficiently triangulating each monotone polygon. If $\mathcal{C}_{free}$ is simply connected, then, surprisingly, a triangulation can be computed in linear time [190]. Unfortunately, this algorithm is too complicated to use in practice (there are, however, simpler algorithms for which the complexity is close to $O(n)$; see [90] and the end of Chapter 3 of [264] for surveys).

Cylindrical decomposition The *cylindrical decomposition* is very similar to the vertical decomposition, except that when any of the cases in Figure 6.2 occurs, then a vertical line slices through all faces, all the way from $y = -\infty$ to $y = \infty$. The result is shown in Figure 6.18, which may be considered as a singular complex. This may appear very inefficient in comparison to the vertical decomposition; however, it is presented here because it generalizes nicely to any dimension, any C-space topology, and any semi-algebraic model. Therefore, it is presented here to ease

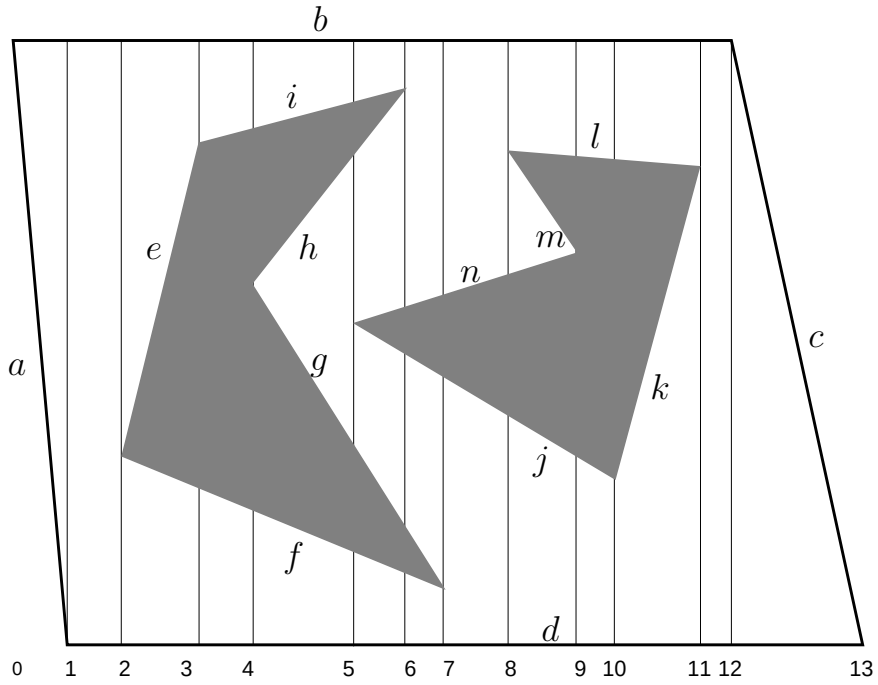


Figure 6.18: The cylindrical decomposition differs from the vertical decomposition in that the rays continue forever instead of stopping at the nearest edge. Compare this figure to Figure 6.6.

the transition to more general decompositions. The most important property of the cylindrical decomposition is shown in Figure 6.19. Consider each vertical strip between two events. When traversing a strip from $y = -\infty$ to $y = \infty$, the points alternate between being $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$. For example, between events 4 and 5, the points below edge f are in $\mathcal{C}_{free}$. Points between f and g lie in $\mathcal{C}_{obs}$. Points between g and h lie in $\mathcal{C}_{free}$, and so forth. The cell decomposition can be defined so that 2D cells are also created in $\mathcal{C}_{obs}$. Let $S(x, y)$ denote the logical predicate (3.6) from Section 3.1.1. When traversing a strip, the value of $S(x, y)$ also alternates. This behavior is the main reason to construct a cylindrical decomposition, which will become clearer in Section 6.4.2. Each vertical strip is actually considered to be a *cylinder*, hence, the name cylindrical decomposition (i.e., there are not necessarily any cylinders in the 3D geometric sense).

6.3.3 3D Vertical Decomposition

It turns out that the vertical decomposition method of Section 6.2.2 can be extended to any dimension by recursively applying the sweeping idea. The method requires, however, that $\mathcal{C}_{obs}$ is piecewise linear. In other words, $\mathcal{C}_{obs}$ is represented as a semi-algebraic model for which all primitives are linear. Unfortunately, most of the general motion planning problems involve nonlinear algebraic primitives because of the nonlinear transformations that arise from rotations. Recall the

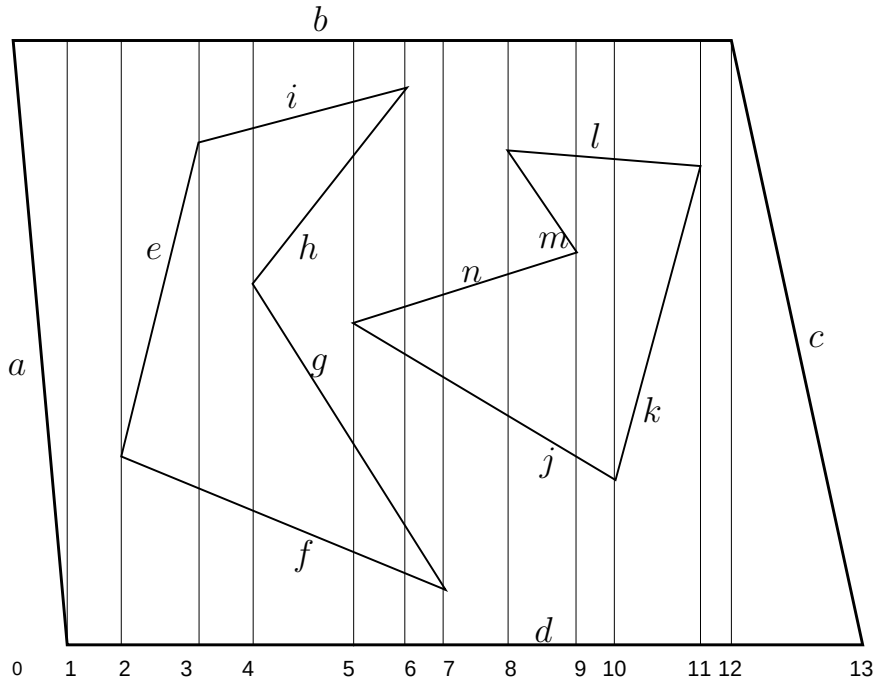


Figure 6.19: The cylindrical decomposition produces vertical strips. Inside of a strip, there is a stack of collision-free cells, separated by $\mathcal{C}_{obs}$.

complicated algebraic $\mathcal{C}_{obs}$ model constructed in Section 4.3.3. To handle generic algebraic models, powerful techniques from computational algebraic geometry are needed. This will be covered in Section 6.4.

One problem for which $\mathcal{C}_{obs}$ is piecewise linear is a polyhedral robot that can translate in $\mathbb{R}^3$, and the obstacles in $\mathcal{W}$ are polyhedra. Since the transformation equations are linear in this case, $\mathcal{C}_{obs} \subset \mathbb{R}^3$ is polyhedral. The polygonal faces of $\mathcal{C}_{obs}$ are obtained by forming geometric primitives for each of the Type FV, Type VF, and Type EE cases of contact between $\mathcal{A}$ and $\mathcal{O}$, as mentioned in Section 4.3.2.

Figure 6.20 illustrates the algorithm that constructs the 3D vertical decomposition. Compare this to the algorithm in Section 6.2.2. Let (x, y, z) denote a point in $\mathcal{C} = \mathbb{R}^3$. The vertical decomposition yields convex 3-cells, 2-cells, and 1-cells. Neglecting degeneracies, a generic 3-cell is bounded by six planes. The cross section of a 3-cell for some fixed x value yields a trapezoid or triangle, exactly as in the 2D case, but in a plane parallel to the yz plane. Two sides of a generic 3-cell are parallel to the yz plane, and two other sides are parallel to the xz plane. The 3-cell is bounded above and below by two polygonal faces of $\mathcal{C}_{obs}$.

Initially, sort the $\mathcal{C}_{obs}$ vertices by their x coordinate to obtain the events. Now consider sweeping a plane perpendicular to the x-axis. The plane for a fixed value of x produces a 2D polygonal slice of $\mathcal{C}_{obs}$. Three such slices are shown at the bottom of Figure 6.20. Each slice is parallel to the yz plane and appears to look exactly like a problem that can be solved by the 2D vertical decomposition method.

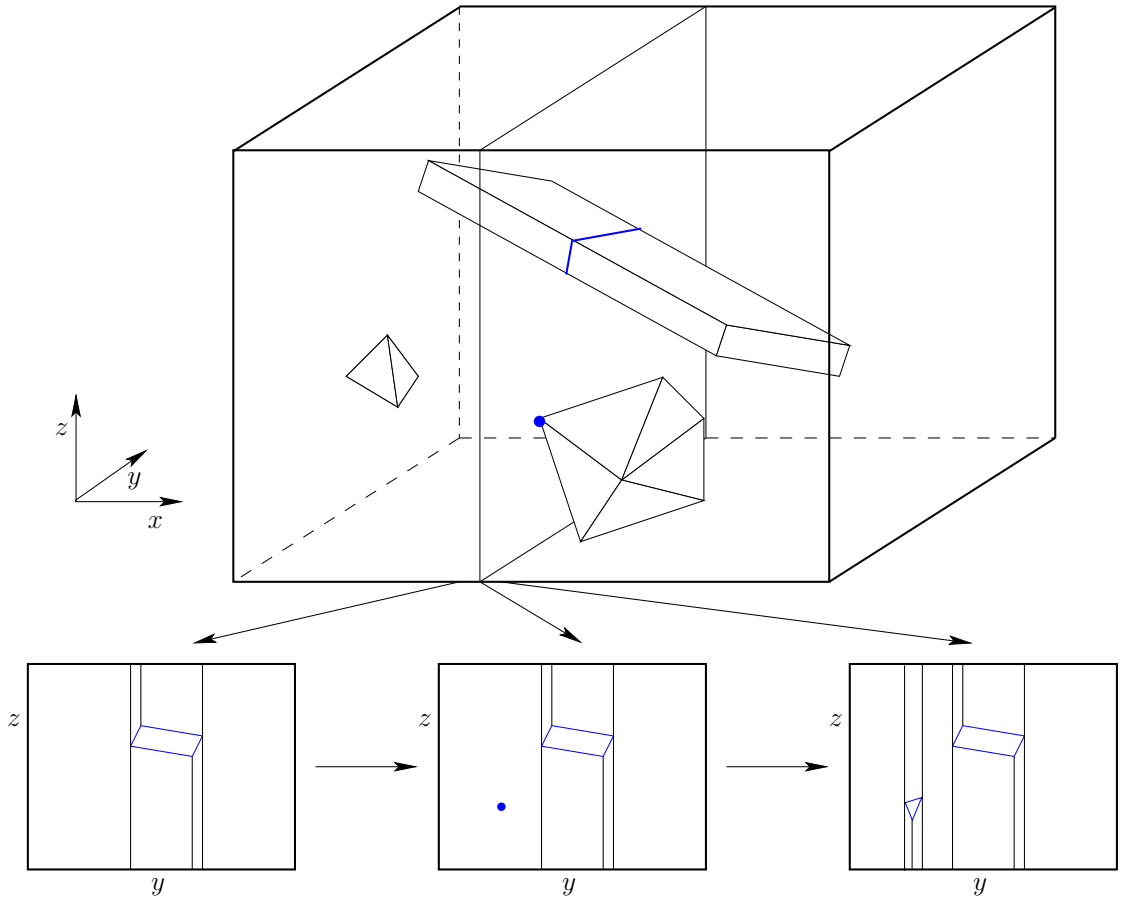


Figure 6.20: In higher dimensions, the sweeping idea can be applied recursively.

The 2-cells in a slice are actually slices of 3-cells in the 3D decomposition. The only places in which these 3-cells can critically change is when the sweeping plane stops at some x value. The center slice in Figure 6.20 corresponds to the case in which a vertex of a convex polyhedron is encountered, and all of the polyhedron lies to right of the sweep plane (i.e., the rest of the polyhedron has not been encountered yet). This corresponds to a place where a critical change must occur in the slices. These are 3D versions of the cases in Figure 6.2, which indicate how the vertical decomposition needs to be updated. The algorithm proceeds by first building the 2D vertical decomposition at the first x event. At each event, the 2D vertical decomposition must be updated to take into account the critical changes. During this process, the 3D cell decomposition and roadmap can be incrementally constructed, as in the 2D case.

The roadmap is constructed by placing a sample point in the center of each 3-cell and 2-cell. The vertices are the sample points, and edges are added to the roadmap by connecting the sample points for each case in which a 3-cell is adjacent to a 2-cell.

This same principle can be extended to any dimension, but the applications to

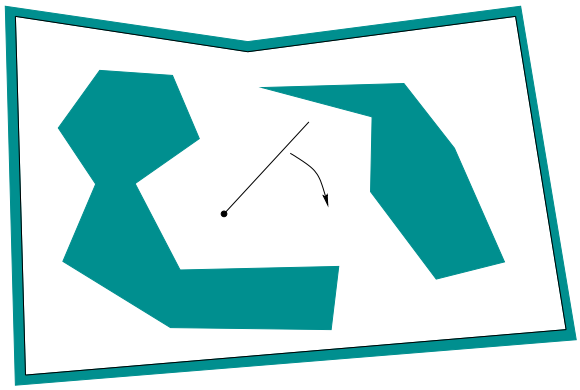


Figure 6.21: Motion planning for a line segment that can translate and rotate in a 2D world.

motion planning are limited because the method requires linear models (or at least it is very challenging to adapt to nonlinear models; in some special cases, this can be done). See [426] for a summary of the complexity of vertical decompositions for various geometric primitives and dimensions.

6.3.4 A Decomposition for a Line-Segment Robot

This section presents one of the simplest cell decompositions that involves nonlinear models, yet it is already fairly complicated. This will help to give an appreciation of the difficulty of combinatorial planning in general. Consider the planning problem shown in Figure 6.21. The robot, $\mathcal{A}$, is a single line segment that can translate or rotate in $\mathcal{W} = \mathbb{R}^2$. The dot on one end of $\mathcal{A}$ is used to illustrate its origin and is not part of the model. The C-space, $\mathcal{C}$, is homeomorphic to $\mathbb{R}^2 \times \mathbb{S}^1$. Assume that the parameterization $\mathbb{R}^2 \times [0, 2\pi]/\sim$ is used in which the identification equates $\theta = 0$ and $\theta = 2\pi$. A point in $\mathcal{C}$ is represented as (x, y, θ).

An approximate solution First consider making a cell decomposition for the case in which the segment can only translate. The method from Section 4.3.2 can be used to compute $\mathcal{C}_{obs}$ by treating the robot-obstacle interaction with Type EV and Type VE contacts. When the interior of $\mathcal{A}$ touches an obstacle vertex, then Type EV is obtained. An endpoint of $\mathcal{A}$ touching an object interior yields Type VE. Each case produces an edge of $\mathcal{C}_{obs}$, which is polygonal. Once this is represented, the vertical decomposition can be used to solve the problem. This inspires a reasonable numerical approach to the rotational case, which is to discretize θ into K values, $i\Delta\theta$, for $0 \leq i \leq K$, and $\Delta\theta = 2\pi/K$ [20]. The obstacle region, $\mathcal{C}_{obs}$, is polygonal for each case, and we can imagine having a stack of K polygonal regions. A roadmap can be formed by connecting sampling points inside of a slice in the usual way, and also by connecting samples between corresponding cells in neighboring slices. If K is large enough, this strategy works well, but the method is not complete because a sufficient value for K cannot be determined in

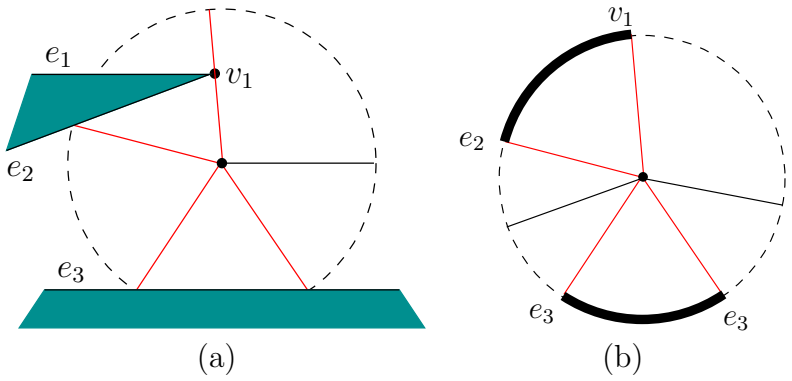


Figure 6.22: Fix (x, y) and swing the segment around for all values of $\theta \in [0, 2\pi]/\sim$. (a) Note the vertex and edge features that are hit by the segment. (b) Record orientation intervals over which the robot is not in collision.

advance. The method is actually an interesting hybrid between combinatorial and sampling-based motion planning. A resolution-complete version can be imagined.

In the limiting case, as K tends to infinity, the surfaces of $\mathcal{C}_{obs}$ become curved along the θ direction. The conditions in Section 4.3.3 must be applied to generate the actual obstacle regions. This is possible, but it yields a semi-algebraic representation of $\mathcal{C}_{obs}$ in terms of implicit polynomial primitives. It is no easy task to determine an explicit representation in terms of simple cells that can be used for motion planning. The method of Section 6.3.3 cannot be used because $\mathcal{C}_{obs}$ is not polyhedral. Therefore, special analysis is warranted to produce a cell decomposition.

The general idea is to construct a cell decomposition in $\mathbb{R}^2$ by considering only the translation part, (x, y). Each cell in $\mathbb{R}^2$ is then lifted into $\mathcal{C}$ by considering θ as a third axis that is "above" the xy plane. A cylindrical decomposition results in which each cell in the xy plane produces a cylindrical stack of cells for different θ values. Recall the cylinders in Figures 6.18 and 6.19. The vertical axis corresponds to θ in the current setting, and the horizontal axis is replaced by two axes, x and y.

To construct the decomposition in $\mathbb{R}^2$, consider the various robot-obstacle contacts shown in Figure 6.22. In Figure 6.22a, the segment swings around from a fixed (x, y). Two different kinds of contacts arise. For some orientation (value of θ), the segment contacts v_1, forming a Type EV contact. For three other orientations, the segment contacts an edge, forming Type VE contacts. Once again using the *feature* concept, there are four orientations at which the segment contacts a feature. Each feature may be either a vertex or an edge. Between the two contacts with e_2 and e_3, the robot is not in collision. These configurations lie in $\mathcal{C}_{free}$. Also, configurations for which the robot is between contacts e_3 (the rightmost contact) and v_1 are also in $\mathcal{C}_{free}$. All other orientations produce configurations in $\mathcal{C}_{obs}$. Note that the line segment cannot get from being between e_2 and e_3 to being between

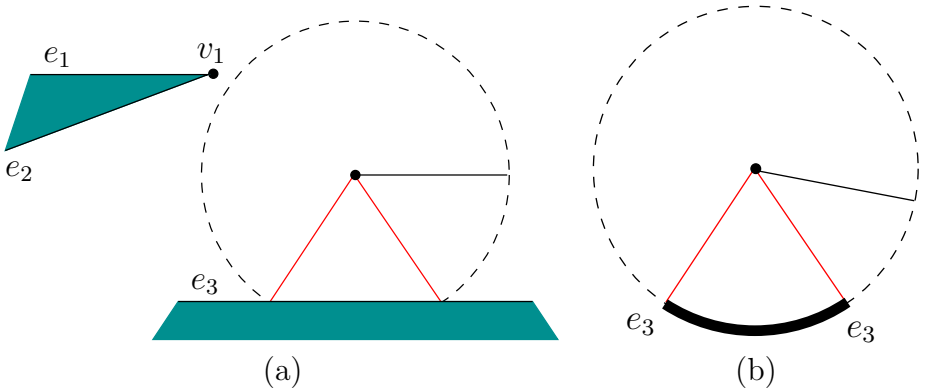


Figure 6.23: If x is increased enough, a critical change occurs in the radar map because v_1 can no longer be reached by the robot.

e_3 and v_1, unless the (x, y) position is changed. It therefore seems sensible that these must correspond to different cells in whatever decomposition is made.

Radar maps Figure 6.22b illustrates which values of θ produce collision. We will refer to this representation as a *radar map*. The four contact orientations are indicated by the contact feature. The notation $[e_3, v_1]$ and $[e_2, e_3]$ identifies the two intervals for which $(x, y, \theta) \in \mathcal{C}_{free}$. Now imagine changing (x, y) by a small amount, to obtain (x', y'). How would the radar map change? The precise angles at which the contacts occur would change, but the notation $[e_3, v_1]$ and $[e_2, e_3]$, for configurations that lie in $\mathcal{C}_{free}$, remains unchanged. Even though the angles change, there is no interesting change in terms of the contacts; therefore, it makes sense to declare (x, y, θ) and (x, y, θ') to lie in the same cell in $\mathcal{C}_{free}$ because θ and θ' both place the segment between the same contacts. Imagine a column of two 3-cells above a small area around (x, y). One 3-cell is for orientations in $[e_3, v_1]$, and the other is for orientations in $[e_2, e_3]$. These appear to be 3D regions in $\mathcal{C}_{free}$ because each of x, y, and θ can be perturbed a small amount without leaving the cell.

Of course, if (x, y) is changed enough, then eventually we expect a dramatic change to occur in the radar map. For example, imagine e_3 is infinitely long, and the x value is gradually increased in Figure 6.22a. The black band between v_1 and e_2 in Figure 6.22b shrinks in length. Eventually, when the distance from (x', y') to v_1 is greater than the length of $\mathcal{A}$, the black band disappears. This situation is shown in Figure 6.23. The change is very important to notice because after that region vanishes, any orientation θ' between e_3 and e_3, traveling the long way around the circle, produces a configuration $(x', y', \theta') \in \mathcal{C}_{free}$. This seems very important because it tells us that we can travel between the original two cells by moving the robot further way from v_1, rotating the robot, and then moving back. Now move from the position shown in Figure 6.23 into the positive y direction. The remaining black band begins to shrink and finally disappears when the distance

to e_3 is further than the robot length. This represents another critical change.

The radar map can be characterized by specifying a circular ordering

$$([f_1, f_2], [f_3, f_4], [f_5, f_6], \ldots, [f_{2k-1}, f_{2k}]), \tag{6.6}$$

when there are k orientation intervals over which the configurations lie in $\mathcal{C}_{free}$. For the radar map in Figure 6.22b, this representation yields $([e_3, v_1], [e_2, e_3])$. Each f_i is a feature, which may be an edge or a vertex. Some of the f_i may be identical; the representation for Figure 6.23b is $([e_3, e_3])$. The intervals are specified in counterclockwise order around the radar map. Since the ordering is circular, it does not matter which interval is specified first. There are two degenerate cases. If $(x, y, \theta) \in \mathcal{C}_{free}$ for all $\theta \in [0, 2\pi)$, then we write () for the ordering. On the other hand, if $(x, y, \theta) \in \mathcal{C}_{obs}$ for all $\theta \in [0, 2\pi)$, then we write $\emptyset$.

Critical changes in cells Now we are prepared to explain the cell decomposition in more detail. Imagine traveling along a path in $\mathbb{R}^2$ and producing an animated version of the radar map in Figure 6.22b. We say that a *critical change* occurs each time the circular ordering representation of (6.6) changes. Changes occur when intervals: 1) appear, 2) disappear, 3) split apart, 4) merge into one, or 5) when the feature of an interval changes. The first task is to partition $\mathbb{R}^2$ into maximal 2-cells over which no critical changes occur. Each one of these 2-cells, R, represents the projection of a strip of 3-cells in $\mathcal{C}_{free}$. Each 3-cell is defined as follows. Let $\{R, [f_i, f_{i+1}]\}$ denote the 3D region in $\mathcal{C}_{free}$ for which $(x, y) \in R$ and θ places the segment between contacts f_i and f_{i+1}. The *cylinder* of cells above R is given by $\{R, [f_i, f_{i+1}]\}$ for each interval in the circular ordering representation, (6.6). If any orientation is possible because $\mathcal{A}$ never contacts an obstacle while in R, then we write $\{R\}$.

What are the positions in $\mathbb{R}^2$ that cause critical changes to occur? It turns out that there are five different cases to consider, each of which produces a set of *critical curves* in $\mathbb{R}^2$. When one of these curves is crossed, a critical change occurs. If none of these curves is crossed, then no critical change can occur. Therefore, these curves precisely define the boundaries of the desired 2-cells in $\mathbb{R}^2$. Let L denote the length of the robot (which is the line segment).

Consider how the five cases mentioned above may occur. Two of the five cases have already been observed in Figures 6.22 and 6.23. These appear in Figures 6.24a and Figures 6.24b, and occur if (x, y) is within L of an edge or a vertex. The third and fourth cases are shown in Figures 6.24c and 6.24d, respectively. The third case occurs because crossing the curve causes $\mathcal{A}$ to change between being able to touch e and being able to touch v. This must be extended from any edge at an endpoint that is a reflex vertex (interior angle is greater than π). The fourth case is actually a return of the bitangent case from Figure 6.10, which arose for the shortest path graph. If the vertices are within L of each other, then a linear critical curve is generated because $\mathcal{A}$ is no longer able to touch v_2 when crossing it from right to left. Bitangents always produce curves in pairs; the curve above v_2 is not shown. The final case, shown in Figure 6.25, is the most complicated. It is a

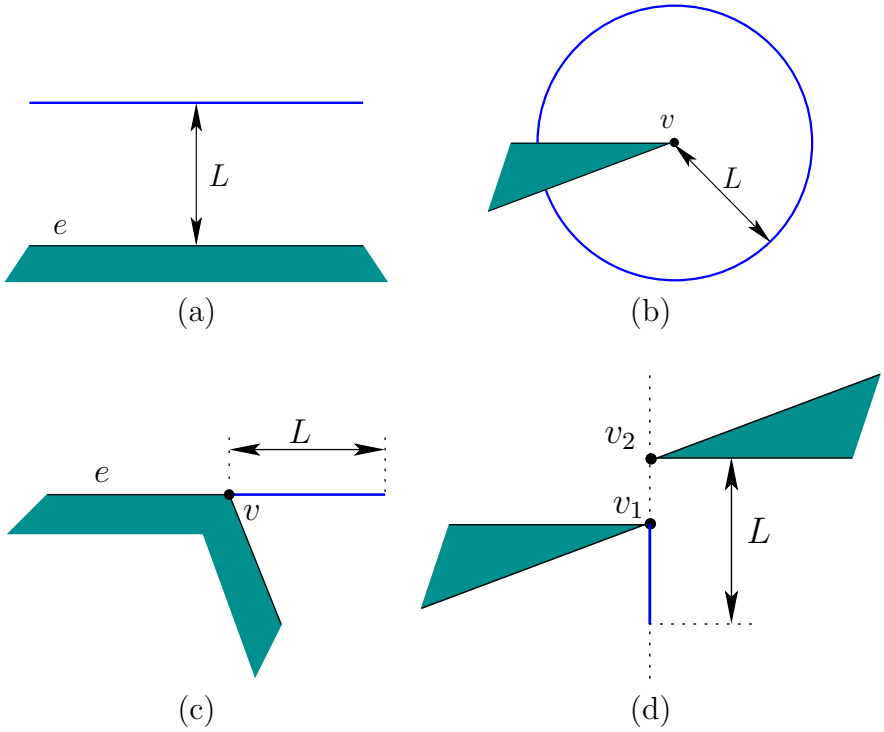


Figure 6.24: Four of the five cases that produce critical curves in $\mathbb{R}^2$.

fourth-degree algebraic curve called the Conchoid of Nicomedes, which arises from $\mathcal{A}$ being in simultaneous contact between v and e. Inside of the teardrop-shaped curve, $\mathcal{A}$ can contact e but not v. Just outside of the curve, it can touch v. If the xy coordinate frame is placed so that v is at $(0, 0)$, then the equation of the curve is

$$(x^2 - y^2)(y + d)^2 - y^2 L^2 = 0, \tag{6.7}$$

in which d is the distance from v to e.

Putting all of the curves together generates a cell decomposition of $\mathbb{R}^2$. There are *noncritical regions*, over which there is no change in (6.6); these form the 2-cells. The boundaries between adjacent 2-cells are sections of the critical curves

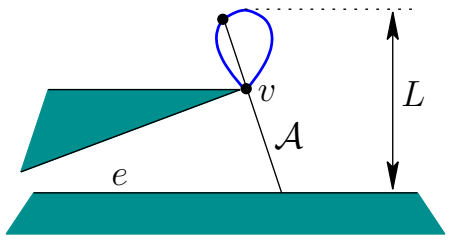


Figure 6.25: The fifth case is the most complicated. It results in a fourth-degree algebraic curve called the Conchoid of Nicomedes.

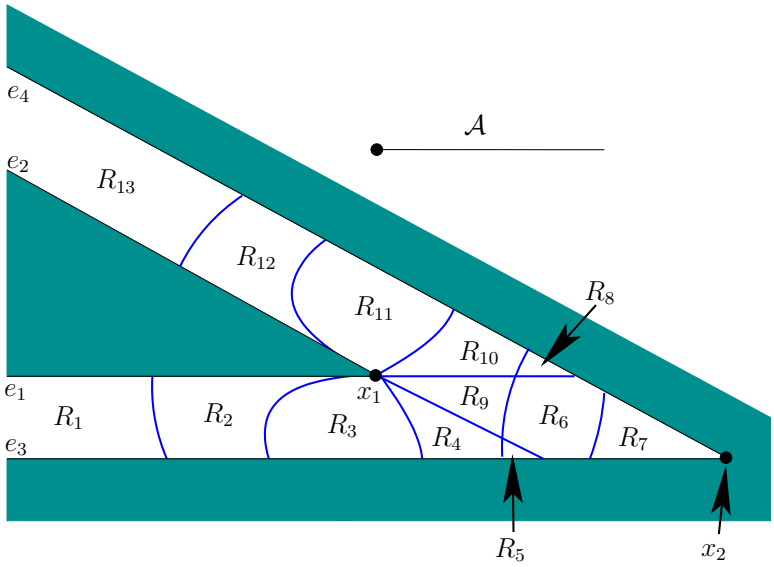


Figure 6.26: The critical curves form the boundaries of the noncritical regions in $\mathbb{R}^2$.

and form 1-cells. There are also 0-cells at places where critical curves intersect. Figure 6.26 shows an example adapted from [588]. Note that critical curves are not drawn if their corresponding configurations are all in $\mathcal{C}_{obs}$. The method still works correctly if they are included, but unnecessary cell boundaries are made. Just for fun, they could be used to form a nice cell decomposition of $\mathcal{C}_{obs}$, in addition to $\mathcal{C}_{free}$. Since $\mathcal{C}_{obs}$ is avoided, is seems best to avoid wasting time on decomposing it. These unnecessary cases can be detected by imagining that $\mathcal{A}$ is a laser with range L. As the laser sweeps around, only features that are contacted by the laser are relevant. Any features that are hidden from view of the laser correspond to unnecessary boundaries.

After the cell decomposition has been constructed in $\mathbb{R}^2$, it needs to be lifted into $\mathbb{R}^2 \times [0, 2\pi]/\sim$. This generates a cylinder of 3-cells above each 2D noncritical region, R. The roadmap could easily be defined to have a vertex for every 3-cell and 2-cell, which would be consistent with previous cell decompositions; however, vertices at 2-cells are not generated here to make the coming example easier to understand. Each 3-cell, $\{R, [f_i, f_{i+1}]\}$, corresponds to the vertex in a roadmap. The roadmap edges connect neighboring 3-cells that have a 2-cell as part of their common boundary. This means that in $\mathbb{R}^2$ they share a one-dimensional portion of a critical curve.

Constructing the roadmap The problem is to determine which 3-cells are actually adjacent. Figure 6.27 depicts the cases in which connections need to be made. The xy plane is represented as one axis (imagine looking in a direction parallel to it). Consider two neighboring 2-cells (noncritical regions), R and R', in the plane. It is assumed that a 1-cell (critical curve) in $\mathbb{R}^2$ separates them. The

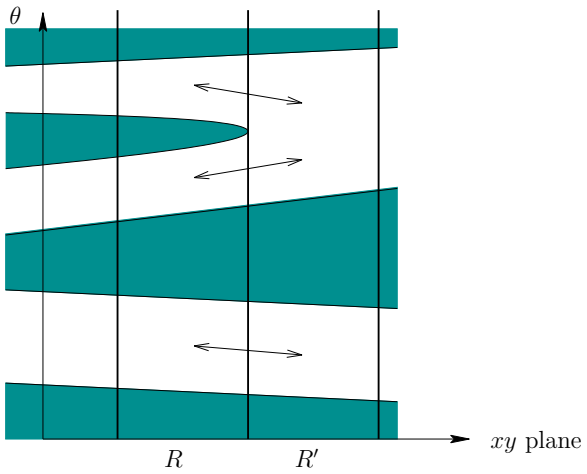


Figure 6.27: Connections are made between neighboring 3-cells that lie above neighboring noncritical regions.

task is to connect together 3-cells in the cylinders above R and R'. If neighboring cells share the same feature pair, then they are connected. This means that $\{R, [f_i, f_{i+1}]\}$ and $\{R', [f_i, f_{i+1}]\}$ must be connected. In some cases, one feature may change, while the interval of orientations remains unchanged. This may happen, for example, when the robot changes from contacting an edge to contacting a vertex of the edge. In these cases, a connection must also be made. One case illustrated in Figure 6.27 is when a splitting or merging of orientation intervals occurs. Traveling from R to R', the figure shows two regions merging into one. In this case, connections must be made from each of the original two 3-cells to the merged 3-cell. When constructing the roadmap edges, sample points of both the 3-cells and 2-cells should be used to ensure collision-free paths are obtained, as in the case of the vertical decomposition in Section 6.2.2. Figure 6.28 depicts the cells for the example in Figure 6.26. Each noncritical region has between one and three cells above it. Each of the various cells is indicated by a shortened robot that points in the general direction of the cell. The connections between the cells are also shown. Using the noncritical region and feature names from Figure 6.26, the resulting roadmap is depicted abstractly in Figure 6.29. Each vertex represents a 3-cell in $\mathcal{C}_{free}$, and each edge represents the crossing of a 2-cell between adjacent 3-cells. To make the roadmap consistent with previous roadmaps, we could insert a vertex into every edge and force the path to travel through the sample point of the corresponding 2-cell.

Once the roadmap has been constructed, it can be used in the same way as other roadmaps in this chapter to solve a query. Many implementation details have been neglected here. Due to the fifth case, some of the region boundaries in $\mathbb{R}^2$ are fourth-degree algebraic curves. Ways to prevent the explicit characterization of every noncritical region boundary, and other implementation details, are covered in [56]. Some of these details are also summarized in [588].

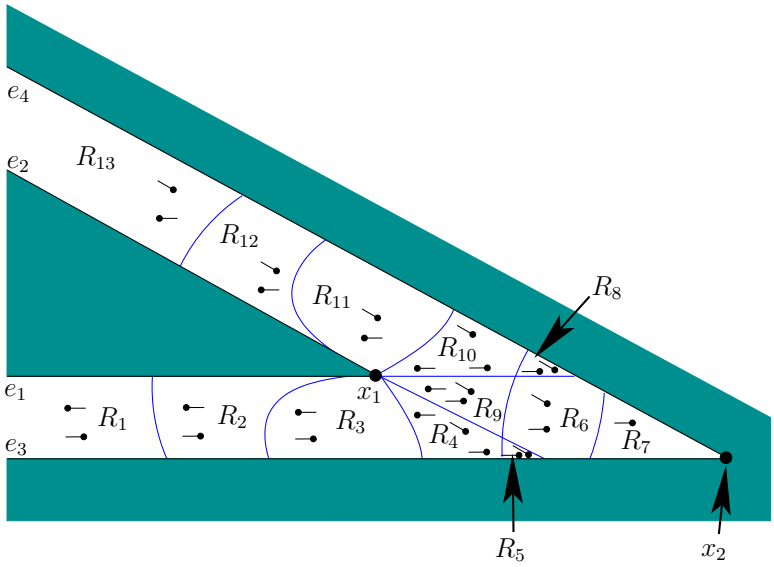


Figure 6.28: A depiction of the 3-cells above the noncritical regions. Sample rod orientations are shown for each cell (however, the rod length is shortened for clarity). Edges between cells are shown in Figure 6.29.

Complexity How many cells can there possibly be in the worst case? First count the number of noncritical regions in $\mathbb{R}^2$. There are $O(n)$ different ways to generate critical curves of the first three types because each corresponds to a single feature. Unfortunately, there are $O(n^2)$ different ways to generate bitangents and the Conchoid of Nicomedes because these are based on pairs of features. Assuming no self-intersections, a collection of $O(n^2)$ curves in $\mathbb{R}^2$, may intersect to generate at most $O(n^4)$ regions. Above each noncritical region in $\mathbb{R}^2$, there could be a cylinder of $O(n)$ 3-cells. Therefore, the size of the cell decomposition is $O(n^5)$ in the worst case. In practice, however, it is highly unlikely that all of these intersections will occur, and the number of cells is expected to be reasonable. In [851], an $O(n^5)$-time algorithm is given to construct the cell decomposition. Algorithms that have much better running time are mentioned in Section 6.5.3, but they are more complicated to understand and implement.

6.4 Computational Algebraic Geometry

This section presents algorithms that are so general that they solve any problem of Formulation 4.1 and even the closed-chain problems of Section 4.4. It is amazing that such algorithms exist; however, it is also unfortunate that they are both extremely challenging to implement and not efficient enough for most applications. The concepts and tools of this section were mostly developed in the context of computational real algebraic geometry [77, 250]. They are powerful enough to conquer numerous problems in robotics, computer vision, geometric modeling, computer-

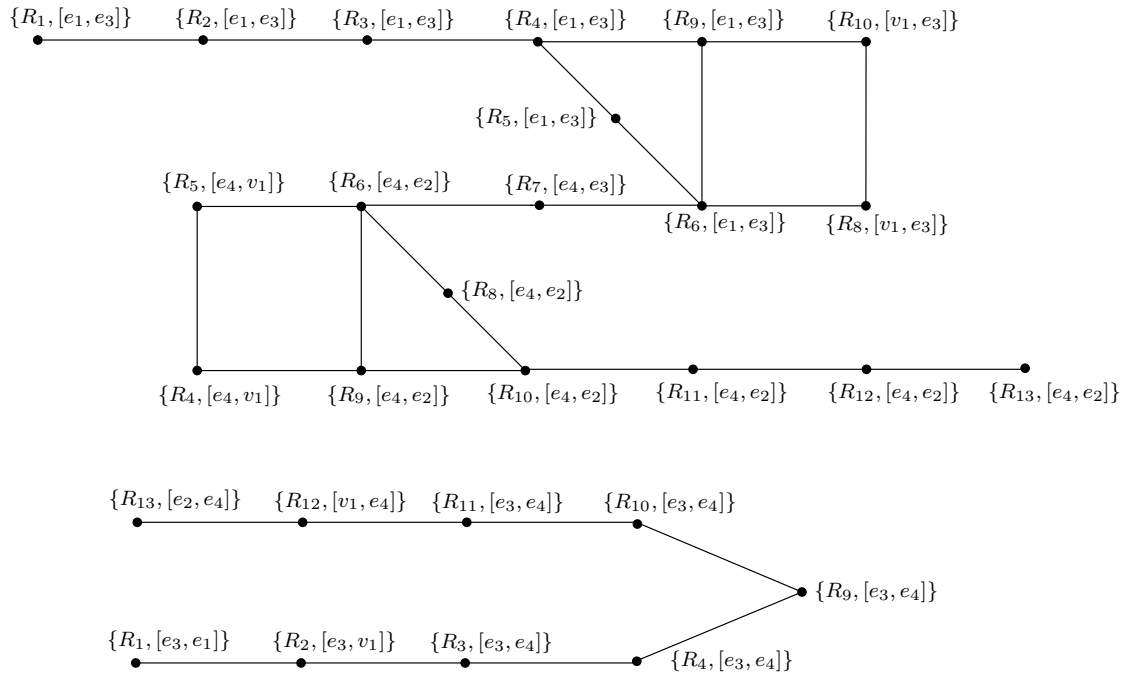


Figure 6.29: The roadmap corresponding to the example in Figure 6.26.

aided design, and geometric theorem proving. One of these problems happens to be motion planning, for which the connection to computational algebraic geometry was first recognized in [852].

6.4.1 Basic Definitions and Concepts

This section builds on the semi-algebraic model definitions from Section 3.1 and the polynomial definitions from Section 4.4.1. It will be assumed that $\mathcal{C} \subseteq \mathbb{R}^n$, which could for example arise by representing each copy of $SO(2)$ or $SO(3)$ in its 2×2 or 3×3 matrix form. For example, in the case of a 3D rigid body, we know that $\mathcal{C} = \mathbb{R}^3 \times \mathbb{RP}^3$, which is a six-dimensional manifold, but it can be embedded in $\mathbb{R}^{12}$, which is obtained from the Cartesian product of $\mathbb{R}^3$ and the set of all 3×3 matrices. The constraints that force the matrices to lie in $SO(2)$ or $SO(3)$ are polynomials, and they can therefore be added to the semi-algebraic models of $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$. If the dimension of $\mathcal{C}$ is less than n, then the algorithm presented below is sufficient, but there are some representation and complexity issues that motivate using a special parameterization of $\mathcal{C}$ to make both dimensions the same while altering the topology of $\mathcal{C}$ to become homeomorphic to $\mathbb{R}^n$. This is discussed briefly in Section 6.4.2.

Suppose that the models in $\mathbb{R}^n$ are all expressed using polynomials from $\mathbb{Q}[x_1, \ldots, x_n]$, the set of polynomials[6] over the field of rational numbers $\mathbb{Q}$. Let $f \in \mathbb{Q}[x_1, \ldots, x_n]$ denote a polynomial.

[6]It will be explained shortly why $\mathbb{Q}[x_1, \ldots, x_n]$ is preferred over $\mathbb{R}[x_1, \ldots, x_n]$.

Tarski sentences Recall the logical predicates that were formed in Section 3.1. They will be used again here, but now they are defined with a little more flexibility. For any $f \in \mathbb{Q}[x_1, \ldots, x_n]$, an *atom* is an expression of the form $f \bowtie 0$, in which $\bowtie$ may be any relation in the set $\{=, \neq, <, >, \leq, \geq\}$. In Section 3.1, such expressions were used to define logical predicates. Here, we assume that relations other than $\leq$ can be used and that the vector of polynomial variables lies in $\mathbb{R}^n$.

A *quantifier-free formula*, $\phi(x_1, \ldots, x_n)$, is a logical predicate composed of atoms and logical connectives, "and," "or," and "not," which are denoted by $\wedge$, $\vee$, and $\neg$, respectively. Each atom itself is considered as a logical predicate that yields TRUE if and only if the relation is satisfied when the polynomial is evaluated at the point $(x_1, \ldots, x_n) \in \mathbb{R}^n$.

Example 6.2 (An Example Predicate) Let ϕ be a predicate over $\mathbb{R}^3$, defined as

$$\phi(x_1, x_2, x_3) = (x_1^2 x_3 - x_2^4 < 0) \vee \big(\neg(3x^2x^3 \neq 0) \wedge (2x_3^2 - x_1x_2x_3 + 2 \geq 0)\big). \quad (6.8)$$

The precedence order of the connectives follows the laws of Boolean algebra. ■

Let a *quantifier* $\mathcal{Q}$ be either of the symbols, $\forall$, which means "for all," or $\exists$, which means "there exists." A *Tarski sentence* Φ is a logical predicate that may additionally involve quantifiers on some or all of the variables. In general, a Tarski sentence takes the form

$$\Phi(x_1, \ldots, x_{n-k}) = (\mathcal{Q}z_1)(\mathcal{Q}z_2) \ldots (\mathcal{Q}z_k)\ \phi(z_1, \ldots, z_k, x_1, \ldots, x_{n-k}), \quad (6.9)$$

in which the z_i are the *quantified variables*, the x_i are the *free variables*, and ϕ is a quantifier-free formula. The quantifiers do not necessarily have to appear at the left to be a valid Tarski sentence; however, any expression can be manipulated into an equivalent expression that has all quantifiers in front, as shown in (6.9). The procedure for moving quantifiers to the front is as follows [705]: 1) Eliminate any redundant quantifiers; 2) rename some of the variables to ensure that the same variable does not appear both free and bound; 3) move negation symbols as far inward as possible; and 4) push the quantifiers to the left.

Example 6.3 (Several Tarski Sentences) Tarski sentences that have no free variables are either TRUE or FALSE in general because there are no arguments on which the results depend. The sentence

$$\Phi = \forall x \exists y\ (x^2 - y < 0), \quad (6.10)$$

is TRUE because for any $x \in \mathbb{R}$, some $y \in \mathbb{R}$ can always be chosen so that $y > x^2$. In the general notation of (6.9), this example becomes $\mathcal{Q}z_1 = \forall x$, $\mathcal{Q}z_2 = \exists y$, and $\phi(z_1, z_2) = (x^2 - y < 0)$.

Swapping the order of the quantifiers yields the Tarski sentence

$$\Phi = \exists y \forall x\ (x^2 - y < 0), \quad (6.11)$$

which is FALSE because for any y, there is always an x such that $x^2 > y$.

Now consider a Tarski sentence that has a free variable:

$$\Phi(z) = \exists y \forall x \; (x^2 - zx^2 - y < 0). \tag{6.12}$$

This yields a function $\Phi : \mathbb{R} \rightarrow \{\text{TRUE}, \text{FALSE}\}$, in which

$$\Phi(z) = \begin{cases} \text{TRUE} & \text{if } z \geq 1 \\ \text{FALSE} & \text{if } z < 1. \end{cases} \tag{6.13}$$

An equivalent quantifier-free formula ϕ can be defined as $\phi(z) = (z > 1)$, which takes on the same truth values as the Tarski sentence in (6.12). This might make you wonder whether it is always possible to make a simplification that eliminates the quantifiers. This is called the *quantifier-elimination problem*, which will be explained shortly. ■

The decision problem The sentences in (6.10) and (6.11) lead to an interesting problem. Consider the set of all Tarski sentences that have no free variables. The subset of these that are TRUE comprise the *first-order theory of the reals*. Can an algorithm be developed to determine whether such a sentence is true? This is called the *decision problem* for the first-order theory of the reals. At first it may appear hopeless because $\mathbb{R}^n$ is uncountably infinite, and an algorithm must work with a finite set. This is a familiar issue faced throughout motion planning. The sampling-based approaches in Chapter 5 provided one kind of solution. This idea could be applied to the decision problem, but the resulting lack of completeness would be similar. It is not possible to check all possible points in $\mathbb{R}^n$ by sampling. Instead, the decision problem can be solved by constructing a combinatorial representation that exactly represents the decision problem by partitioning $\mathbb{R}^n$ into a finite collection of regions. Inside of each region, only one point needs to be checked. This should already seem related to cell decompositions in motion planning; it turns out that methods developed to solve the decision problem can also conquer motion planning.

The quantifier-elimination problem Another important problem was exemplified in (6.12). Consider the set of all Tarski sentences of the form (6.9), which may or may not have free variables. Can an algorithm be developed that takes a Tarski sentence Φ and produces an equivalent quantifier-free formula ϕ? Let $x_1, \ldots, x_n$ denote the free variables. To be equivalent, both must take on the same true values over $\mathbb{R}^n$, which is the set of all assignments $(x_1, \ldots, x_n)$ for the free variables.

Given a Tarski sentence, (6.9), the *quantifier-elimination problem* is to find a quantifier-free formula ϕ such that

$$\Phi(x_1, \ldots, x_n) = \phi(x_1, \ldots, x_n) \tag{6.14}$$

for all $(x_1, \ldots, x_n) \in \mathbb{R}^n$. This is equivalent to constructing a semi-algebraic model because ϕ can always be expressed in the form

$$\phi(x_1, \ldots, x_n) = \bigvee_{i=1}^{k} \bigwedge_{j=1}^{m_i} (f_{i,j}(x_1, \ldots, x_n) \bowtie 0), \tag{6.15}$$

in which $\bowtie$ may be either $<$, $=$, or $>$. This appears to be the same (3.6), except that (6.15) uses the relations $<$, $=$, and $>$ to allow open and closed semi-algebraic sets, whereas (3.6) only used $\leq$ to construct closed semi-algebraic sets for $\mathcal{O}$ and $\mathcal{A}$.

Once again, the problem is defined on $\mathbb{R}^n$, which is uncountably infinite, but an algorithm must work with a finite representation. This will be achieved by the cell decomposition technique presented in Section 6.4.2.

Semi-algebraic decomposition As stated in Section 6.3.1, motion planning inside of each cell in a complex should be trivial. To solve the decision and quantifier-elimination problems, a cell decomposition was developed for which these problems become trivial in each cell. The decomposition is designed so that only a single point in each cell needs to be checked to solve the decision problem.

The semi-algebraic set $Y \subseteq \mathbb{R}^n$ that is expressed with (6.15) is

$$Y = \bigcup_{i=1}^{k} \bigcap_{j=1}^{m_i} \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid \mathrm{sgn}(f_{i,j}(x_1, \ldots, x_n)) = s_{i,j}\}, \tag{6.16}$$

in which sgn is the sign function, and each $s_{i,j} \in \{-1, 0, 1\}$, which is the range of sgn. Once again the nice relationship between set-theory and logic, which was described in Section 3.1, appears here. We convert from a set-theoretic description to a logical predicate by changing $\cup$ and $\cap$ to $\vee$ and $\wedge$, respectively.

Let $\mathcal{F}$ denote the set of $m = \sum_{i=1}^{k} m_i$ polynomials that appear in (6.16). A *sign assignment* with respect to $\mathcal{F}$ is a vector-valued function, $\mathrm{sgn}_{\mathcal{F}} : \mathbb{R}^n \rightarrow \{-1, 0, 1\}^m$. Each $f \in \mathcal{F}$ has a corresponding position in the sign assignment vector. At this position, the sign, $\mathrm{sgn}(f(x_1, \ldots, x_n)) \in \{-1, 0, 1\}$, appears. A *semi-algebraic decomposition* is a partition of $\mathbb{R}^n$ into a finite set of connected regions that are each *sign invariant*. This means that inside of each region, $\mathrm{sgn}_{\mathcal{F}}$ must remain constant. The regions will not be called *cells* because a semi-algebraic decomposition is not necessarily a singular complex as defined in Section 6.3.1; the regions here may contain holes.

Example 6.4 (Sign assignment) Recall Example 3.1 and Figure 3.4 from Section 3.1.2. Figure 3.4a shows a sign assignment for a case in which there is only one polynomial, $\mathcal{F} = \{x^2 + y^2 - 4\}$. The sign assignment is defined as

$$\mathrm{sgn}_{\mathcal{F}}(x, y) = \begin{cases} -1 & \text{if } x^2 + y^2 - 4 < 0 \\ 0 & \text{if } x^2 + y^2 - 4 = 0 \\ 1 & \text{if } x^2 + y^2 - 4 > 0. \end{cases} \tag{6.17}$$

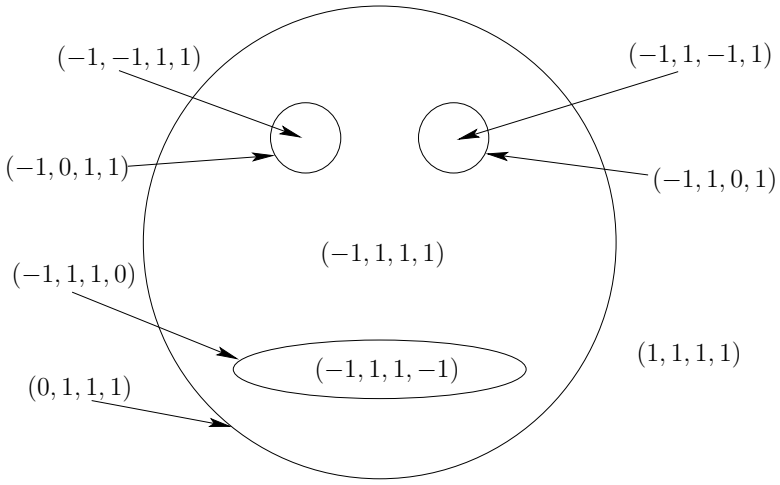


Figure 6.30: A semi-algebraic decomposition of the gingerbread face yields 9 sign-invariant regions.

Now consider the sign assignment $\text{sgn}_{\mathcal{F}}$, shown in Figure 6.30 for the gingerbread face of Figure 3.4b. The polynomials of the semi-algebraic model are $\mathcal{F} = \{f_1, f_2, f_3, f_4\}$, as defined in Example 3.1. In order, these are the "head," "left eye," "right eye," and "mouth." The sign assignment produces a four-dimensional vector of signs. Note that if (x, y) lies on one of the zeros of a polynomial in $\mathcal{F}$, then a 0 appears in the sign assignment. If the curves of two or more of the polynomials had intersected, then the sign assignment would produce more than one 0 at the intersection points.

For the semi-algebraic decomposition for the gingerbread face in Figure 6.30, there are nine regions. Five 2D regions correspond to: 1) being outside of the face, 2)inside of the left eye, 3) inside of the right eye, 4) inside of the mouth, and 5) inside of the face but outside of the mouth and eyes. There are four 1D regions, each of which corresponds to points that lie on one of the zero sets of a polynomial. The resulting decomposition is not a singular complex because the $(-1, 1, 1, 1)$ region contains three holes. ■

A decomposition such as the one in Figure 6.30 would not be very useful for motion planning because of the holes in the regions. Further refinement is needed for motion planning, which is fortunately produced by cylindrical algebraic decomposition. On the other hand, any semi-algebraic decomposition is quite useful for solving the decision problem. Only one point needs to be checked inside of each region to determine whether some Tarski sentence that has no free variables is true. Why? If the polynomial signs cannot change over some region, then the TRUE/FALSE value of the corresponding logical predicate, Φ, cannot change. Therefore, it sufficient only to check one point per sign-invariant region.

6.4.2 Cylindrical Algebraic Decomposition

Cylindrical algebraic decomposition is a general method that produces a cylindrical decomposition in the same sense considered in Section 6.3.2 for polygons in $\mathbb{R}^2$ and also the decomposition in Section 6.3.4 for the line-segment robot. It is also referred to as *Collins decomposition* after its original developer [40, 232, 233]. The decomposition in Figure 6.19 can even be considered as a cylindrical algebraic decomposition for a semi-algebraic set in which every geometric primitive is a linear polynomial. In this section, such a decomposition is generalized to any semi-algebraic set in $\mathbb{R}^n$.

The idea is to develop a sequence of projections that drops the dimension of the semi-algebraic set by one each time. Initially, the set is defined over $\mathbb{R}^n$, and after one projection, a semi-algebraic set is obtained in $\mathbb{R}^{n-1}$. Eventually, the projection reaches $\mathbb{R}$, and a univariate polynomial is obtained for which the zeros are at the critical places where cell boundaries need to be formed. A cell decomposition of 1-cells (intervals) and 0-cells is formed by partitioning $\mathbb{R}$. The sequence is then reversed, and decompositions are formed from $\mathbb{R}^2$ up to $\mathbb{R}^n$. Each iteration starts with a cell decomposition in $\mathbb{R}^i$ and lifts it to obtain a cylinder of cells in $\mathbb{R}^{i+1}$. Figure 6.35 shows how the decomposition looks for the gingerbread example; since $n = 2$, it only involves one projection and one lifting.

Semi-algebraic projections are semi-algebraic The following is implied by the *Tarski-Seidenberg Theorem* [77]:

> *A projection of a semi-algebraic set from dimension n to dimension $n - 1$ is a semi-algebraic set.*

This gives a kind of closure of semi-algebraic sets under projection, which is required to ensure that every projection of a semi-algebraic set in $\mathbb{R}^i$ leads to a semi-algebraic set in $\mathbb{R}^{i-1}$. This property is actually not true for (real) algebraic varieties, which were introduced in Section 4.4.1. Varieties are defined using only the $=$ relation and are not closed under the projection operation. Therefore, it is a good thing (not just a coincidence!) that we are using semi-algebraic sets.

Real algebraic numbers As stated previously, the sequence of projections ends with a univariate polynomial over $\mathbb{R}$. The sides of the cells will be defined based on the precise location of the roots of this polynomial. Furthermore, representing a sample point for a cell of dimension k in a complex in $\mathbb{R}^n$ for $k < n$ requires perfect precision. If the coordinates are slightly off, the point will lie in a different cell. This raises the complicated issue of how these roots are represented and manipulated in a computer.

For univariate polynomials of degree 4 or less, formulas exist to compute all of the roots in terms of functions of square roots and higher order roots. From Galois theory [469, 769], it is known that such formulas and nice expressions for roots do not exist for most higher degree polynomials, which can certainly arise

in the complicated semi-algebraic models that are derived in motion planning. The roots in $\mathbb{R}$ could be any real number, and many real numbers require infinite representations.

One way of avoiding this mess is to assume that only polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$ are used, instead of the more general $\mathbb{R}[x_1, \ldots, x_n]$. The field $\mathbb{Q}$ is not *algebraically closed* because zeros of the polynomials lie outside of $\mathbb{Q}^n$. For example, if $f(x_1) = x_1^2 - 2$, then $f = 0$ for $x_1 = \pm\sqrt{2}$, and $\sqrt{2} \notin \mathbb{Q}$. However, some elements of $\mathbb{R}$ can never be roots of a polynomial in $\mathbb{Q}[x_1, \ldots, x_n]$.

The set $\mathbb{A}$ of all real roots to all polynomials in $\mathbb{Q}[x]$ is called the set of *real algebraic numbers*. The set $\mathbb{A} \subset \mathbb{R}$ actually represents a field (recall from Section 4.4.1). Several nice algorithmic properties of the numbers in $\mathbb{A}$ are 1) they all have finite representations, 2) addition and multiplication operations on elements of $\mathbb{A}$ can be computed in polynomial time, and 3) conversions between different representations of real algebraic numbers can be performed in polynomial time. This means that all operations can be computed efficiently without resorting to some kind of numerical approximation. In some applications, such approximations are fine; however, for algebraic decompositions, they destroy critical information by potentially confusing roots (e.g., how can we know for sure whether a polynomial has a double root or just two roots that are very close together?).

The details are not presented here, but there are several methods for representing real algebraic numbers and the corresponding algorithms for manipulating them efficiently. The running time of cylindrical algebraic decomposition ultimately depends on this representation. In practice, a numerical root-finding method that has a precision parameter, ϵ, can be used by choosing ϵ small enough to ensure that roots will not be confused. A sufficiently small value can be determined by applying *gap theorems*, which give lower bounds on the amount of real root separation, expressed in terms of the polynomial coefficients [173]. Some methods avoid requiring a precision parameter. One well-known example is the derivation of a Sturm sequence of polynomials based on the given polynomial. The polynomials in the Sturm sequence are then used to find isolating intervals for each of the roots [77]. The polynomial, together with its isolating interval, can be considered as an exact root representation. Algebraic operations can even be performed using this representation in time $O(d \lg^2 d)$, in which d is the degree of the polynomial [852]. See [77, 173, 852] for detailed presentations on the exact representation and calculation with real algebraic numbers.

One-dimensional decomposition To explain the cylindrical algebraic decomposition method, we first perform a semi-algebraic decomposition of $\mathbb{R}$, which is the final step in the projection sequence. Once this is explained, then the multi-dimensional case follows more easily.

Let $\mathcal{F}$ be a set of m univariate polynomials,

$$\mathcal{F} = \{f_i \in \mathbb{Q}[x] \mid i = 1, \ldots, m\}, \tag{6.18}$$

which are used to define some semi-algebraic set in $\mathbb{R}$. The polynomials in $\mathcal{F}$ could

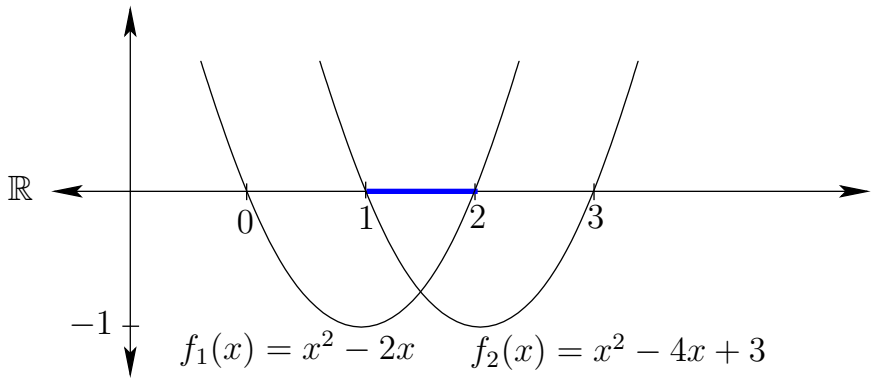


Figure 6.31: Two parabolas are used to define the semi-algebraic set $[1, 2]$.

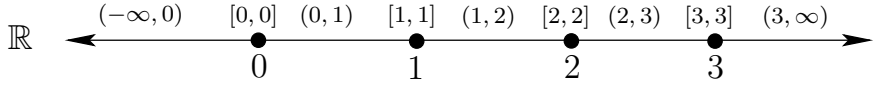


Figure 6.32: A semi-algebraic decomposition for the polynomials in Figure 6.31.

come directly from a quantifier-free formula ϕ (which could even appear inside of a Tarski sentence, as in (6.9)).

Define a single polynomial as $f = \prod_{i=1}^{m} f_i$. Suppose that f has k distinct, real roots, which are sorted in increasing order:

$$-\infty < \beta_1 < \beta_2 < \cdots < \beta_{i-1} < \beta_i < \beta_{i+1} < \cdots < \beta_k < \infty. \quad (6.19)$$

The one-dimensional semi-algebraic decomposition is given by the following sequence of alternating 1-cells and 0-cells:

$$\begin{aligned} &(-\infty, \beta_1),\ [\beta_1, \beta_1],\ (\beta_1, \beta_2),\ \ldots,\ (\beta_{i-1}, \beta_i),\ [\beta_i, \beta_i], \\ &(\beta_i, \beta_{i+1}),\ \ldots,\ [\beta_k, \beta_k],\ (\beta_k, \infty). \end{aligned} \quad (6.20)$$

Any semi-algebraic set that can be expressed using the polynomials in $\mathcal{F}$ can also be expressed as the union of some of the 0-cells and 1-cells given in (6.20). This can also be considered as a singular complex (it can even be considered as a simplicial complex, but this does not extend to higher dimensions).

Sample points can be generated for each of the cells as follows. For the unbounded cells $[-\infty, \beta_1)$ and $(\beta_k, \infty]$, valid samples are $\beta_1 - 1$ and $\beta_k + 1$, respectively. For each finite 1-cell, (β_i, β_{i+1}), the midpoint $(\beta_i + \beta_{i+1})/2$ produces a sample point. For each 0-cell, $[\beta_i, \beta_i]$, the only choice is to use β_i as the sample point.

Example 6.5 (One-Dimensional Decomposition) Figure 6.31 shows a semi-algebraic subset of $\mathbb{R}$ that is defined by two polynomials, $f_1(x) = x^2 - 2x$ and $f_2(x) = x^2 - 4x + 3$. Here, $\mathcal{F} = \{f_1, f_2\}$. Consider the quantifier-free formula

$$\phi(x) = (x^2 - 2x \geq 0) \wedge (x^2 - 4x + 3 \geq 0). \quad (6.21)$$

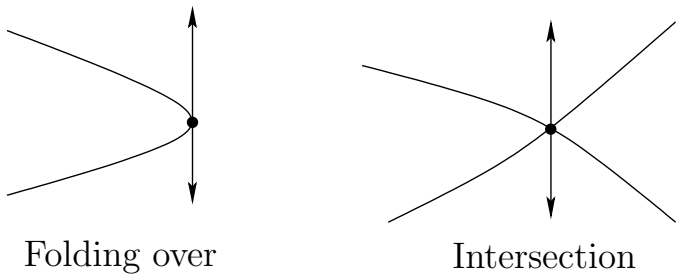


Figure 6.33: Critical points occur either when the surface folds over in the vertical direction or when surfaces intersect.

The semi-algebraic decomposition into five 1-cells and four 0-cells is shown in Figure 6.32. Each cell is sign invariant. The sample points for the 1-cells are -1, $1/2$, $3/2$, $5/2$, and 4, respectively. The sample points for the 0-cells are 0, 1, 2, and 3, respectively.

A decision problem can be nicely solved using the decomposition. Suppose a Tarski sentence that uses the polynomials in $\mathcal{F}$ has been given. Here is one possibility:

$$\Phi = \exists x[(x^2 - 2x \geq 0) \wedge (x^2 - 4x + 3 \geq 0)] \tag{6.22}$$

The sample points alone are sufficient to determine whether Φ is TRUE or FALSE. Once $x = 1$ is attempted, it is discovered that Φ is TRUE. The quantifier-elimination problem cannot yet be considered because more dimensions are needed. ■

The inductive step to higher dimensions Now consider constructing a cylindrical algebraic decomposition for $\mathbb{R}^n$ (note the decomposition is actually semi-algebraic). Figure 6.35 shows an example for $\mathbb{R}^2$. First consider how to iteratively project the polynomials down to $\mathbb{R}$ to ensure that when the decomposition of $\mathbb{R}^n$ is constructed, the sign-invariant property is maintained. The resulting decomposition corresponds to a singular complex.

There are two cases that cause cell boundaries to be formed, as shown in Figure 6.33. Let $\mathcal{F}_n$ denote the original set of polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$ that are used to define the semi-algebraic set (or Tarski sentence) in $\mathbb{R}^n$. Form a single polynomial $f = \prod_{i=1}^{m} f_i$. Let $f' = \partial f / \partial x_n$, which is also a polynomial. Let $g = GCD(f, f')$, which is the greatest common divisor of f and f'. The set of zeros of g is the set of all points that are zeros of both f and f'. Being a zero of f' means that the surface given by $f = 0$ does not vary locally when perturbing x_n. These are places where a cell boundary needs to be formed because the surface may fold over itself in the x_n direction, which is not permitted for a cylindrical decomposition. Another place where a cell boundary needs to be formed is at the intersection of two or more polynomials in $\mathcal{F}_n$. The projection technique from $\mathbb{R}^n$ to $\mathbb{R}^{n-1}$ generates a set, $\mathcal{F}_{n-1}$, of polynomials in $\mathbb{Q}[x_1, \ldots, x_{n-1}]$ that satisfies these requirements.

The polynomials $\mathcal{F}_{n-1}$ have the property that at least one contains a zero point below every point in $x \in \mathbb{R}^n$ for which $f(x) = 0$ and $f'(x) = 0$, or polynomials in $\mathcal{F}_n$ intersect. The projection method that constructs $\mathcal{F}_{n-1}$ involves computing *principle subresultant coefficients*, which are covered in [77, 853]. Resultants, of which the subresultants are an extension, are covered in [250].

The polynomials in $\mathcal{F}_{n-1}$ are then projected to $\mathbb{R}^{n-2}$ to obtain $\mathcal{F}_{n-2}$. This process continues until $\mathcal{F}_1$ is obtained, which is a set of polynomials in $\mathbb{Q}[x_1]$. A one-dimensional decomposition is formed, as defined earlier. From $\mathcal{F}_1$, a single polynomial is formed by taking the product, and $\mathbb{R}$ is partitioned into 0-cells and 1-cells. We next describe the process of lifting a decomposition over $\mathbb{R}^{i-1}$ up to $\mathbb{R}^i$. This technique is applied iteratively until $\mathbb{R}^n$ is reached.

Assume inductively that a cylindrical algebraic decomposition has been computed for a set of polynomials $\mathcal{F}_{i-1}$ in $\mathbb{Q}[x_1, \ldots, x_{i-1}]$. The decomposition consists of k-cells for which $0 \leq k \leq i$. Let $p = (x_1, \ldots, x_{i-1}) \in \mathbb{R}^{i-1}$. For each one of the k-cells C_{i-1}, a *cylinder* over C_{i-1} is defined as the $(k+1)$-dimensional set

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1}\}. \tag{6.23}$$

The cylinder is sliced into a strip of k-dimensional and $k+1$-dimensional cells by using polynomials in $\mathcal{F}_i$. Let f_j denote one of the ℓ slicing polynomials in the cylinder, sorted in increasing x_i order as f_1, f_2, …, f_j, f_{j+1}, …, f_ℓ. The following kinds of cells are produced (see Figure 6.34):

1. **Lower unbounded sector:**

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } x_i < f_1(p)\}. \tag{6.24}$$

2. **Section:**

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } x_i = f_j(p)\}. \tag{6.25}$$

3. **Bounded sector:**

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } f_j(p) < x_i < f_{j+1}(p)\}. \tag{6.26}$$

4. **Upper unbounded sector:**

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } f_\ell(p) < x_i\}. \tag{6.27}$$

There is one degenerate possibility in which there are no slicing polynomials and the cylinder over C_{i-1} can be extended into one unbounded cell. In general, the sample points are computed by picking a point in $p \in C_{i-1}$ and making a vertical column of samples of the form (p, x_i). A polynomial in $\mathbb{Q}[x_i]$ can be generated, and the samples are placed using the same assignment technique that was used for the one-dimensional decomposition.

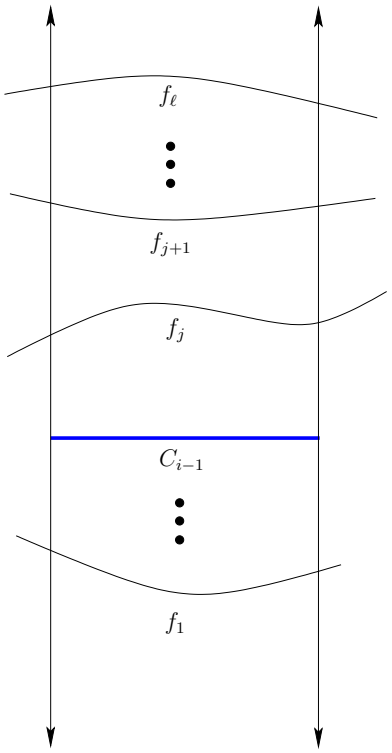


Figure 6.34: A cylinder over every k-cell C_{i-1} is formed. A sequence of polynomials, $f_1, \ldots, f_\ell$, slices the cylinder into k-dimensional sections and $(k+1)$-dimensional sectors.

Example 6.6 (Mutilating the Gingerbread Face) Figure 6.35 shows a cylindrical algebraic decomposition of the gingerbread face. Observe that the resulting complex is very similar to that obtained in Figure 6.19. ■

Note that the cells do not necessarily project onto a rectangular set, as in the case of a higher dimensional vertical decomposition. For example, a generic n-cell C_n for a decomposition of $\mathbb{R}^n$ is described as the open set of $(x_1, \ldots, x_n) \in \mathbb{R}^n$ such that

- $C_0 < x_n < C_0'$ for some 0-cells $C_0, C_0' \in \mathbb{R}$, which are roots of some $f, f' \in \mathcal{F}_1$.
- (x_{n-1}, x_n) lies between C_1 and C_1' for some 1-cells C_1, C_1', which are zeros of some $f, f' \in \mathcal{F}_2$.

  $\vdots$

- $(x_{n-i+1}, \ldots, x_n)$ lies between C_{i-1} and C_{i-1}' for some i-cells C_{i-1}, C_{i-1}', which are zeros of some $f, f' \in \mathcal{F}_i$.

  $\vdots$

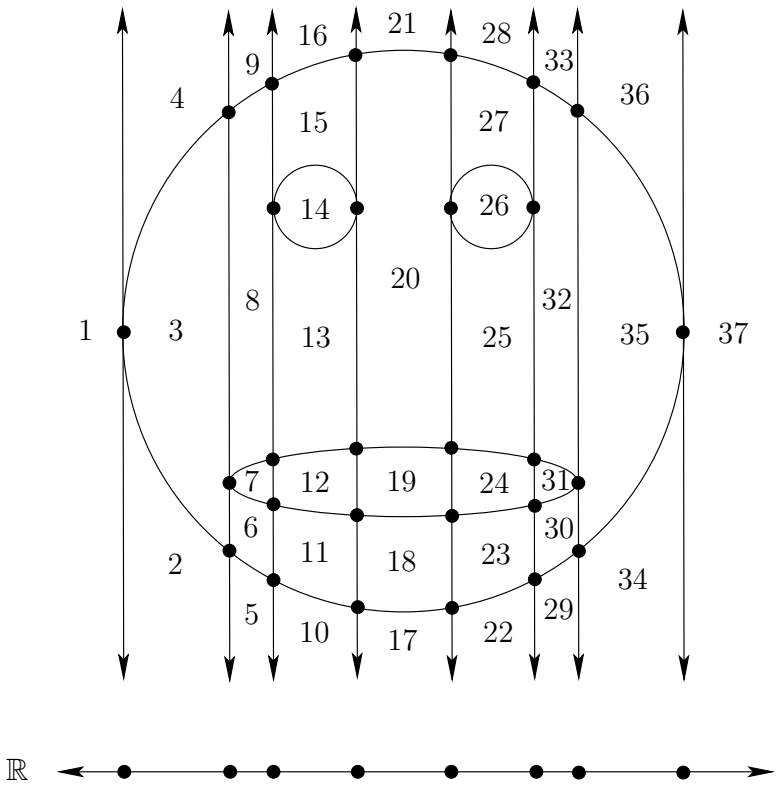


Figure 6.35: A cylindrical algebraic decomposition of the gingerbread face. There are 37 2-cells, 64 1-cells, and 28 0-cells. The straight 1-cells are intervals of the vertical lines, and the curved ones are portions of the zero set of a polynomial in $\mathcal{F}$. The decomposition of $\mathbb{R}$ is also shown.

- $(x_1, \ldots, x_n)$ lies between C_{n-1} and C'_{n-1} for some $(n-1)$-cells C_{n-1}, C'_{n-1}, which are zeros of some $f, f' \in \mathcal{F}_n$.

The resulting decomposition is sign invariant, which allows the decision and quantifier-elimination problems to be solved in finite time. To solve a decision problem, the polynomials in $\mathcal{F}_n$ are evaluated at every sample point to determine whether one of them satisfies the Tarski sentence. To solve the quantifier-elimination problem, note that any semi-algebraic sets that can be constructed from $\mathcal{F}_n$ can be defined as a union of some cells in the decomposition. For the given Tarski sentence, $\mathcal{F}_n$ is formed from all polynomials that are mentioned in the sentence, and the cell decomposition is performed. Once obtained, the sign information is used to determine which cells need to be included in the union. The resulting union of cells is designed to include only the points in $\mathbb{R}^n$ at which the Tarski sentence is TRUE.

Solving a motion planning problem Cylindrical algebraic decomposition is also capable of solving any of the motion planning problems formulated in Chapter 4. First assume that $\mathcal{C} = \mathbb{R}^n$. As for other decompositions, a roadmap is formed

in which every vertex is an n-cell and edges connect every pair of adjacent n-cells by traveling through an $(n-1)$-cell. It is straightforward to determine adjacencies inside of a cylinder, but there are several technical details associated with determining adjacencies of cells from different cylinders (pages 152–154 of [77] present an example that illustrates the problem). The cells of dimension less than $n-1$ are not needed for motion planning purposes (just as vertices were not needed for the vertical decomposition in Section 6.2.2). The query points q_I and q_G are connected to the roadmap depending on the cell in which they lie, and a discrete search is performed.

If $\mathcal{C} \subset \mathbb{R}^n$ and its dimension is k for $k < n$, then all of the interesting cells are of lower dimension. This occurs, for example, due to the constraints on the matrices to force them to lie in $SO(2)$ or $SO(3)$. This may also occur for problems from Section 4.4, in which closed chains reduce the degrees of freedom. The cylindrical algebraic decomposition method can still solve such problems; however, the exact root representation problem becomes more complicated when determining the cell adjacencies. A discussion of these issues appears in [852]. For the case of $SO(2)$ and $SO(3)$, this complication can be avoided by using *stereographic projection* to map $\mathbb{S}^1$ or $\mathbb{S}^3$ to $\mathbb{R}$ or $\mathbb{R}^3$, respectively. This mapping removes a single point from each, but the connectivity of $\mathcal{C}_{free}$ remains unharmed. The antipodal identification problem for unit quaternions represented by $\mathbb{S}^3$ also does not present a problem; there is a redundant copy of $\mathcal{C}$, which does not affect the connectivity.

The running time for cylindrical algebraic decomposition depends on many factors, but in general it is polynomial in the number of polynomials in $\mathcal{F}_n$, polynomial in the maximum algebraic degree of the polynomials, and doubly exponential in the dimension. Complexity issues are covered in more detail in Section 6.5.3.

6.4.3 Canny's Roadmap Algorithm

The doubly exponential running time of cylindrical algebraic decomposition inspired researchers to do better. It has been shown that quantifier elimination requires doubly exponential time [262]; however, motion planning is a different problem. Canny introduced a method that produces a roadmap directly from the semi-algebraic set, rather than constructing a cell decomposition along the way. Since there are doubly exponentially many cells in the cylindrical algebraic decomposition, avoiding this construction pays off. The resulting roadmap method of Canny solves the motion planning problem in time that is again polynomial in the number of polynomials and polynomial in the algebraic degree, but it is only singly exponential in dimension [170, 173]; see also [77].

Much like the other combinatorial motion planning approaches, it is based on finding critical curves and critical points. The main idea is to construct linear mappings from $\mathbb{R}^n$ to $\mathbb{R}^2$ that produce *silhouette curves* of the semi-algebraic sets. Performing one such mapping on the original semi-algebraic set yields a roadmap, but it might not preserve the original connectivity. Therefore, linear mappings from $\mathbb{R}^{n-1}$ to $\mathbb{R}^2$ are performed on some $(n-1)$-dimensional slices of the orig-

inal semi-algebraic set to yield more roadmap curves. This process is applied recursively until the slices are already one-dimensional. The resulting roadmap is formed from the union of all of the pieces obtained in the recursive calls. The resulting roadmap has the same connectivity as the original semi-algebraic set [173].

Suppose that $\mathcal{C} = \mathbb{R}^n$. Let $\mathcal{F} = \{f_1, \ldots, f_m\}$ denote the set of polynomials that define the semi-algebraic set, which is assumed to be a disjoint union of manifolds. Assume that each $f_i \in \mathbb{Q}[x_1, \ldots, x_n]$. First, a small perturbation to the input polynomials $\mathcal{F}$ is performed to ensure that every sign-invariant set of $\mathbb{R}^n$ is a manifold. This forces the polynomials into a kind of general position, which can be achieved with probability one using random perturbations; there are also deterministic methods to solve this problem. The general position requirements on the input polynomials and the 2D projection directions are fairly strong, which has stimulated more recent work that eliminates many of the problems [77]. From this point onward, it will be assumed that the polynomials are in general position.

Recall the sign-assignment function from Section 6.4.1. Each sign-invariant set is a manifold because of the general position assumption. Canny's method computes a roadmap for any k-dimensional manifold for $k < n$. Such a manifold has precisely $n - k$ signs that are 0 (which means that points lie precisely on the zero sets of $n - k$ polynomials in $\mathcal{F}$). At least one of the signs must be 0, which means that Canny's roadmap actually lies in $\partial\mathcal{C}_{free}$ (this technically is not permitted, but the algorithm nevertheless correctly decides whether a solution path exists through $\mathcal{C}_{free}$).

Recall that each f_i is a function, $f_i : \mathbb{R}^n \to \mathbb{R}$. Let x denote $(x_1, \ldots, x_n) \in \mathbb{R}^n$. The k polynomials that have zero signs can be put together sequentially to produce a mapping $\psi : \mathbb{R}^n \to \mathbb{R}^k$. The ith component of the vector $\psi(x)$ is $\psi_i(x) = f_i(x)$. This is closely related to the sign assignment function of Section 6.4.1, except that now the real value from each polynomial is directly used, rather than taking its sign.

Now introduce a function $g : \mathbb{R}^n \to \mathbb{R}^j$, in which either $j = 1$ or $j = 2$ (the general concepts presented below work for other values of j, but 1 and 2 are the only values needed for Canny's method). The function g serves the same purpose as a projection in cylindrical algebraic decomposition, but note that g immediately drops from dimension n to dimension 2 or 1, instead of dropping to $n - 1$ as in the case of cylindrical projections.

Let $h : \mathbb{R}^n \to \mathbb{R}^{k+j}$ denote a mapping constructed directly from ψ and g as follows. For the ith component, if $i \leq k$, then $h_i(x) = \psi_i(x) = f_i(x)$. Assume that $k + j \leq n$. If $i > k$, then $h_i(x) = g_{i-k}(x)$. Let $J_x(h)$ denote the *Jacobian* of h and

be defined at x as

$$J_x(h) = \begin{pmatrix} \frac{\partial h_1(x)}{\partial x_1} & \cdots & \frac{\partial h_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial h_{m+k}(x)}{\partial x_1} & \cdots & \frac{\partial h_{m+k}(x)}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_k(x)}{\partial x_1} & \cdots & \frac{\partial f_k(x)}{\partial x_n} \\ \\ \frac{\partial g_1(x)}{\partial x_1} & \cdots & \frac{\partial g_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial g_j(x)}{\partial x_1} & \cdots & \frac{\partial g_j(x)}{\partial x_n} \end{pmatrix}. \quad (6.28)$$

A point $x \in \mathbb{R}^n$ at which $J_x(h)$ is singular is called a *critical point*. The matrix is defined to be *singular* if every $(m+k)\times(m+k)$ subdeterminant is zero. Each of the first k rows of $J_x(h)$ calculates the surface normal to $f_i(x) = 0$. If these normals are not linearly independent of the directions given by the last j rows, then the matrix becomes singular. The following example from [169] nicely illustrates this principle.

Example 6.7 (Canny's Roadmap Algorithm) Let $n = 3$, $k = 1$, and $j = 1$. The zeros of a single polynomial f_1 define a 2D subset of $\mathbb{R}^3$. Let f_1 be the unit sphere, $\mathbb{S}^2$, defined as the zeros of the polynomial

$$f_1(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_3^2 - 1. \quad (6.29)$$

Suppose that $g : \mathbb{R}^3 \to \mathbb{R}$ is defined as $g(x_1, x_2, x_3) = x_1$. The Jacobian, (6.28), becomes

$$\begin{pmatrix} 2x_1 & 2x_2 & 2x_3 \\ 1 & 0 & 0 \end{pmatrix} \quad (6.30)$$

and is singular when all three of the possible 2×2 subdeterminants are zero. This occurs if and only if $x_2 = x_3 = 0$. This yields the critical points $(-1, 0, 0)$ and $(1, 0, 0)$ on $\mathbb{S}^2$. Note that this is precisely when the surface normals of $\mathbb{S}^2$ are parallel to the vector [1 0 0].

Now suppose that $j = 2$ to obtain $g : \mathbb{R}^3 \to \mathbb{R}^2$, and suppose $g(x_1, x_2, x_3) = (x_1, x_2)$. In this case, (6.28) becomes

$$\begin{pmatrix} 2x_1 & 2x_2 & 2x_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad (6.31)$$

which is singular if and only if $x_3 = 0$. The critical points are therefore the x_1x_2 plane intersected with $\mathbb{S}^3$, which yields the equator points (all $(x_1, x_2) \in \mathbb{R}^2$ such that $x_1^2 + x_2^2 = 1$). In this case, more points are generated because the matrix becomes degenerate for any surface normal of $\mathbb{S}^2$ that is parallel to [1 0 0], [0 1 0] or any linear combination of these. ■

The first mapping in Example 6.7 yielded two isolated critical points, and the second mapping yielded a one-dimensional set of critical points, which is referred to as a *silhouette*. The union of the silhouette and the isolated critical points yields a roadmap for $\mathbb{S}^2$. Now consider generalizing this example to obtain the full algorithm for general n and k. A linear mapping $g : \mathbb{R}^n \rightarrow \mathbb{R}^2$ is constructed that might not be axis-aligned as in Example 6.7 because it must be chosen in general position (otherwise degeneracies might arise in the roadmap). Define ψ to be the set of polynomials that become zero on the desired manifold on which to construct a roadmap. Form the matrix (6.28) and determine the silhouette. This is accomplished in general using subresultant techniques that were also needed for cylindrical algebraic decomposition; see [77, 173] for details. Let g_1 denote the first component of g, which yields a mapping $g_1 : \mathbb{R}^n \rightarrow \mathbb{R}$. Forming (6.28) using g_1 yields a finite set of critical points. Taking the union of the critical points and the silhouette produces part of the roadmap.

So far, however, there are no guarantees that the connectivity is preserved. To handle this problem, Canny's algorithm proceeds recursively. For each of the critical points $x \in \mathbb{R}^n$, an $n-1$-dimensional hyperplane through x is chosen for which the g_1 row of (6.28) is the normal (hence it is perpendicular in some sense to the flow of g_1). Inside of this hyperplane, a new g mapping is formed. This time a new direction is chosen, and the mapping takes the form $g : \mathbb{R}^{n-1} \rightarrow \mathbb{R}^2$. Once again, the silhouettes and critical points are found and added to the roadmap. This process is repeated recursively until the base case in which the silhouettes and critical points are directly obtained without forming g.

It is helpful to consider an example. Since the method involves a *sequence* of 2D projections, it is difficult to visualize. Problems in $\mathbb{R}^4$ and higher involve two or more 2D projections and would therefore be more interesting. An example over $\mathbb{R}^3$ is presented here, even though it unfortunately has only one projection; see [173] for another example over $\mathbb{R}^3$.

Example 6.8 (Canny's Algorithm on a Torus) Consider the 3D algebraic set shown in Figure 6.36. After defining the mapping $g(x_1, x_2, x_3) = (x_1, x_2)$, the roadmap shown in Figure 6.37 is obtained. The silhouettes are obtained from g, and the critical points are obtained from g_1 (this is the first component of g). Note that the original connectivity of the solid torus is not preserved because the inner ring does not connect to the outer ring. This illustrates the need to also compute the roadmap for lower dimensional slices. For each of the four critical points, the critical curves are computed for a plane that is parallel to the x_2x_3 plane and for which the x_1 position is determined by the critical point. The slice for one of the inner critical points is shown in Figure 6.38. In this case, the slice already has two dimensions. New silhouette curves are added to the roadmap to obtain the final result shown in Figure 6.39. ■

To solve a planning problem, the query points q_I and q_G are artificially declared to be critical points in the top level of recursion. This forces the algorithm to

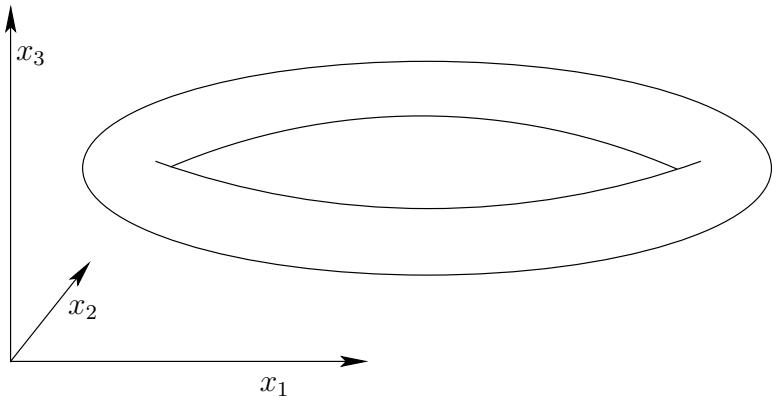


Figure 6.36: Suppose that the semi-algebraic set is a solid torus in $\mathbb{R}^3$.

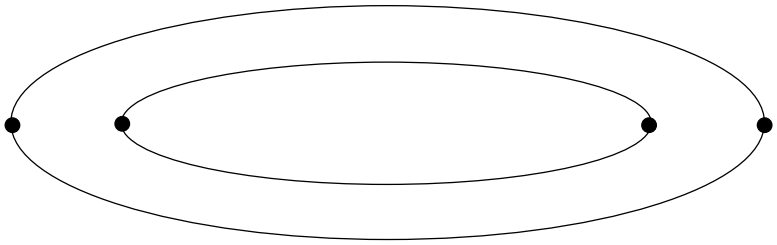


Figure 6.37: The projection into the x_1x_2 plane yields silhouettes for the inner and outer rings and also four critical points.

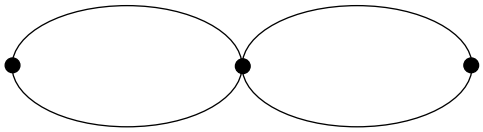


Figure 6.38: A slice taken for the inner critical points is parallel to the x_2x_3 plane. The roadmap for the slice connects to the silhouettes from Figure 6.37, thereby preserving the connectivity of the original set in Figure 6.36.

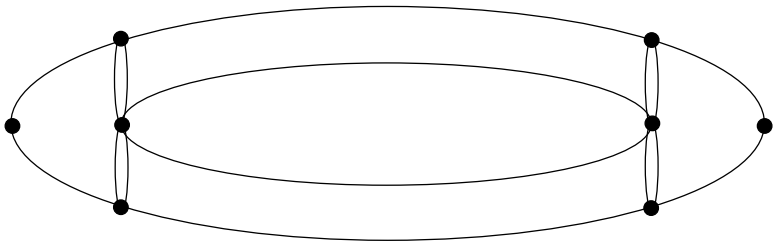


Figure 6.39: All of the silhouettes and critical points are merged to obtain the roadmap.

generate curves that connect them to the rest of the roadmap.

The completeness of the method requires very careful analysis, which is thoroughly covered in [77, 173]. The main elements of the analysis are showing that: 1) the polynomials can be perturbed and g can be chosen to ensure general position, 2) the singularity conditions on (6.28) lead to algebraic sets (varieties), and 3) the resulting roadmap has the required properties mentioned in Section 6.1 of being accessible and connectivity-preserving for $\mathcal{C}_{free}$ (actually it is shown for $\partial\mathcal{C}_{free}$). The method explained above computes the roadmap for each sign-invariant set, but to obtain a roadmap for the planning problem, the roadmaps from each sign-invariant set must be connected together correctly; fortunately, this has been solved via the Linking Lemma of [169]. A major problem, however, is that even after knowing the connectivity of the roadmap, it is a considerable challenge to obtain a parameterization of each curve on the roadmap. For this and many other technical reasons, no general implementation of Canny's algorithm appears to exist at present. Another problem is the requirement of a Whitney stratification (which can be fixed by perturbation of the input). The *Basu-Pollack-Roy roadmap algorithm* overcomes this problem [77].

6.5 Complexity of Motion Planning

This section summarizes theoretical work that characterizes the complexity of motion planning problems. Note that this is not equivalent to characterizing the running time of particular algorithms. The existence of an algorithm serves as an *upper bound* on the problem's difficulty because it is a proof by example that solving the problem requires no more time than what is needed by the algorithm. On the other hand, *lower bounds* are also very useful because they give an indication of the difficulty of the problem itself. Suppose, for example, you are given an algorithm that solves a problem in time $O(n^2)$. Does it make sense to try to find a more efficient algorithm? Does it make sense to try to find a general-purpose motion planning algorithm that runs in time that is polynomial in the dimension? Lower bounds provide answers to questions such as this. Usually lower bounds are obtained by concocting bizarre, complicated examples that are allowed by the problem definition but were usually not considered by the person who first formulated the problem. In this line of research, progress is made by either raising the lower bound (unless it is already tight) or by showing that a narrower version of the problem still allows such bizarre examples. The latter case occurs often in motion planning.

6.5.1 Lower Bounds

Lower bounds have been established for a variety of motion planning problems and also a wide variety of planning problems in general. To interpret these bounds a basic understanding of the *theory of computation* is required [462, 891]. This fascinating subject will be unjustly summarized in a few paragraphs. A *problem* is

a set of *instances* that are each carefully encoded as a binary string. An *algorithm* is formally considered as a *Turing machine*, which is a finite-state machine that can read and write bits to an unbounded piece of tape. Other models of computation also exist, such as integer RAM and real RAM (see [118]); there are debates as to which model is most appropriate, especially when performing geometric computations with real numbers. The standard Turing machine model will be assumed from here onward. Algorithms are usually formulated to make a binary output, which involves *accepting* or *rejecting* a problem instance that is initially written to the tape and given to the algorithm. In motion planning, this amounts to deciding whether a solution path exists for a given problem instance.

Languages A *language* is a set of binary strings associated with a problem. It represents the complete set of instances of a problem. An algorithm is said to *decide* a language if in finite time it correctly accepts all strings that belong to it and rejects all others. The interesting question is: How much time or space is required to decide a language? This question is asked of the problem, under the assumption that the best possible algorithm would be used to decide it. (We can easily think of inefficient algorithms that waste resources.)

A *complexity class* is a set of languages that can all be decided within some specified resource bound. The class P is the set of all languages (and hence problems) for which a polynomial-time algorithm exists (i.e., the algorithm runs in time $O(n^k)$ for some integer k). By definition, an algorithm is called *efficient* if it decides its associated language in polynomial time.[7] If no efficient algorithm exists, then the problem is called *intractable*. The relationship between several other classes that often emerge in theoretical motion planning is shown in Figure 6.40. The class NP is the set of languages that can be solved in polynomial time by a *nondeterministic Turing machine*. Some discussion of nondeterministic machines appears in Section 11.3.2. Intuitively, it means that solutions can be verified in polynomial time because the machine magically knows which choices to make while trying to make the decision. The class PSPACE is the set of languages that can be decided with no more than a polynomial amount of storage space during the execution of the algorithm (NPSPACE=PSPACE, so there is no nondeterministic version). The class EXPTIME is the set of languages that can be decided in time $O(2^{n^k})$ for some integer k. It is known that EXPTIME is larger than P, but it is not known precisely where the boundaries of NP and PSPACE lie. It might be the case that P = NP = PSPACE (although hardly anyone believes this), or it could be that NP = PSPACE = EXPTIME.

Hardness and completeness Since an easier class is included as a subset of a harder one, it is helpful to have a notion of a language (i.e., problem) being among the hardest possible within a class. Let X refer to either P, NP, PSPACE,

[7]Note that this definition may be absurd in practice; an algorithm that runs in time $O(n^{90125})$ would probably not be too efficient for most purposes.

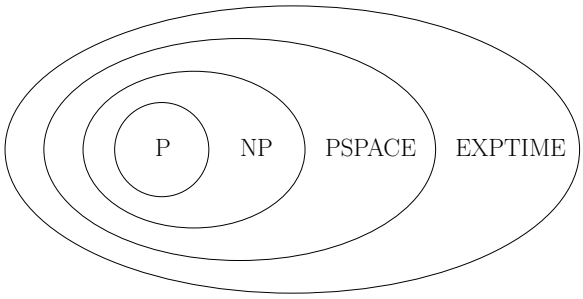


Figure 6.40: It is known that P $\subset$ EXPTIME is a strict subset; however, it is not known precisely how large NP and PSPACE are.

or EXPTIME. A language A is called *X-hard* if every language B in class X is *polynomial-time reducible* to A. In short, this means that in polynomial time, any language in B can be translated into instances for language A, and then the decisions for A can be correctly translated back in polynomial time to correctly decide B. Thus, if A can be decided, then within a polynomial-time factor, every language in X can be decided. The hardness concept can even be applied to a language (problem) that does not belong to the class. For example, we can declare that a language A is NP-hard even if $A \notin$NP (it could be harder and lie in EXPTIME, for example). If it is known that the language is both hard for some class X and is also a member of X, then it is called *X-complete* (i.e., NP-complete, PSPACE-complete, etc.).[8] Note that because of this uncertainty regarding P, NP, and PSPACE, one cannot say that a problem is intractable if it is NP-hard or PSPACE-hard, but one can, however, if the problem is EXPTIME-hard. One additional remark: it is useful to remember that PSPACE-hard implies NP-hard.

Lower bounds for motion planning The general motion planning problem, Formulation 4.1, was shown in 1979 to be PSPACE-hard by Reif [817]. In fact, the problem was restricted to polyhedral obstacles and a finite number of polyhedral robot bodies attached by spherical joints. The coordinates of all polyhedra are assumed to be in $\mathbb{Q}$ (this enables a finite-length string encoding of the problem instance). The proof introduces a fascinating motion planning instance that involves many attached, dangling robot parts that must work their way through a complicated system of tunnels, which together simulates the operation of a *symmetric Turing machine*. Canny later established that the problem in Formulation 4.1 (expressed using polynomials that have rational coefficients) lies in PSPACE [173]. Therefore, the general motion planning problem is PSPACE-complete.

Many other lower bounds have been shown for a variety of planning problems. One famous example is the Warehouseman's problem shown in Figure 6.41. This

[8]If you remember hearing that a planning problem is NP-something, but cannot remember whether it was NP-hard or NP-complete, then it is safe to say NP-hard because NP-complete implies NP-hard. This can similarly be said for other classes, such as PSPACE-complete vs. PSPACE-hard.

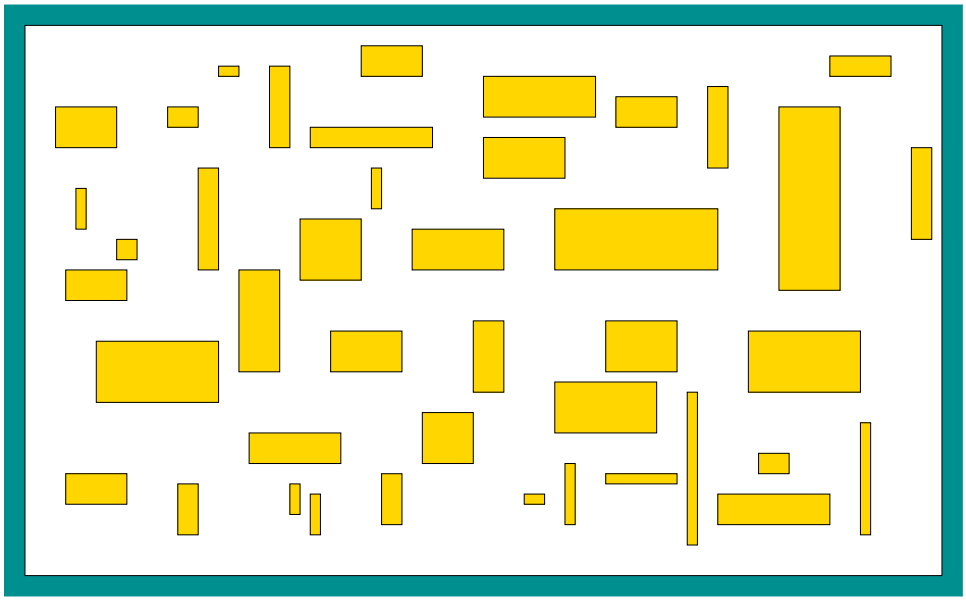


Figure 6.41: Even motion planning for a bunch of translating rectangles inside of a rectangular box in $\mathbb{R}^2$ is PSPACE-hard (and hence, NP-hard).

problem involves a finite number of translating, axis-aligned rectangles in a rectangular world. It was shown in [461] to be PSPACE-hard. This example is a beautiful illustration of how such a deceptively simple problem formulation can lead to such a high lower bound. More recently, it was even shown that planning for Sokoban, which is a warehouseman's problem on a discrete 2D grid, is also PSPACE-hard [255]. Other general motion planning problems that were shown to be PSPACE-hard include motion planning for a chain of bodies in the plane [460, 490] and motion planning for a chain of bodies among polyhedral obstacles in $\mathbb{R}^3$. Many lower bounds have been established for a variety of extensions and variations of the general motion planning problem. For example, in [172] it was established that a certain form of planning under uncertainty for a robot in a 3D polyhedral environment is NEXPTIME-hard, which is harder than any of the classes shown in Figure 6.40; the hardest problems in this NEXPTIME are believed to require doubly exponential time to solve.

The lower bound or hardness results depend significantly on the precise representation of the problem. For example, it is possible to make problems look easier by making instance encodings that are exponentially longer than they should be. The running time or space required is expressed in terms of n, the input size. If the motion planning problem instances are encoded with exponentially more bits than necessary, then a language that belongs to P is obtained. As long as the instance encoding is within a polynomial factor of the optimal encoding (this can be made precise using Kolmogorov complexity [630]), then this bizarre behavior is avoided. Another important part of the representation is to pay attention to how parameters in the problem formulation can vary. We can redefine motion

planning to be all instances for which the dimension of $\mathcal{C}$ is never greater than 2^{1000}. The number of dimensions is sufficiently large for virtually any application. The resulting language for this problem belongs to P because cylindrical algebraic decomposition and Canny's algorithm can solve any motion planning problem in polynomial time. Why? This is because now the dimension parameter in the time-complexity expressions can be replaced by 2^{1000}, which is a constant. This formally implies that an *efficient* algorithm is already known for any motion planning problem that we would ever care about. This implication has no practical value, however. Thus, be very careful when interpreting theoretical bounds.

The lower bounds may appear discouraging. There are two general directions to go from here. One is to weaken the requirements and tolerate algorithms that yield some kind of resolution completeness or probabilistic completeness. This approach was taken in Chapter 5 and leads to many efficient algorithms. Another direction is to define narrower problems that do not include the bizarre constructions that lead to bad lower bounds. For the narrower problems, it may be possible to design interesting, efficient algorithms. This approach was taken for the methods in Sections 6.2 and 6.3. In Section 6.5.3, upper bounds for some algorithms that address these narrower problems will be presented, along with bounds for the general motion planning algorithms. Several of the upper bounds involve Davenport-Schinzel sequences, which are therefore covered next.

6.5.2 Davenport-Schinzel Sequences

Davenport-Schinzel sequences provide a powerful characterization of the structure that arises from the lower or upper envelope of a collection of functions. The lower envelope of five functions is depicted in Figure 6.42. Such envelopes arise in many problems throughout computational geometry, including many motion planning problems. They are an important part of the design and analysis of many modern algorithms, and the resulting algorithm's time complexity usually involves terms that follow directly from the sequences. Therefore, it is worthwhile to understand some of the basics before interpreting some of the results of Section 6.5.3. Much more information on Davenport-Schinzel sequences and their applications appears in [866]. The brief introduction presented here is based on [865].

For positive integers n and s, an (n, s) *Davenport-Schinzel sequence* is a sequence $(u_1, \ldots, u_m)$ composed from a set of n *symbols* such that:

1. The same symbol may not appear consecutively in the sequence. In other words, $u_i \neq u_{i+1}$ for any i such that $1 \leq i < m$.

2. The sequence does not contain any alternating subsequence that uses two symbols and has length $s+2$. A subsequence can be formed by deleting any elements in the original sequence. The condition can be expressed as: There do not exist $s+2$ indices $i_1 < i_2 < \cdots < i_{s+2}$ for which $u_{i_1} = u_{i_3} = u_{i_5} = a$ and $u_{i_2} = u_{i_4} = u_{i_6} = b$, for some symbols a and b.

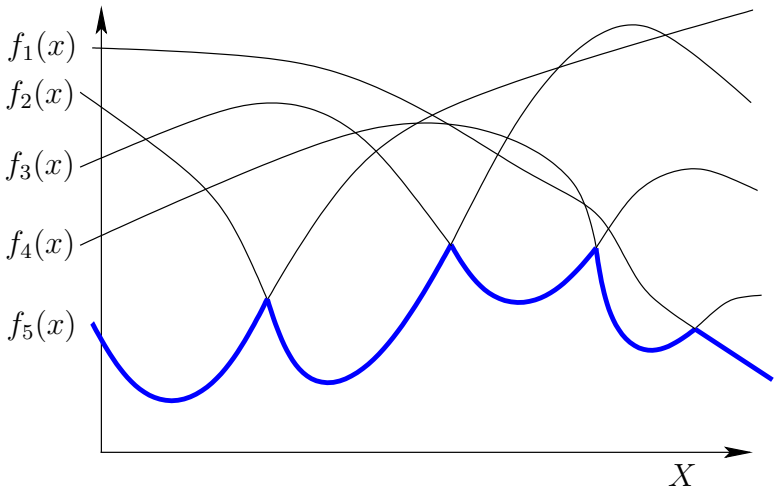


Figure 6.42: The lower envelope of a collection of functions.

As an example, an $(n, 3)$ sequence cannot appear as $(a \cdots b \cdots a \cdots b \cdots a)$, in which each $\cdots$ is filled in with any sequence of symbols. Let $\lambda_s(n)$ denote the maximum possible length of an (n, s) Davenport-Schinzel sequence.

The connection between Figure 6.42 and these sequences can now be explained. Consider the sequence of function indices that visit the lower envelope. In the example, this sequence is $(5, 2, 3, 4, 1)$. Suppose it is known that each pair of functions intersects in at most s places. If there are n real-valued continuous functions, then the sequence of function indices must be an (n, s) Davenport-Schinzel sequence. It is amazing that such sequences cannot be very long. For a fixed s, they are close to being linear.

The standard bounds for Davenport-Schinzel sequences are [865][9]

$$\lambda_1(n) = n \tag{6.32}$$

$$\lambda_2(n) = 2n - 1 \tag{6.33}$$

$$\lambda_3(n) = \Theta(n\alpha(n)) \tag{6.34}$$

$$\lambda_4(n) = \Theta(n \cdot 2^{\alpha(n)}) \tag{6.35}$$

$$\lambda_{2s}(n) \leq n \cdot 2^{\alpha(n)^{s-1} + C_{2s}(n)} \tag{6.36}$$

$$\lambda_{2s+1}(n) \leq n \cdot 2^{\alpha(n)^{s-1} \lg \alpha(n) + C'_{2s+1}(n)} \tag{6.37}$$

$$\lambda_{2s}(n) = \Omega(n \cdot 2^{\frac{1}{(s-1)!}\alpha(n)^{s-1} + C'_{2s}(n)}). \tag{6.38}$$

In the expressions above $C_r(n)$ and $C'_r(n)$ are terms that are smaller than their leading exponents. The $\alpha(n)$ term is the inverse Ackerman function, which is an extremely slow-growing function that appears frequently in algorithms. The *Ackerman function* is defined as follows. Let $A_1(m) = 2m$ and $A_{n+1}(m)$ represent m

[9]The following asymptotic notion is used: $O(f(n))$ denotes an upper bound, $\Omega(f(n))$ denotes a lower bound, and $\Theta(f(n))$ means that the bound is tight (both upper and lower). This notation is used in most books on algorithms [243].

applications of A_n. Thus, $A_1(m)$ performs doubling, $A_2(m)$ performs exponentiation, and $A_3(m)$ performs *tower exponentiation*, which makes a stack of 2's,

$$2^{2^{\cdot^{\cdot^{\cdot^{2^2}}}}}, \tag{6.39}$$

that has height m. The *Ackerman function* is defined as $A(n) = A_n(n)$. This function grows so fast that $A(4)$ is already an exponential tower of 2's that has height 65536. Thus, the *inverse Ackerman function*, α, grows very slowly. If n is less than or equal to an exponential tower of 65536 2's, then $\alpha(n) \leq 4$. Even when it appears in exponents of the Davenport-Schinzel bounds, it does not represent a significant growth rate.

Example 6.9 (Lower Envelope of Line Segments) One interesting application of Davenport-Schinzel applications is to the lower envelope of a set of line segments in $\mathbb{R}^2$. Since segments in general position may appear multiple times along the lower envelope, the total number of edges is $\Theta(\lambda_3(n)) = \Theta(n\alpha(n))$, which is higher than one would obtain from infinite lines. There are actually arrangements of segments in $\mathbb{R}^2$ that reach this bound; see [866]. ■

6.5.3 Upper Bounds

The upper bounds for motion planning problems arise from the existence of complete algorithms that solve them. This section proceeds by starting with the most general bounds, which are based on the methods of Section 6.4, and concludes with bounds for simpler motion planning problems.

General algorithms The first upper bound for the general motion planning problem of Formulation 4.1 came from the application of cylindrical algebraic decomposition [852]. Let n be the dimension of $\mathcal{C}$. Let m be the number of polynomials in $\mathcal{F}$, which are used to define $\mathcal{C}_{obs}$. Recall from Section 4.3.3 how quickly this grows for simple examples. Let d be the maximum degree among the polynomials in $\mathcal{F}$. The maximum degree of the resulting polynomials is bounded by $O(d^{2^{n-1}})$, and the total number of polynomials is bounded by $O((md)^{3^{n-1}})$. The total running time required to use cylindrical algebraic decomposition for motion planning is bounded by $(md)^{O(1)^n}$.[10] Note that the algorithm is doubly exponential in dimension n but polynomial in m and d. It can theoretically be declared to be *efficient* on a space of motion planning problems of bounded dimension (although, it certainly is not efficient for motion planning in any practical sense).

[10]It may seem odd for $O(\cdot)$ to appear in the middle of an expression. In this context, it means that there exists some $c \in [0, \infty)$ such that the running time is bounded by $(md)^{c^n}$. Note that another O is not necessary in front of the whole formula.

Since the general problem is PSPACE-complete, it appears unavoidable that a complete, general motion planning algorithm will require a running time that is exponential in dimension. Since cylindrical algebraic decomposition is doubly exponential, it led many in the 1980s to wonder whether this upper bound could be lowered. This was achieved by Canny's roadmap algorithm, for which the running time is bounded by $m^n(\lg m)d^{O(n^4)}$. Hence, it is singly exponential, which appears very close to optimal because it is up against the lower bound that seems to be implied by PSPACE-hardness (and the fact that problems exist that require a roadmap with $(md)^n$ connected components [77]). Much of the algorithm's complexity is due to finding a suitable deterministic perturbation to put the input polynomials into general position. A randomized algorithm can alternatively be used, for which the randomized expected running time is bounded by $m^n(\lg m)d^{O(n^2)}$. For a *randomized algorithm* [719], the *randomized expected* running time is still a worst-case upper bound, but averaged over random "coin tosses" that are introduced internally in the algorithm; it does *not* reflect any kind of average over the expected input distribution. Thus, these two bounds represent the best-known upper bounds for the general motion planning problem. Canny's algorithm may also be applied to solve the kinematic closure problems of Section 4.4, but the complexity does not reflect the fact that the dimension, k, of the algebraic variety is less than n, the dimension of $\mathcal{C}$. A roadmap algorithm that is particularly suited for this problem is introduced in [76], and its running time is bounded by $m^{k+1}d^{O(n^2)}$. This serves as the best-known upper bound for the problems of Section 4.4.

Specialized algorithms Now upper bounds are summarized for some narrower problems, which can be solved more efficiently than the general problem. All of the problems involve either two or three degrees of freedom. Therefore, we expect that the bounds are much lower than those for the general problem. In many cases, the Davenport-Schinzel sequences of Section 6.5.2 arise. Most of the bounds presented here are based on algorithms that are not practical to implement; they mainly serve to indicate the best asymptotic performance that can be obtained for a problem. Most of the bounds mentioned here are included in [865].

Consider the problem from Section 6.2, in which the robot translates in $\mathcal{W} = \mathbb{R}^2$ and $\mathcal{C}_{obs}$ is polygonal. Suppose that $\mathcal{A}$ is a convex polygon that has k edges and $\mathcal{O}$ is the union of m disjoint, convex polygons with disjoint interiors, and their total number of edges is n. In this case, the boundary of $\mathcal{C}_{free}$ (computed by Minkowski difference; see Section 4.3.2) has at most $6m-12$ nonreflex vertices (interior angles less than π) and $n + km$ reflex vertices (interior angles greater than π). The free space, $\mathcal{C}_{free}$, can be decomposed and searched in time $O((n+km)\lg^2 n)$ [518, 865]. Using randomized algorithms, the bound reduces to $O((n+km)\cdot 2^{\alpha(n)} \lg n)$ randomized expected time. Now suppose that $\mathcal{A}$ is a single nonconvex polygonal region described by k edges and that $\mathcal{O}$ is a similar polygonal region described by n edges. The Minkowski difference could yield as many as $\Omega(k^2n^2)$ edges for $\mathcal{C}_{obs}$. This can be avoided if the search is performed within a single connected component of $\mathcal{C}_{free}$.

Based on analysis that uses Davenport-Schinzel sequences, it can be shown that the worst connected component may have complexity $\Theta(kn\alpha(k))$, and the planning problem can be solved in time $O(kn \lg^2 n)$ deterministically or for a randomized algorithm, $O(kn \cdot 2^{\alpha(n)} \lg n)$ randomized expected time is needed. More generally, if $\mathcal{C}_{obs}$ consists of n algebraic curves in $\mathbb{R}^2$, each with degree no more than d, then the motion planning problem for translation only can be solved deterministically in time $O(\lambda_{s+2}(n) \lg^2 n)$, or with a randomized algorithm in $O(\lambda_{s+2}(n) \lg n)$ randomized expected time. In these expressions, $\lambda_{s+2}(n)$ is the bound (6.37) obtained from the $(n, s+2)$ Davenport-Schinzel sequence, and $s \leq d^2$.

For the case of the line-segment robot of Section 6.3.4 in an obstacle region described with n edges, an $O(n^5)$-time algorithm was given. This is not the best possible running time for solving the line-segment problem, but the method is easier to understand than others that are more efficient. In [748], a roadmap algorithm based on retraction is given that solves the problem in $O(n^2 \lg n \lg^* n)$ time, in which $\lg^* n$ is the number of times that lg has to be iterated on n to yield a result less than or equal to 1 (i.e., it is a very small, insignificant term; for practical purposes, you can imagine that the running time is $O(n^2 \lg n)$). The tightest known upper bound is $O(n^2 \lg n)$ [625]. It is established in [517] that there exist examples for which the solution path requires $\Omega(n^2)$ length to encode. For the case of a line segment moving in $\mathbb{R}^3$ among polyhedral obstacles with a total of n vertices, a complete algorithm that runs in time $O(n^4 + \epsilon)$ for any $\epsilon > 0$ was given in [547]. In [517] it was established that solution paths of complexity $\Omega(n^4)$ exist.

Now consider the case for which $\mathcal{C} = SE(2)$, $\mathcal{A}$ is a convex polygon with k edges, and $\mathcal{O}$ is a polygonal region described by n edges. The boundary of $\mathcal{C}_{free}$ has no more than $O(kn\lambda_6(kn))$ edges and can be computed to solve the motion planning problem in time $O(kn\lambda_6(kn) \lg kn)$ [10, 11]. An algorithm that runs in time $O(k^4 n \lambda_3(n) \lg n)$ and provides better clearance between the robot and obstacles is given in [205]. In [55] (some details also appear in [588]), an algorithm is presented, and even implemented, that solves the more general case in which $\mathcal{A}$ is nonconvex in time $O(k^3 n^3 \lg(kn))$. The number of faces of $\mathcal{C}_{obs}$ could be as high as $\Omega(k^3 n^3)$ for this problem. By explicitly representing and searching only one connected component, the best-known upper bound for the problem is $O((kn)^{2+\epsilon})$, in which $\epsilon > 0$ may be chosen arbitrarily small [428].

In the final case, suppose that $\mathcal{A}$ translates in $\mathcal{W} = \mathbb{R}^3$ to yield $\mathcal{C} = \mathbb{R}^3$. For a polyhedron or polyhedral region, let its *complexity* be the total number of faces, edges, and vertices. If $\mathcal{A}$ is a polyhedron with complexity k, and $\mathcal{O}$ is a polyhedral region with complexity n, then the boundary of $\mathcal{C}_{free}$ is a polyhedral surface of complexity $\Theta(k^3 n^3)$. As for other problems, if the search is restricted to a single component, then the complexity is reduced. The motion planning problem in this case can be solved in time $O((kn)^{2+\epsilon})$ [42]. If $\mathcal{A}$ is convex and there are m convex obstacles, then the best-known bound is $O(kmn \lg^2 m)$ time. More generally, if $\mathcal{C}_{obs}$ is bounded by n algebraic patches of constant maximum degree, then a vertical decomposition method solves the motion planning problem

within a single connected component of $\mathcal{C}_{free}$ in time $O(n^{2+\epsilon})$.

Further Reading

Most of the literature on combinatorial planning is considerably older than the sampling-based planning literature. A nice collection of early papers appears in [854]; this includes [460, 748, 749, 817, 851, 852, 853]. The classic motion planning textbook of Latombe [588] covers most of the methods presented in this chapter. The coverage here does not follow [588], which makes separate categories for cell decomposition methods and roadmap methods. A cell decomposition is constructed to produce a roadmap; hence, they are unified in this chapter. An excellent reference for material in combinatorial algorithms, computational geometry, and complete algorithms for motion planning is the collection of survey papers in [403].

Section 6.2 follows the spirit of basic algorithms from computational geometry. For a gentle introduction to computational geometry, including a nice explanation of vertical composition, see [264]. Other sources for computational geometry include [129, 302, 806]. To understand the difficulties in computing optimal decompositions of polygons, see [757]. See [650, 709, 832] for further reading on computing shortest paths.

Cell decompositions and cell complexes are very important in computational geometry and algebraic topology. Section 6.3 provided a brief perspective that was tailored to motion planning. For simplicial complexes in algebraic topology, see [496, 535, 834]; for singular complexes, see [834]. In computational geometry, various kinds of cell decompositions arise. Some of the most widely studied decompositions are *triangulations* [90] and *arrangements* [426], which are regions generated by a collection of primitives, such as lines or circles in the plane. For early cell decomposition methods in motion planning, see [854]. A survey of computational topology appears in [954].

The most modern and complete reference for the material in Section 6.4 is [77]. A gentle introduction to computational algebraic geometry is given in [250]. For details regarding algebraic computations with polynomials, see [704]. A survey of computational algebraic geometry appears in [705]. In addition to [77], other general references to cylindrical algebraic decomposition are [40, 232]. For its use in motion planning, see [588, 852]. The main reference for Canny's roadmap algorithm is [173]. Alternative high-level overviews to the one presented in Section 6.4.3 appear in [220, 588]. Variations and improvements to the algorithm are covered in [77]. A potential function-based extension of Canny's roadmap algorithm is developed in [176].

For further reading on the complexity of motion planning, consult the numerous references given in Section 6.5.

Exercises

1. Extend the vertical decomposition algorithm to correctly handle the case in which $\mathcal{C}_{obs}$ has two or more points that lie on the same vertical line. This includes the case of vertical segments. Random perturbations are not allowed.

2. Fully describe and prove the correctness of the bitangent computation method shown in Figure 6.14, which avoids trigonometric functions. Make certain that all types of bitangents (in general position) are considered.

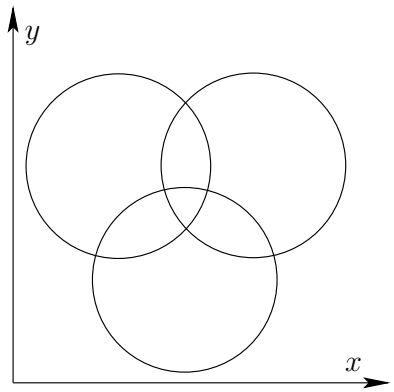


Figure 6.43: Determine the cylindrical algebraic decomposition obtained by projecting onto the x-axis.

3. Develop an algorithm that uses the plane-sweep principle to efficiently compute a representation of the union of two nonconvex polygons.

4. Extend the vertical cell decomposition algorithm of Section 6.2.2 to work for obstacle boundaries that are described as chains of circular arcs and line segments.

5. Extend the shortest-path roadmap algorithm of Section 6.2.4 to work for obstacle boundaries that are described as chains of circular arcs and line segments.

6. Derive the equation for the Conchoid of Nicomedes, shown in Figure 6.24, for the case of a line-segment robot contacting an obstacle vertex and edge simultaneously.

7. Propose a resolution-complete algorithm for motion planning of the line-segment robot in a polygonal obstacle region. The algorithm should compute exact C-space obstacle slices for any fixed orientation, θ; however, the algorithm should use van der Corput sampling over the set $[0, 2\pi)$ of orientations.

8. Determine the result of cylindrical algebraic decomposition for unit spheres $\mathbb{S}^1$, $\mathbb{S}^2$, $\mathbb{S}^3$, $\mathbb{S}^4$, ... Each $\mathbb{S}^n$ is expressed as a unit sphere in $\mathbb{R}^{n+1}$. Graphically depict the cases of $\mathbb{S}^1$ and $\mathbb{S}^2$. Also, attempt to develop an expression for the number of cells as a function of n.

9. Determine the cylindrical algebraic decomposition for the three intersecting circles shown in Figure 6.43. How many cells are obtained?

10. Using the matrix in (6.28), show that the result of Canny's roadmap for the torus, shown in Figure 6.39, is correct. Use the torus equation

$$(x_1^2 + x_2^2 + x_3^2 - (r_1^2 + r_2^2))^2 - 4r_1^2(r_2^2 - x_3^2) = 0, \tag{6.40}$$

in which r_1 is the major circle, r_2 is the minor circle, and $r_1 > r_2$.

11. Propose a vertical decomposition algorithm for a polygonal robot that can translate in the plane and even continuously vary its scale. How would the algorithm be modified to instead work for a robot that can translate or be sheared?

12. Develop a shortest-path roadmap algorithm for a flat torus, defined by identifying opposite edges of a square. Use Euclidean distance but respect the identifications when determining the shortest path. Assume the robot is a point and the obstacles are polygonal.

Implementations

13. Implement the vertical cell decomposition planning algorithm of Section 6.2.2.

14. Implement the maximum-clearance roadmap planning algorithm of Section 6.2.3.

15. Implement a planning algorithm for a point robot that moves in $\mathcal{W} = \mathbb{R}^3$ among polyhedral obstacles. Use vertical decomposition.

16. Implement an algorithm that performs a cylindrical decomposition of a polygonal obstacle region.

17. Implement an algorithm that computes the cell decomposition of Section 6.3.4 for the line-segment robot.

18. Experiment with cylindrical algebraic decomposition. The project can be greatly facilitated by utilizing existing packages for performing basic operations in computational algebraic geometry.

19. Implement the algorithm proposed in Exercise 7.