2 Line Segment Intersection

Thematic Map Ove

When you are visiting a country, maps are an invaluable source of information. They tell you where tourist attractions are located, they indicate the roads and railway lines to get there, they show small lakes, and so on. Unfortunately, they can also be a source of frustration, as it is often difficult to find the right information: even when you know the approximate position of a small town,

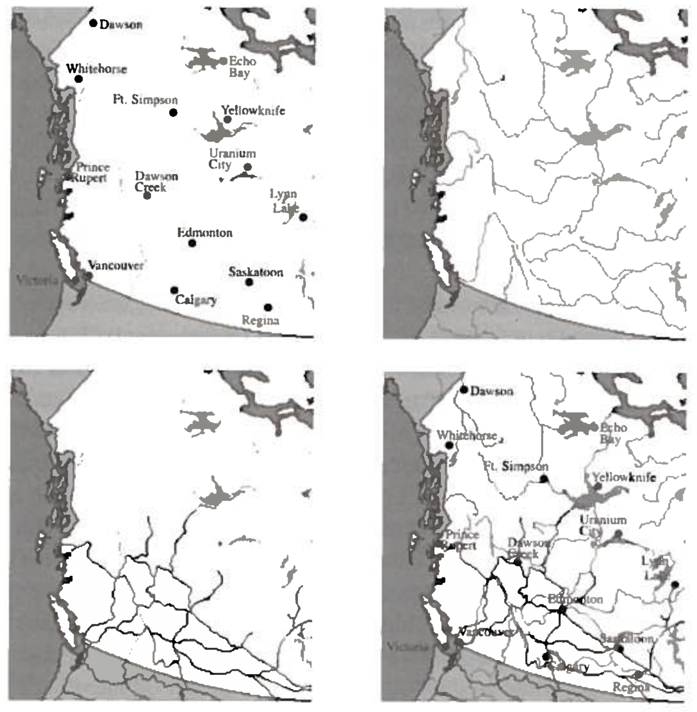


Figure 2.1
Cities, rivers, railroads, and the
overlay in western Canada

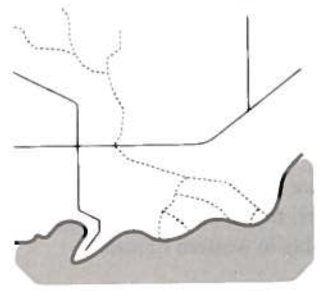
it can still be difficult to spot it on the map. To make maps more readable, geographic information systems split them into several *layers*. Each layer is a thematic map, that is, it stores only one type of information. Thus there will be a layer storing the roads, a layer storing the cities, a layer storing the rivers,

■ grizzly bear

and so on. The theme of a layer can also be more abstract. For instance, there could be a layer for the population density, for average precipitation, habitat of the grizzly bear, or for vegetation. The type of geometric information stored in a layer can be very different: the layer for a road map could store the roads as collections of line segments (or curves, perhaps), the layer for cities could contain points labeled with city names, and the layer for vegetation could store a subdivision of the map into regions labeled with the type of vegetation.

Users of a geographic information system can select one of the thematic maps for display. To find a small town you would select the layer storing cities, and you would not be distracted by information such as the names of rivers and lakes. After you have spotted the town, you probably want to know how to get there. To this end geographic information systems allow users to view an *overlay* of several maps—see Figure 2.1. Using an overlay of the road map and the map storing cities you can now figure out how to get to the town. When two or more thematic map layers are shown together, intersections in the overlay are positions of special interest. For example, when viewing the overlay of the layer for the roads and the layer for the rivers, it would be useful if the intersections were clearly marked. In this example the two maps are basically networks, and the intersections are points. In other cases one is interested in the intersection of complete regions. For instance, geographers studying the climate could be interested in finding regions where there is pine forest and the annual precipitation is between 1000 mm and 1500 mm. These regions are the intersections of the regions labeled "pine forest" in the vegetation map and the regions labeled "1000–1500" in the precipitation map.

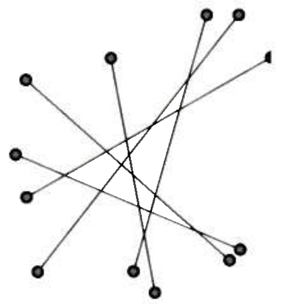
2.1 Line Segment Intersection



We first study the simplest form of the map overlay problem, where the two map layers are networks represented as collections of line segments. For example, a map layer storing roads, railroads, or rivers at a small scale. Note that curves can be approximated by a number of small segments. At first we won't be interested in the regions induced by these line segments. Later we shall look at the more complex situation where the maps are not just networks, but subdivisions of the plane into regions that have an explicit meaning. To solve the network overlay problem we first have to state it in a geometric setting. For the overlay of two networks the geometric situation is the following: given two sets of line segments, compute all intersections between a segment from one set and a segment from the other. This problem specification is not quite precise enough yet, as we didn't define when two segments intersect. In particular, do two segments intersect when an endpoint of one of them lies on the other? In other words, we have to specify whether the input segments are open or closed. To make this decision we should go back to the application, the network overlay problem. Roads in a road map and rivers in a river map are represented by chains of segments, so a crossing of a road and a river corresponds to the interior of one chain intersecting the interior of another chain. This does not mean that

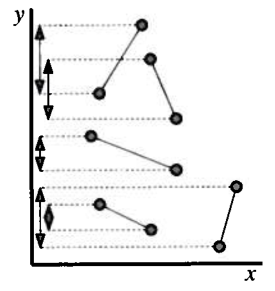
there is an intersection between the interior of two segments: the intersection point could happen to coincide with an endpoint of a segment of a chain. In fact, this situation is not uncommon because windy rivers are represented by many small segments and coordinates of endpoints may have been rounded when maps are digitized. We conclude that we should define the segments to be closed, so that an endpoint of one segment lying on another segment counts as an intersection.

To simplify the description somewhat we shall put the segments from the two sets into one set, and compute all intersections among the segments in that set. This way we certainly find all the intersections we want. We may also find intersections between segments from the same set. Actually, we certainly will, because in our application the segments from one set form a number of chains, and we count coinciding endpoints as intersections. These other intersections can be filtered out afterwards by simply checking for each reported intersection whether the two segments involved belong to the same set. So our problem specification is as follows: given a set S of n closed segments in the plane, report all intersection points among the segments in S.

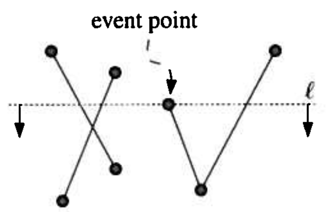


This doesn't seem like a challenging problem: we can simply take each pair of segments, compute whether they intersect, and, if so, report their intersection point. This brute-force algorithm clearly requires $O(n^2)$ time. In a sense this is optimal: when each pair of segments intersects any algorithm must take $\Omega(n^2)$ time, because it has to report all intersections. A similar example can be given when the overlay of two networks is considered. In practical situations, however, most segments intersect no or only a few other segments, so the total number of intersection points is much smaller than quadratic. It would be nice to have an algorithm that is faster in such situations. In other words, we want an algorithm whose running time depends not only on the number of segments in the input, but also on the number of intersection points. Such an algorithm is called an *output-sensitive algorithm*: the running time of the algorithm is sensitive to the size of the output. We could also call such an algorithm *intersection-sensitive*, since the number of intersections is what determines the size of the output.

How can we avoid testing all pairs of segments for intersection? Here we must make use of the geometry of the situation: segments that are close together are candidates for intersection, unlike segments that are far apart. Below we shall see how we can use this observation to obtain an output-sensitive algorithm for the line segment intersection problem.



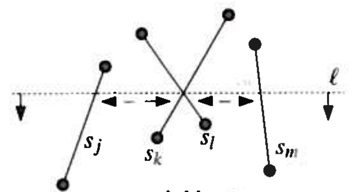
Let $S := \{s_1, s_2, \ldots, s_n\}$ be the set of segments for which we want to compute all intersections. We want to avoid testing pairs of segments that are far apart. But how can we do this? Let's first try to rule out an easy case. Define the y-interval of a segment to be its orthogonal projection onto the y-axis. When the y-intervals of a pair of segments do not overlap—we could say that they are far apart in the y-direction—then they cannot intersect. Hence, we only need to test pairs of segments whose y-intervals overlap, that is, pairs for which there exists a horizontal line that intersects both segments. To find these pairs we imagine sweeping a line ℓ downwards over the plane, starting from a position above all segments. While we sweep the imaginary line, we keep track of all segments intersecting it—the details of this will be explained later—so that we can find the pairs we need.



This type of algorithm is called a *plane sweep algorithm* and the line ℓ is called the *sweep line*. The *status* of the sweep line is the set of segments intersecting it. The status changes while the sweep line moves downwards, but not continuously. Only at particular points is an update of the status required. We call these points the *event points* of the plane sweep algorithm. In this algorithm the event points are the endpoints of the segments.

The moments at which the sweep line reaches an event point are the only moments when the algorithm actually does something: it updates the status of the sweep line and performs some intersection tests. In particular, if the event point is the upper endpoint of a segment, then a new segment starts intersecting the sweep line and must be added to the status. This segment is tested for intersection against the ones already intersecting the sweep line. If the event point is a lower endpoint, a segment stops intersecting the sweep line and must be deleted from the status. This way we only test pairs of segments for which there is a horizontal line that intersects both segments. Unfortunately, this is not enough: there are still situations where we test a quadratic number of pairs, whereas there is only a small number of intersection points. A simple example is a set of vertical segments that all intersect the x-axis. So the algorithm is not output-sensitive. The problem is that two segments that intersect the sweep line can still be far apart in the horizontal direction.

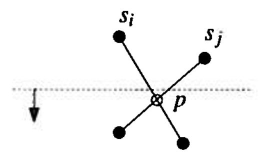
Let's order the segments from left to right as they intersect the sweep line, to include the idea of being close in the horizontal direction. We shall only test segments when they are adjacent in the horizontal ordering. This means that we only test any new segment against two segments, namely, the ones immediately left and right of the upper endpoint. Later, when the sweep line has moved downwards to another position, a segment can become adjacent to other segments against which it will be tested. Our new strategy should be reflected in the status of our algorithm: the status now corresponds to the *ordered* sequence of segments intersecting the sweep line. The new status not only changes at endpoints of segments; it also changes at intersection points, where the order of the intersected segments changes. When this happens we must test the two segments that change position against their new neighbors. This is a new type of event point.



Before trying to turn these ideas into an efficient algorithm, we should convince ourselves that the approach is correct. We have reduced the number of pairs to be tested, but do we still find all intersections? In other words, if two segments s_i and s_j intersect, is there always a position of the sweep line ℓ where s_i and s_j are adjacent along ℓ? Let's first ignore some nasty cases: assume that no segment is horizontal, that any two segments intersect in at most one point—they do not overlap—, and that no three segments meet in a common point. Later we shall see that these cases are easy to handle, but for now it is convenient to forget about them. The intersections where an endpoint of a segment lies on another segment can easily be detected when the sweep line

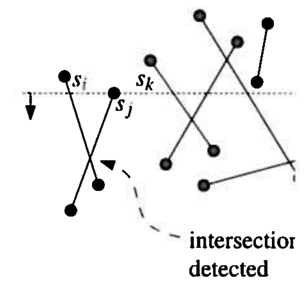
reaches the endpoint. So the only question is whether intersections between the interiors of segments are always detected.

Lemma 2.1 *Let s_i and s_j be two non-horizontal segments whose interiors intersect in a single point p, and assume there is no third segment passing through p. Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.*

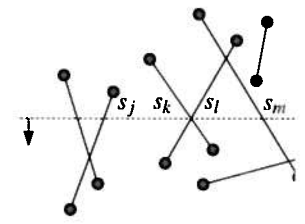


Proof. Let ℓ be a horizontal line slightly above p. If ℓ is close enough to p then s_i and s_j must be adjacent along ℓ. (To be precise, we should take ℓ such that there is no event point on ℓ, nor in between ℓ and the horizontal line through p.) In other words, there is a position of the sweep line where s_i and s_j are adjacent. On the other hand, s_i and s_j are not yet adjacent when the algorithm starts, because the sweep line starts above all line segments and the status is empty. Hence, there must be an event point q where s_i and s_j become adjacent and are tested for intersection. ⊟

So our approach is correct, at least when we forget about the nasty cases mentioned earlier. Now we can proceed with the development of the plane sweep algorithm. Let's briefly recap the overall approach. We imagine moving a horizontal sweep line ℓ downwards over the plane. The sweep line halts at certain event points; in our case these are the endpoints of the segments, which we know beforehand, and the intersection points, which are computed on the fly. While the sweep line moves we maintain the ordered sequence of segments intersected by it. When the sweep line halts at an event point the sequence of segments changes and, depending on the type of event point, we have to take several actions to update the status and detect intersections.

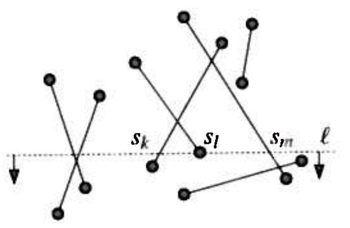


When the event point is the upper endpoint of a segment, there is a new segment intersecting the sweep line. This segment must be tested for intersection against its two neighbors along the sweep line. Only intersection points below the sweep line are important; the ones above the sweep line have been detected already. For example, if segments s_i and s_k are adjacent on the sweep line, and a new upper endpoint of a segment s_j appears in between, then we have to test s_j for intersection with s_i and s_k. If we find an intersection below the sweep line, we have found a new event point. After the upper endpoint is handled we continue to the next event point.



When the event point is an intersection, the two segments that intersect change their order. Each of them gets (at most) one new neighbor against which it is tested for intersection. Again, only intersections below the sweep line are still interesting. Suppose that four segments s_j, s_k, s_l, and s_m appear in this order on the sweep line when the intersection point of s_k and s_l is reached. Then s_k and s_l switch position and we must test s_l and s_j for intersection below the sweep line, and also s_k and s_m. The new intersections that we find are, of course, also event points for the algorithm. Note, however, that it is possible that these events have already been detected earlier, namely if a pair becoming adjacent has been adjacent before.

When the event point is the lower endpoint of a segment, its two neighbors now become adjacent and must be tested for intersection. If they intersect below the sweep line, then their intersection point is an event point. (Again, this event could have been detected already.) Assume three segments s_k, s_l, and s_m appear in this order on the sweep line when the lower endpoint of s_l is encountered. Then s_k and s_m will become adjacent and we test them for intersection.

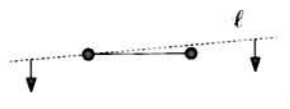


After we have swept the whole plane—more precisely, after we have treated the last event point—we have computed all intersection points. This is guaranteed by the following invariant, which holds at any time during the plane sweep: all intersection points above the sweep line have been computed correctly.

After this sketch of the algorithm, it's time to go into more detail. It's also time to look at the degenerate cases that can arise, like three or more segments meeting in a point. We should first specify what we expect from the algorithm in these cases. We could require the algorithm to simply report each intersection point once, but it seems more useful if it reports for each intersection point a list of segments that pass through it or have it as an endpoint. There is another special case for which we should define the required output more carefully, namely that of two partially overlapping segments, but for simplicity we shall ignore this case in the rest of this section.

We start by describing the data structures the algorithm uses.

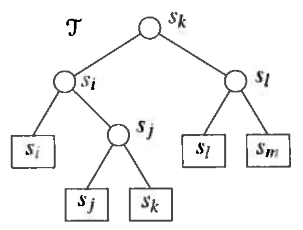
First of all we need a data structure—called the *event queue*—that stores the events. We denote the event queue by $\mathcal{Q}$. We need an operation that removes the next event that will occur from $\mathcal{Q}$, and returns it so that it can be treated. This event is the highest event below the sweep line. If two event points have the same y-coordinate, then the one with smaller x-coordinate will be returned. In other words, event points on the same horizontal line are treated from left to right. This implies that we should consider the left endpoint of a horizontal segment to be its upper endpoint, and its right endpoint to be its lower endpoint. You can also think about our convention as follows: instead of having a horizontal sweep line, imagine it is sloping just a tiny bit upward. As a result the sweep line reaches the left endpoint of a horizontal segment just before reaching the right endpoint. The event queue must allow insertions, because new events will be computed on the fly. Notice that two event points can coincide. For example, the upper endpoints of two distinct segments may coincide. It is convenient to treat this as one event point. Hence, an insertion must be able to check whether an event is already present in $\mathcal{Q}$.



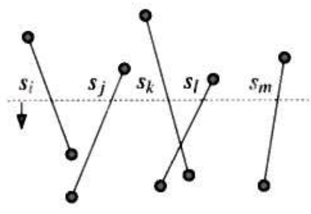
We implement the event queue as follows. Define an order $\prec$ on the event points that represents the order in which they will be handled. Hence, if p and q are two event points then we have $p \prec q$ if and only if $p_y > q_y$ holds or $p_y = q_y$ and $p_x < q_x$ holds. We store the event points in a balanced binary search tree, ordered according to $\prec$. With each event point p in $\mathcal{Q}$ we will store the segments starting at p, that is, the segments whose upper endpoint is p. This information will be needed to handle the event. Both operations—fetching the next event and inserting an event—take $O(\log m)$ time, where m is the number of events

in $\mathcal{Q}$. (We do not use a heap to implement the event queue, because we have to be able to test whether a given event is already present in $\mathcal{Q}$.)

Second, we need to maintain the status of the algorithm. This is the ordered sequence of segments intersecting the sweep line. The status structure, denoted by $\mathcal{T}$, is used to access the neighbors of a given segment s, so that they can be tested for intersection with s. The status structure must be dynamic: as segments start or stop to intersect the sweep line, they must be inserted into or deleted from the structure. Because there is a well-defined order on the segments in the status structure we can use a balanced binary search tree as status structure. When you are only used to binary search trees that store numbers, this may be surprising. But binary search trees can store any set of elements, as long as there is an order on the elements.



In more detail, we store the segments intersecting the sweep line ordered in the leaves of a balanced binary search tree $\mathcal{T}$. The left-to-right order of the segments along the sweep line corresponds to the left-to-right order of the leaves in $\mathcal{T}$. We must also store information in the internal nodes to guide the search down the tree to the leaves. At each internal node, we store the segment from the rightmost leaf in its left subtree. (Alternatively, we could store the segments only in interior nodes. This will save some storage. However, it is conceptually simpler to think about the segments in interior nodes as values to guide the search, not as data items. Storing the segments in the leaves also makes some algorithms simpler to describe.) Suppose we search in $\mathcal{T}$ for the segment immediately to the left of some point p that lies on the sweep line. At each internal node v we test whether p lies left or right of the segment stored at v. Depending on the outcome we descend to the left or right subtree of v, eventually ending up in a leaf. Either this leaf, or the leaf immediately to the left of it, stores the segment we are searching for. In a similar way we can find the segment immediately to the right of p, or the segments containing p. It follows that each update and neighbor search operation takes $O(\log n)$ time.



The event queue $\mathcal{Q}$ and the status structure $\mathcal{T}$ are the only two data structures we need. The global algorithm can now be described as follows.

Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The set of intersection points among the segments in S, with for each intersection point the segments that contain it.

```
1.  Initialize an empty event queue $\mathcal{Q}$. Next, insert the segment endpoints into
    $\mathcal{Q}$; when an upper endpoint is inserted, the corresponding segment should
    be stored with it.
2.  Initialize an empty status structure $\mathcal{T}$.
3.  while $\mathcal{Q}$ is not empty
4.     do Determine the next event point $p$ in $\mathcal{Q}$ and delete it.
5.        HANDLEEVENTPOINT($p$)
```

We have already seen how events are handled: at endpoints of segments we have to insert or delete segments from the status structure $\mathcal{T}$, and at intersection points we have to change the order of two segments. In both cases we also have to do intersection tests between segments that become neighbors after the event. In degenerate cases—where several segments are involved in one event point—the details are a little bit more tricky. The next procedure describes how to handle event points correctly; it is illustrated in Figure 2.2.

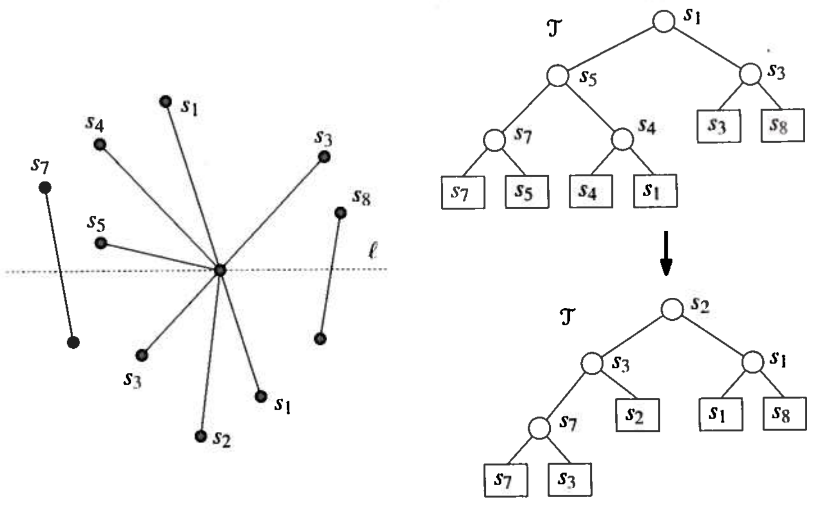


Figure 2.2
An event point and the changes in the status structure

```
HANDLEEVENTPOINT($p$)
1.  Let $U(p)$ be the set of segments whose upper endpoint is $p$; these segments
    are stored with the event point $p$. (For horizontal segments, the upper
    endpoint is by definition the left endpoint.)
2.  Find all segments stored in $\mathcal{T}$ that contain $p$; they are adjacent in $\mathcal{T}$. Let
    $L(p)$ denote the subset of segments found whose lower endpoint is $p$, and
    let $C(p)$ denote the subset of segments found that contain $p$ in their interior.
3.  if $L(p) \cup U(p) \cup C(p)$ contains more than one segment
4.    then Report $p$ as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.
5.  Delete the segments in $L(p) \cup C(p)$ from $\mathcal{T}$.
6.  Insert the segments in $U(p) \cup C(p)$ into $\mathcal{T}$. The order of the segments in $\mathcal{T}$
    should correspond to the order in which they are intersected by a sweep
    line just below $p$. If there is a horizontal segment, it comes last among all
    segments containing $p$.
7.  (* Deleting and re-inserting the segments of $C(p)$ reverses their order. *)
8.  if $U(p) \cup C(p) = \emptyset$
9.    then Let $s_l$ and $s_r$ be the left and right neighbors of $p$ in $\mathcal{T}$.
10.        FINDNEWEVENT($s_l, s_r, p$)
11.   else Let $s'$ be the leftmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
12.        Let $s_l$ be the left neighbor of $s'$ in $\mathcal{T}$.
13.        FINDNEWEVENT($s_l, s', p$)
14.        Let $s''$ be the rightmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
15.        Let $s_r$ be the right neighbor of $s''$ in $\mathcal{T}$.
16.        FINDNEWEVENT($s'', s_r, p$)
```

Note that in lines 8–16 we assume that s_l and s_r actually exist. If they do not exist the corresponding steps should obviously not be performed.

The procedures for finding the new intersections are easy: they simply test two segments for intersection. The only thing we need to be careful about is, when we find an intersection, whether this intersection has already been handled earlier or not. When there are no horizontal segments, then the intersection has not been handled yet when the intersection point lies below the sweep line. But how should we deal with horizontal segments? Recall our convention that events with the same y-coordinate are treated from left to right. This implies that we are still interested in intersection points lying to the right of the current event point. Hence, the procedure FINDNEWEVENT is defined as follows.

```
FINDNEWEVENT($s_l, s_r, p$)
1.  if $s_l$ and $s_r$ intersect below the sweep line, or on it and to the right of the
        current event point $p$, and the intersection is not yet present as an
        event in $\mathcal{Q}$
2.      then Insert the intersection point as an event into $\mathcal{Q}$.
```

What about the correctness of our algorithm? It is clear that FINDINTERSECTIONS only reports true intersection points, but does it find all of them? The next lemma states that this is indeed the case.

Lemma 2.2 *Algorithm* FINDINTERSECTIONS *computes all intersection points and the segments that contain it correctly.*

Proof. Recall that the priority of an event is given by its y-coordinate, and that when two events have the same y-coordinate the one with smaller x-coordinate is given higher priority. We shall prove the lemma by induction on the priority of the event points.

Let p be an intersection point and assume that all intersection points q with a higher priority have been computed correctly. We shall prove that p and the segments that contain p are computed correctly. Let $U(p)$ be the set of segments that have p as their upper endpoint (or, for horizontal segments, their left endpoint), let $L(p)$ be the set of segments having p as their lower endpoint (or, for horizontal segments, their right endpoint), and let $C(p)$ be the set of segments having p in their interior.

First, assume that p is an endpoint of one or more of the segments. In that case p is stored in the event queue $\mathcal{Q}$ at the start of the algorithm. The segments from $U(p)$ are stored with p, so they will be found. The segments from $L(p)$ and $C(p)$ are stored in $\mathcal{T}$ when p is handled, so they will be found in line 2 of HANDLEEVENTPOINT. Hence, p and all the segments involved are determined correctly when p is an endpoint of one or more of the segments.

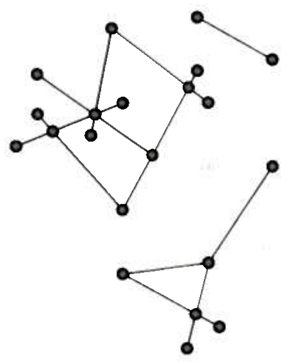
Now assume that p is not an endpoint of a segment. All we need to show is that p will be inserted into $\mathcal{Q}$ at some moment. Note that all segments that are involved have p in their interior. Order these segments by angle around p, and let s_i and s_j be two neighboring segments. Following the proof of Lemma 2.1 we see that there is an event point with a higher priority than p such that s_i and s_j become adjacent when q is passed. In Lemma 2.1 we assumed for simplicity that s_i and s_j are non-horizontal, but it is straightforward to adapt the proof for horizontal segments. By induction, the event point q was handled correctly, which means that p is detected and stored into $\mathcal{Q}$. $\square$

So we have a correct algorithm. But did we succeed in developing an output-sensitive algorithm? The answer is yes: the running time of the algorithm is $O((n+k)\log n)$, where k is the size of the output. The following lemma states an even stronger result: the running time is $O((n+I)\log n)$, where I is the number of intersections. This is stronger, because for one intersection point the output can consist of a large number of segments, namely in the case where many segments intersect in a common point.

Lemma 2.3 *The running time of Algorithm* FINDINTERSECTIONS *for a set S of n line segments in the plane is $O(n\log n + I\log n)$, where I is the number of intersection points of segments in S.*

Proof. The algorithm starts by constructing the event queue on the segment endpoints. Because we implemented the event queue as a balanced binary search tree, this takes $O(n\log n)$ time. Initializing the status structure takes constant time. Then the plane sweep starts and all the events are handled. To handle an event we perform three operations on the event queue $\mathcal{Q}$: the event itself is deleted from $\mathcal{Q}$ in line 4 of FINDINTERSECTIONS, and there can be one or two calls to FINDNEWEVENT, which may cause at most two new events to be inserted into $\mathcal{Q}$. Deletions and insertions on $\mathcal{Q}$ take $O(\log n)$ time each. We also perform operations—insertions, deletions, and neighbor finding—on the status structure $\mathcal{T}$, which take $O(\log n)$ time each. The number of operations is linear in the number $m(p) := \text{card}(L(p)\cup U(p)\cup C(p))$ of segments that are involved in the event. If we denote the sum of all $m(p)$, over all event points p, by m, the running time of the algorithm is $O(m\log n)$.

It is clear that $m = O(n+k)$, where k is the size of the output; after all, whenever $m(p) > 1$ we report all segments involved in the event, and the only events involving one segment are the endpoints of segments. But we want to prove that $m = O(n+I)$, where I is the number of intersection points. To show this, we will interpret the set of segments as a planar graph embedded in the plane. (If you are not familiar with planar graph terminology, you should read the first paragraphs of Section 2.2 first.) Its vertices are the endpoints of segments and intersection points of segments, and its edges are the pieces of the segments connecting vertices. Consider an event point p. It is a vertex of the graph, and $m(p)$ is bounded by the degree of the vertex. Consequently, m is bounded by the sum of the degrees of all vertices of our graph. Every edge of the graph contributes one to the degree of exactly two vertices (its endpoints), so m is bounded by $2n_e$, where n_e is the number of edges of the graph. Let's bound n_e in terms of n and I. By definition, n_v, the number of vertices, is at most $2n+I$. It is well known that in planar graphs $n_e = O(n_v)$, which proves our claim. But, for completeness, let us give the argument here. Every face of the planar graph is bounded by at least three edges—provided that there are at least three segments—and an edge can bound at most two different faces. Therefore n_f, the number of faces, is at most $2n_e/3$. We now use *Euler's formula*, which states that for any planar graph with n_v vertices, n_e edges, and n_f faces, the following relation holds:



$$n_v - n_e + n_f \geqslant 2.$$

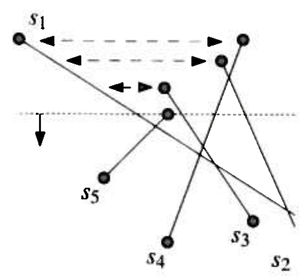
Equality holds if and only if the graph is connected. Plugging the bounds on n_v and n_f into this formula, we get

$$2 \leqslant (2n+I) - n_e + \frac{2n_e}{3} = (2n+I) - n_e/3.$$

So $n_e \leqslant 6n+3I-6$, and $m \leqslant 12n+6I-12$, and the bound on the running time follows. $\square$

We still have to analyze the other complexity aspect, the amount of storage used by the algorithm. The tree $\mathcal{T}$ stores a segment at most once, so it uses $O(n)$ storage. The size of $\mathcal{Q}$ can be larger, however. The algorithm inserts intersection points in $\mathcal{Q}$ when they are detected and it removes them when they are handled. When it takes a long time before intersections are handled, it could happen that $\mathcal{Q}$ gets very large. Of course its size is always bounded by $O(n+I)$, but it would be better if the working storage were always linear.

There is a relatively simple way to achieve this: only store intersection points of pairs of segments that are currently adjacent on the sweep line. The algorithm given above also stores intersection points of segments that have been horizontally adjacent, but aren't anymore. By storing only intersections among adjacent segments, the number of event points in $\mathcal{Q}$ is never more than linear. The modification required in the algorithm is that the intersection point of two segments must be deleted when they stop being adjacent. These segments must become adjacent again before the intersection point is reached, so the intersection point will still be reported correctly. The total time taken by the algorithm remains $O(n\log n + I\log n)$. We obtain the following theorem:



Theorem 2.4 *Let S be a set of n line segments in the plane. All intersection points in S, with for each intersection point the segments involved in it, can be reported in $O(n\log n + I\log n)$ time and $O(n)$ space, where I is the number of intersection points.*

2.2 The Doubly-Connected Edge List

We have solved the easiest case of the map overlay problem, where the two maps are networks represented as collections of line segments. In general, maps have a more complicated structure: they are subdivisions of the plane into labeled regions. A thematic map of forests in Canada, for instance, would be a subdivision of Canada into regions with labels such as "pine", "deciduous", "birch", and "mixed".

Before we can give an algorithm for computing the overlay of two subdivisions, we must develop a suitable representation for a subdivision. Storing a subdivision as a collection of line segments is not such a good idea. Operations like reporting the boundary of a region would be rather complicated. It is better

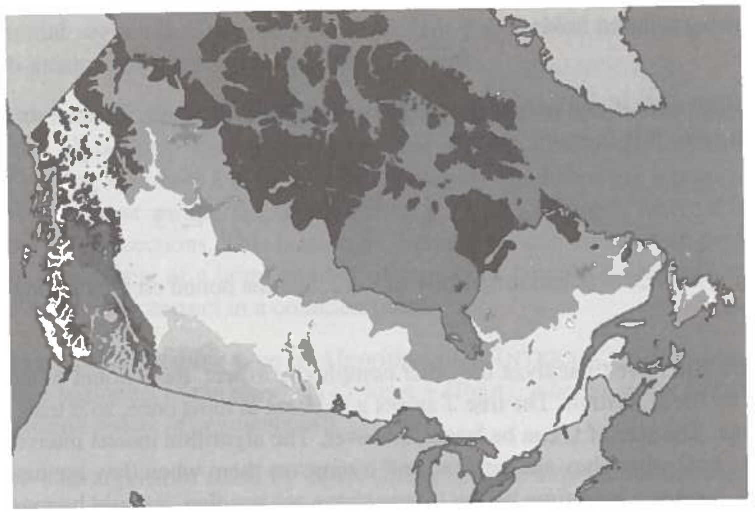
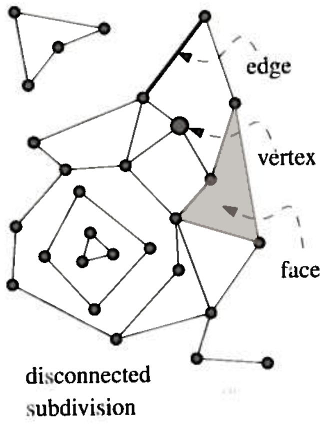


Figure 2.3
Types of forest in Canada

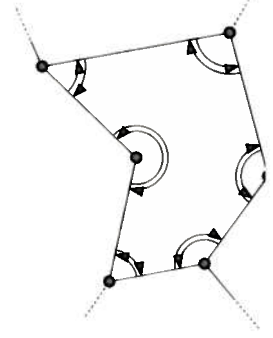
to incorporate structural, topological information: which segments bound a given region, which regions are adjacent, and so on.

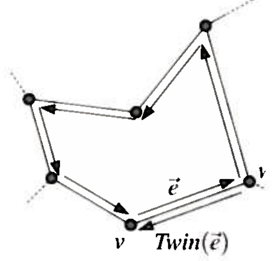


The maps we consider are *planar subdivisions* induced by planar embeddings of graphs. Such a subdivision is *connected* if the underlying graph is connected. The embedding of a node of the graph is called a *vertex*, and the embedding of an arc is called an *edge*. We only consider embeddings where every edge is a straight line segment. In principle, edges in a subdivision need not be straight. A subdivision need not even be a planar embedding of a graph, as it may have unbounded edges. In this section, however, we don't consider such more general subdivisions. We consider an edge to be open, that is, its endpoints—which are vertices of the subdivision—are not part of it. A *face* of the subdivision is a maximal connected subset of the plane that doesn't contain a point on an edge or a vertex. Thus a face is an open polygonal region whose boundary is formed by edges and vertices from the subdivision. The *complexity* of a subdivision is defined as the sum of the number of vertices, the number of edges, and the number of faces it consists of. If a vertex is the endpoint of an edge, then we say that the vertex and the edge are *incident*. Similarly, a face and an edge on its boundary are incident, and a face and a vertex of its boundary are incident.

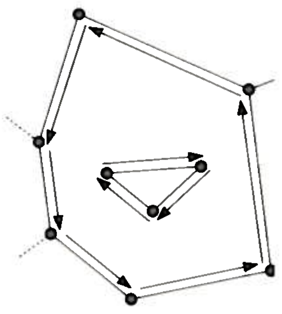
What should we require from a representation of a subdivision? An operation one could ask for is to determine the face containing a given point. This is definitely useful in some applications—indeed, in a later chapter we shall design a data structure for this—but it is a bit too much to ask from a basic representation. The things we can ask for should be more local. For example, it is reasonable to require that we can walk around the boundary of a given face, or that we can access one face from an adjacent one if we are given a common edge. Another operation that could be useful is to visit all the edges around a given vertex. The representation that we shall discuss supports these operations. It is called the doubly-connected edge list.

A *doubly-connected edge list* contains a record for each face, edge, and vertex of the subdivision. Besides the geometric and topological information—to be described shortly—each record may also store additional information. For instance, if the subdivision represents a thematic map for vegetation, the doubly-connected edge list would store in each face record the type of vegetation of the corresponding region. The additional information is also called *attribute information*. The geometric and topological information stored in the doubly-connected edge list should enable us to perform the basic operations mentioned earlier. To be able to walk around a face in counterclockwise order we store a pointer from each edge to the next. It can also come in handy to walk around a face the other way, so we also store a pointer to the previous edge. An edge usually bounds two faces, so we need two pairs of pointers for it. It is convenient to view the different sides of an edge as two distinct *half-edges*, so that we have a unique next half-edge and previous half-edge for every half-edge. This also means that a half-edge bounds only one face. The two half-edges we get for a given edge are called *twins*. Defining the next half-edge of a given half-edge with respect to a counterclockwise traversal of a face induces an orientation on each half-edge: it is oriented such that the face that it bounds lies to its left for an observer walking along the edge. Because half-edges are oriented we can speak of the *origin* and the *destination* of a half-edge. If a half-edge $\vec{e}$ has v as its origin and w as its destination, then its twin $Twin(\vec{e})$ has w as its origin and v as its destination. To reach the boundary of a face we just need to store one pointer in the face record to an arbitrary half-edge bounding the face. Starting from that half-edge, we can step from each half-edge to the next and walk around the face.



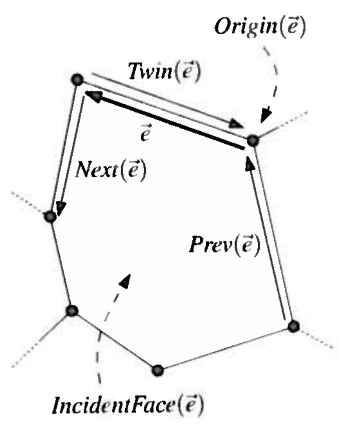


What we just said does not quite hold for the boundaries of holes in a face: if they are traversed in counterclockwise order then the face lies to the right. It will be convenient to orient half-edges such that their face always lies to the same side, so we change the direction of traversal for the boundary of a hole to clockwise. Now a face always lies to the left of any half-edge on its boundary. Another consequence is that twin half-edges always have opposite orientations. The presence of holes in a face also means that one pointer from the face to an arbitrary half-edge on its boundary is not enough to visit the whole boundary: we need a pointer to a half-edge in every boundary component. If a face has isolated vertices that don't have any incident edge, we can store pointers to them as well. For simplicity we'll ignore this case.



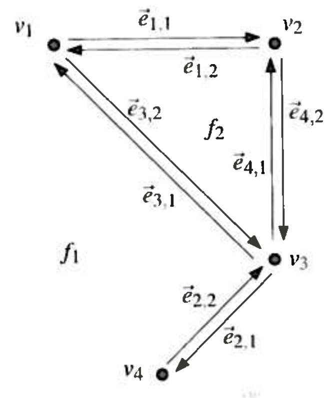
Let's summarize. The doubly-connected edge list consists of three collections of records: one for the vertices, one for the faces, and one for the half-edges. These records store the following geometric and topological information:

- The vertex record of a vertex v stores the coordinates of v in a field called *Coordinates*(v). It also stores a pointer *IncidentEdge*(v) to an arbitrary half-edge that has v as its origin.
- The face record of a face f stores a pointer *OuterComponent*(f) to some half-edge on its outer boundary. For the unbounded face this pointer is **nil**. It also stores a list *InnerComponents*(f), which contains for each hole in the face a pointer to some half-edge on the boundary of the hole.



- The half-edge record of a half-edge $\vec{e}$ stores a pointer $Origin(\vec{e})$ to its origin, a pointer $Twin(\vec{e})$ to its twin half-edge, and a pointer $IncidentFace(\vec{e})$ to the face that it bounds. We don't need to store the destination of an edge, because it is equal to $Origin(Twin(\vec{e}))$. The origin is chosen such that $IncidentFace(\vec{e})$ lies to the left of $\vec{e}$ when it is traversed from origin to destination. The half-edge record also stores pointers $Next(\vec{e})$ and $Prev(\vec{e})$ to the next and previous edge on the boundary of $IncidentFace(\vec{e})$. Thus $Next(\vec{e})$ is the unique half-edge on the boundary of $IncidentFace(\vec{e})$ that has the destination of $\vec{e}$ as its origin, and $Prev(\vec{e})$ is the unique half-edge on the boundary of $IncidentFace(\vec{e})$ that has $Origin(\vec{e})$ as its destination.

A constant amount of information is used for each vertex and edge. A face may require more storage, since the list $InnerComponents(f)$ has as many elements as there are holes in the face. Because any half-edge is pointed to at most once from all $InnerComponents(f)$ lists together, we conclude that the amount of storage is linear in the complexity of the subdivision. An example of a doubly-connected edge list for a simple subdivision is given below. The two half-edges corresponding to an edge e_i are labeled $\vec{e}_{i,1}$ and $\vec{e}_{i,2}$.



Vertex	Coordinates	IncidentEdge
v_1	(0,4)	$\vec{e}_{1,1}$
v_2	(2,4)	$\vec{e}_{4,2}$
v_3	(2,2)	$\vec{e}_{2,1}$
v_4	(1,1)	$\vec{e}_{2,2}$

Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

The information stored in the doubly-connected edge list is enough to perform the basic operations. For example, we can walk around the outer boundary of a given face f by following $Next(\vec{e})$ pointers, starting from the half-edge $OuterComponent(f)$. We can also visit all edges incident to a vertex v. It is a good exercise to figure out for yourself how to do this.

We described a fairly general version of the doubly-connected edge list. In applications where the vertices carry no attribute information we could store

their coordinates directly in the $Origin()$ field of the edge; there is no strict need for a separate type of vertex record. Even more important is to realize that in many applications the faces of the subdivision carry no interesting meaning (think of the network of rivers or roads that we looked at before). If that is the case, we can completely forget about the face records, and the $IncidentFace()$ field of half-edges. As we will see, the algorithm of the next section doesn't need these fields (and is actually simpler to implement if we don't need to update them). Some implementations of doubly-connected edge lists may also insist that the graph formed by the vertices and edges of the subdivision be connected. This can always be achieved by introducing dummy edges, and has two advantages. Firstly, a simple graph transversal can be used to visit all half-edges, and secondly, the $InnerComponents()$ list for faces is not necessary.

2.3 Computing the Overlay of Two Subdivisions

Now that we have designed a good representation of a subdivision, we can tackle the general map overlay problem. We define the overlay of two subdivisions $\mathcal{S}_1$ and $\mathcal{S}_2$ to be the subdivision $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ such that there is a face f in $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ if and only if there are faces f_1 in $\mathcal{S}_1$ and f_2 in $\mathcal{S}_2$ such that f is a maximal connected subset of $f_1 \cap f_2$. This sounds more complicated than it is: what it means is that the overlay is the subdivision of the plane induced by the edges from $\mathcal{S}_1$ and $\mathcal{S}_2$. Figure 2.4 illustrates this. The general map overlay problem

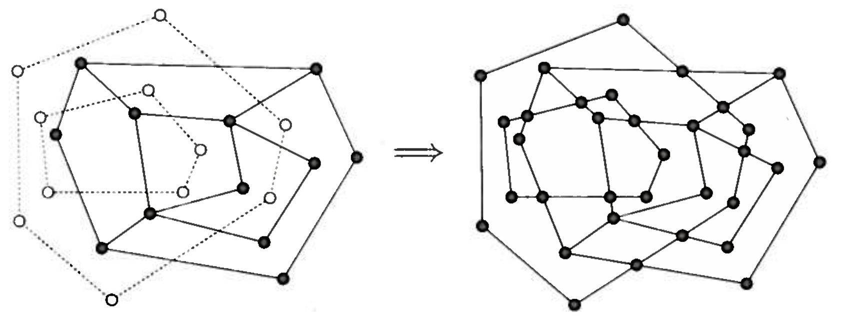


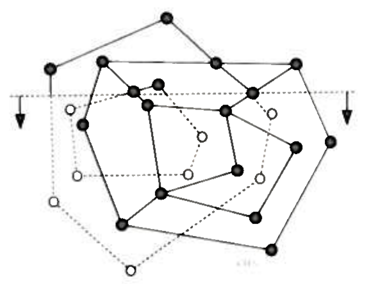
Figure 2.4
Overlaying two subdivisions

is to compute a doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$, given the doubly-connected edge lists of $\mathcal{S}_1$ and $\mathcal{S}_2$. We require that each face in $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ be labeled with the labels of the faces in $\mathcal{S}_1$ and $\mathcal{S}_2$ that contain it. This way we have access to the attribute information stored for these faces. In an overlay of a vegetation map and a precipitation map this would mean that we know for each region in the overlay the type of vegetation and the amount of precipitation.

Let's first see how much information from the doubly-connected edge lists for $\mathcal{S}_1$ and $\mathcal{S}_2$ we can re-use in the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$. Consider the network of edges and vertices of $\mathcal{S}_1$. This network is cut into pieces by the edges of $\mathcal{S}_2$. These pieces are for a large part re-usable; only the edges that have been cut by the edges of $\mathcal{S}_2$ should be renewed. But does this also hold for the half-edge records in the doubly-connected edge list that correspond to the pieces? If the orientation of a half-edge would change, we would still have to change the information in these records. Fortunately, this is not the case. The half-edges are oriented such that the face that they bound lies to the left; the shape of the face may change in the overlay, but it will remain to the same side of the half-edge. Hence, we can re-use half-edge records corresponding to edges that are not intersected by edges from the other map. Stated differently, the only half-edge records in the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ that we cannot borrow from $\mathcal{S}_1$ or $\mathcal{S}_2$ are the ones that are incident to an intersection between edges from different maps.

This suggests the following approach. First, copy the doubly-connected edge lists of $\mathcal{S}_1$ and $\mathcal{S}_2$ into one new doubly-connected edge list. The new doubly-connected edge list is not a valid doubly-connected edge list, of course, in the sense that it does not yet represent a planar subdivision. This is the task of the overlay algorithm: it must transform the doubly-connected edge list into a valid doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ by computing the intersections between the two networks of edges, and linking together the appropriate parts of the two doubly-connected edge lists.

We did not talk about the new face records yet. The information for these records is more difficult to compute, so we leave this for later. We first describe in a little more detail how the vertex and half-edge records of the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ are computed.



Our algorithm is based on the plane sweep algorithm of Section 2.1 for computing the intersections in a set of line segments. We run this algorithm on the set of segments that is the union of the sets of edges of the two subdivisions $\mathcal{S}_1$ and $\mathcal{S}_2$. Here we consider the edges to be closed. Recall that the algorithm is supported by two data structures: an event queue $\mathcal{Q}$, which stores the event points, and the status structure $\mathcal{T}$, which is a balanced binary search tree storing the segments intersecting the sweep line, ordered from left to right. We now also maintain a doubly-connected edge list $\mathcal{D}$. Initially, $\mathcal{D}$ contains a copy of the doubly-connected edge list for $\mathcal{S}_1$ and a copy of the doubly-connected edge list for $\mathcal{S}_2$. During the plane sweep we shall transform $\mathcal{D}$ to a correct doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$. That is to say, as far as the vertex and half-edge records are concerned; the face information will be computed later. We keep cross pointers between the edges in the status structure $\mathcal{T}$ and the half-edge records in $\mathcal{D}$ that correspond to them. This way we can access the part of $\mathcal{D}$ that needs to be changed when we encounter an intersection point. The invariant that we maintain is that at any time during the sweep, the part of the overlay above the sweep line has been computed correctly.

Now, let's consider what we must do when we reach an event point. First of all, we update $\mathcal{T}$ and $\mathcal{Q}$ as in the line segment intersection algorithm. If the event involves only edges from one of the two subdivisions, this is all; the event point is a vertex that can be re-used. If the event involves edges from both subdivisions, we must make local changes to $\mathcal{D}$ to link the doubly-connected edge lists of the two original subdivisions at the intersection point. This is tedious but not difficult.

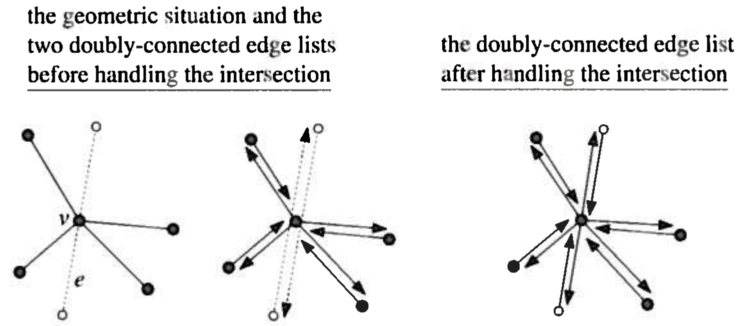
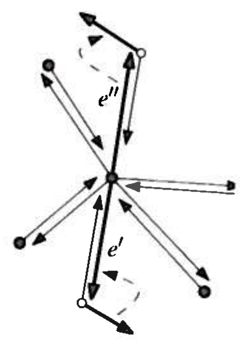
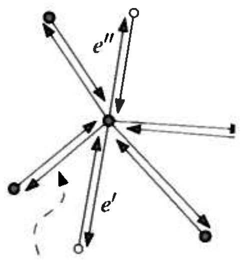

the geometric situation and the two doubly-connected edge lists before handling the intersection
the doubly-connected edge list after handling the intersection
v
e

Figure 2.5
An edge of one subdivision pa
through a vertex of the other

We describe the details for one of the possible cases, namely when an edge e of $\mathcal{S}_1$ passes through a vertex v of $\mathcal{S}_2$, see Figure 2.5. The edge e must be replaced by two edges denoted e' and e''. In the doubly-connected edge list, the two half-edges for e must become four. We create two new half-edge records, both with v as the origin. The two existing half-edges for e keep the endpoints of e as their origin, as shown in Figure 2.5. Then we pair up the existing half-edges with the new half-edges by setting their *Twin*() pointers. So e' is represented by one new and one existing half-edge, and the same holds for e''. Now we must set a number of *Prev*() and *Next*() pointers. We first deal with the situation around the endpoints of e; later we'll worry about the situation around v. The *Next*() pointers of the two new half-edges each copy the *Next*() pointer of the old half-edge that is not its twin. The half-edges to which these pointers point must also update their *Prev*() pointer and set it to the new half-edges. The correctness of this step can be verified best by looking at a figure.



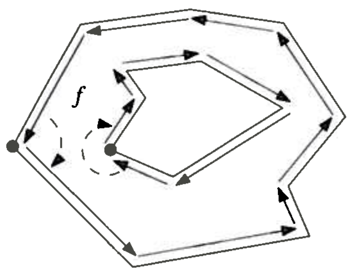
It remains to correct the situation around vertex v. We must set the *Next*() and *Prev*() pointers of the four half-edges representing e' and e'', and of the four half-edges incident from $\mathcal{S}_2$ to v. We locate these four half-edges from $\mathcal{S}_2$ by testing where e' and e'' should be in the cyclic order of the edges around vertex v. There are four pairs of half-edges that become linked by a *Next*() pointer from the one and a *Prev*() pointer from the other. Consider the half-edge for e' that has v as its destination. It must be linked to the first half-edge, seen clockwise from e', with v as its origin. The half-edge for e' with v as its origin must be linked to the first counterclockwise half-edge with v as its destination. The same statements hold for e''.



Most of the steps in the description above take only constant time. Only locating where e' and e'' appear in the cyclic order around v may take longer: it will take time linear in the degree of v. The other cases that can arise—crossings of two edges from different maps, and coinciding vertices—are not more difficult than the case we just discussed. These cases also take time $O(m)$, where m is the number of edges incident to the event point. This means that updating $\mathcal{D}$ does not increase the running time of the line segment intersection algorithm asymptotically. Notice that every intersection that we find is a vertex of the overlay. It follows that the vertex records and the half-edge records of the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ can be computed in $O(n\log n + k\log n)$ time, where n denotes the sum of the complexities of $\mathcal{S}_1$ and $\mathcal{S}_2$, and k is the complexity of the overlay.

After the fields involving vertex and half-edge records have been set, it remains to compute the information about the faces of $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$. More precisely, we have to create a face record for each face f in $\mathcal{O}(\mathcal{S}_1, \mathcal{S}_2)$, we have to make *OuterComponent*(f) point to a half-edge on the outer boundary of f, and we have to make a list *InnerComponents*(f) of pointers to half-edges on the boundaries of the holes inside f. Furthermore, we must set the *IncidentFace*() fields of the half-edges on the boundary of f so that they point to the face record of f. Finally, each of the new faces must be labeled with the names of the faces in the old subdivisions that contain it.

How many face records will there be? Well, except for the unbounded face, every face has a unique outer boundary, so the number of face records we have to create is equal to the number of outer boundaries plus one. From the part of the doubly-connected edge list we have constructed so far we can easily extract all boundary cycles. But how do we know whether a cycle is an outer boundary or the boundary of a hole in a face? This can be decided by looking at the leftmost vertex v of the cycle, or, in case of ties, at the lowest of the leftmost ones. Recall that half-edges are directed in such a way that their incident face locally lies to the left. Consider the two half-edges of the cycle that are incident to v. Because we know that the incident face lies to the left, we can compute the angle these two half-edges make inside the incident face. If this angle is smaller than 180° then the cycle is an outer boundary, and otherwise it is the boundary of a hole. This property holds for the leftmost vertex of a cycle, but not necessarily for other vertices of that cycle.



To decide which boundary cycles bound the same face we construct a graph $\mathcal{G}$. For every boundary cycle—inner and outer—there is a node in $\mathcal{G}$. There is also one node for the imaginary outer boundary of the unbounded face. There is an arc between two cycles if and only if one of the cycles is the boundary of a hole and the other cycle has a half-edge immediately to the left of the leftmost vertex of that hole cycle. If there is no half-edge to the left of the leftmost vertex of a cycle, then the node representing the cycle is linked to the node of the unbounded face. Figure 2.6 gives an example. The dotted segments in the figure indicate the linking of the hole cycles to other cycles. The graph corresponding to the subdivision is also shown in the figure. The hole cycles are shown as single circles, and the outer boundary cycles are shown as double circles. Observe that $\mathcal{C}_3$ and $\mathcal{C}_6$ are in the same connected component as $\mathcal{C}_2$. This indicates that $\mathcal{C}_3$ and $\mathcal{C}_6$ are hole cycles in the face whose outer boundary is $\mathcal{C}_2$. If there is only one hole in a face f, then the graph $\mathcal{G}$ links the boundary cycle of the hole to the outer boundary of f. In general this need not be the case: a hole can also be linked to another hole, as you can see in Figure 2.6. This hole, which lies in the same face f, may be linked to the outer boundary of f, or it may be linked to yet another hole. But eventually we must end up linking a hole to the outer boundary, as the next lemma shows.

Lemma 2.5 *Each connected component of the graph $\mathcal{G}$ corresponds exactly to the set of cycles incident to one face.*

Proof. Consider a cycle $\mathcal{C}$ bounding a hole in a face f. Because f lies locally to the left of the leftmost vertex of $\mathcal{C}$, $\mathcal{C}$ must be linked to another cycle that also

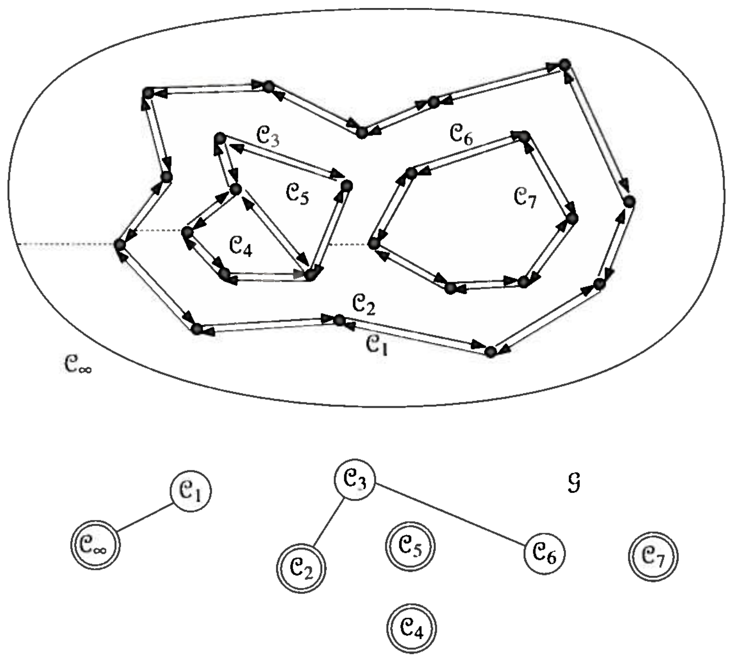
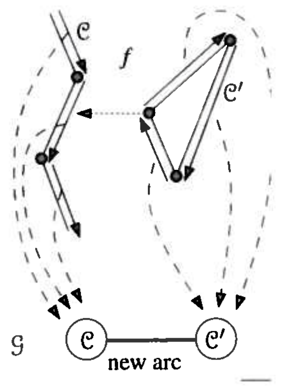


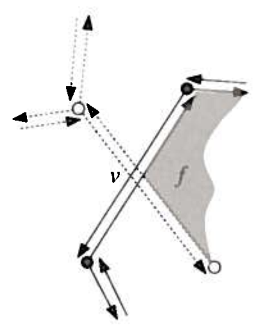
Figure 2.6
A subdivision and the correspond
graph $\mathcal{G}$

bounds f. It follows that cycles in the same connected component of $\mathcal{G}$ bound the same face.

To finish the proof, we show that every cycle bounding a hole in f is in the same connected component as the outer boundary of f. Suppose there is a cycle for which this is not the case. Let $\mathcal{C}$ be the leftmost such cycle, that is, the one whose the leftmost vertex is leftmost. By definition there is an arc between the $\mathcal{C}$ and another cycle $\mathcal{C}'$ that lies partly to the left of the leftmost vertex of $\mathcal{C}$. Hence, $\mathcal{C}'$ is in the same connected component as $\mathcal{C}$, which is not the component of the outer boundary of f. This contradicts the definition of $\mathcal{C}$. $\square$

Lemma 2.5 shows that once we have the graph $\mathcal{G}$, we can create a face record for every component. Then we can set the *IncidentFace*() pointers of the half-edges that bound each face f, and we can construct the list *InnerComponents*(f) and the set *OuterComponent*(f). How can we construct $\mathcal{G}$? Recall that in the plane sweep algorithm for line segment intersection we always looked for the segments immediately to the left of an event point. (They had to be tested for intersection against the leftmost edge through the event point.) Hence, the information we need to construct $\mathcal{G}$ is determined during the plane sweep. So, to construct $\mathcal{G}$, we first make a node for every cycle. To find the arcs of $\mathcal{G}$, we consider the leftmost vertex v of every cycle bounding a hole. If $\vec{e}$ is the half-edge immediately left of v, then we add an arc between the two nodes in $\mathcal{G}$ representing the cycle containing $\vec{e}$ and the hole cycle of which v is the leftmost vertex. To find these nodes in $\mathcal{G}$ efficiently we need pointers from every half-edge record to the node in $\mathcal{G}$ representing the cycle it is in. So the face information of the doubly-connected edge list can be set in $O(n+k)$ additional time, after the plane sweep.





One thing remains: each face f in the overlay must be labeled with the names of the faces in the old subdivisions that contained it. To find these faces, consider an arbitrary vertex v of f. If v is the intersection of an edge e_1 from $\mathcal{S}_1$ and an edge e_2 from $\mathcal{S}_2$ then we can decide which faces of $\mathcal{S}_1$ and $\mathcal{S}_2$ contain f by looking at the *IncidentFace*() pointer of the appropriate half-edges corresponding to e_1 and e_2. If v is not an intersection but a vertex of, say, $\mathcal{S}_1$, then we only know the face of $\mathcal{S}_1$ containing f. To find the face of $\mathcal{S}_2$ containing f, we have to do some more work: we have to determine the face of $\mathcal{S}_2$ that contains v. In other words, if we knew for each vertex of $\mathcal{S}_1$ in which face of $\mathcal{S}_2$ it lay, and vice versa, then we could label the faces of $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ correctly. How can we compute this information? The solution is to apply the paradigm that has been introduced in this chapter, plane sweep, once more. However, we won't explain this final step here. It is a good exercise to test your understanding of the plane sweep approach to design the algorithm yourself. (In fact, it is not necessary to compute this information in a separate plane sweep. It can also be done in the sweep that computes the intersections.)

Putting everything together we get the following algorithm.

Algorithm MAPOVERLAY($\mathcal{S}_1,\mathcal{S}_2$)
Input. Two planar subdivisions $\mathcal{S}_1$ and $\mathcal{S}_2$ stored in doubly-connected edge lists.
Output. The overlay of $\mathcal{S}_1$ and $\mathcal{S}_2$ stored in a doubly-connected edge list $\mathcal{D}$.

```
1.  Copy the doubly-connected edge lists for $\mathcal{S}_1$ and $\mathcal{S}_2$ to a new doubly-
    connected edge list $\mathcal{D}$.
2.  Compute all intersections between edges from $\mathcal{S}_1$ and $\mathcal{S}_2$ with the plane
    sweep algorithm of Section 2.1. In addition to the actions on $\mathcal{T}$ and $\mathcal{Q}$
    required at the event points, do the following:
      ■ Update $\mathcal{D}$ as explained above if the event involves edges of both $\mathcal{S}_1$
        and $\mathcal{S}_2$. (This was explained for the case where an edge of $\mathcal{S}_1$ passes
        through a vertex of $\mathcal{S}_2$.)
      ■ Store the half-edge immediately to the left of the event point at the
        vertex in $\mathcal{D}$ representing it.
3.  (* Now $\mathcal{D}$ is the doubly-connected edge list for $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$, except that the
    information about the faces has not been computed yet. *)
4.  Determine the boundary cycles in $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ by traversing $\mathcal{D}$.
5.  Construct the graph $\mathcal{G}$ whose nodes correspond to boundary cycles and
    whose arcs connect each hole cycle to the cycle to the left of its leftmost ver-
    tex, and compute its connected components. (The information to determine
    the arcs of $\mathcal{G}$ has been computed in line 2, second item.)
6.  for each connected component in $\mathcal{G}$
7.      do Let $\mathcal{C}$ be the unique outer boundary cycle in the component and let
           $f$ denote the face bounded by the cycle. Create a face record for $f$,
           set OuterComponent($f$) to some half-edge of $\mathcal{C}$, and construct the
           list InnerComponents($f$) consisting of pointers to one half-edge in
           each hole cycle in the component. Let the IncidentFace() pointers
           of all half-edges in the cycles point to the face record of $f$.
8.  Label each face of $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$ with the names of the faces of $\mathcal{S}_1$ and $\mathcal{S}_2$
    containing it, as explained above.
```

Theorem 2.6 *Let $\mathcal{S}_1$ be a planar subdivision of complexity n_1, let $\mathcal{S}_2$ be a subdivision of complexity n_2, and let $n := n_1 + n_2$. The overlay of $\mathcal{S}_1$ and $\mathcal{S}_2$ can be constructed in $O(n\log n + k\log n)$ time, where k is the complexity of the overlay.*

Proof. Copying the doubly-connected edge lists in line 1 takes $O(n)$ time, and the plane sweep of line 2 takes $O(n\log n + k\log n)$ time by Lemma 2.3. Steps 4–7, where we fill in the face records, takes time linear in the complexity of $\mathcal{O}(\mathcal{S}_1,\mathcal{S}_2)$. (The connected components of a graph can be determined in linear time by a simple depth first search.) Finally, labeling each face in the resulting subdivision with the faces of the original subdivisions that contain it can be done in $O(n\log n + k\log n)$ time. $\square$

2.4 Boolean Operations

The map overlay algorithm is a powerful instrument that can be used for various other applications. One particular useful one is performing the Boolean operations union, intersection, and difference on two polygons $\mathcal{P}_1$ and $\mathcal{P}_2$. See Figure 2.7 for an example. Note that the output of the operations might no longer be a polygon. It can consist of a number of polygonal regions, some with holes.

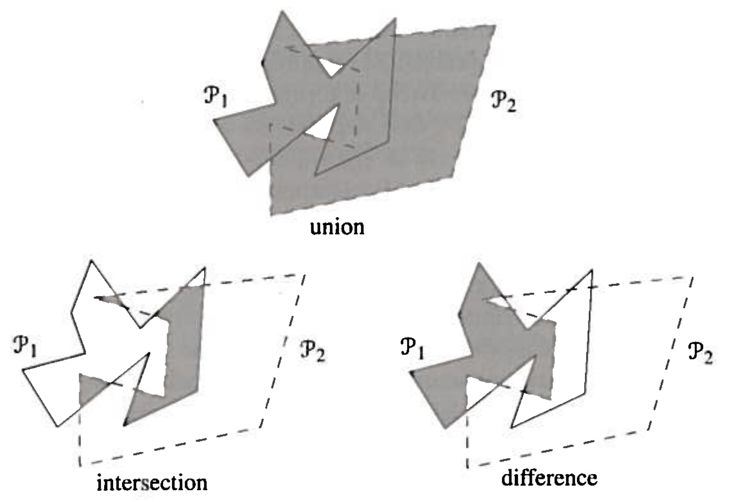


Figure 2.7
The Boolean operations union, intersection and difference on tv
polygons $\mathcal{P}_1$ and $\mathcal{P}_2$

To perform the Boolean operation we regard the polygons as planar maps whose bounded faces are labeled $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. We compute the overlay of these maps, and we extract the faces in the overlay whose labels correspond to the particular Boolean operation we want to perform. If we want to compute the intersection $\mathcal{P}_1 \cap \mathcal{P}_2$, we extract the faces in the overlay that are labeled with $\mathcal{P}_1$ and $\mathcal{P}_2$. If we want to compute the union $\mathcal{P}_1 \cup \mathcal{P}_2$, we extract the faces in the overlay that are labeled with $\mathcal{P}_1$ or $\mathcal{P}_2$. And if we want to compute the difference $\mathcal{P}_1 \setminus \mathcal{P}_2$, we extract the faces in the overlay that are labeled with $\mathcal{P}_1$ and not with $\mathcal{P}_2$.

Because every intersection point of an edge of $\mathcal{P}_1$ and an edge of $\mathcal{P}_2$ is a vertex of $\mathcal{P}_1 \cap \mathcal{P}_2$, the running time of the algorithm is $O(n \log n + k \log n)$, where n is the total number of vertices in $\mathcal{P}_1$ and $\mathcal{P}_2$, and k is the complexity of $\mathcal{P}_1 \cap \mathcal{P}_2$. The same holds for the other Boolean operations: every intersection of two edges is a vertex of the final result, no matter which operation we want to perform. We immediately get the following result.

Corollary 2.7 *Let $\mathcal{P}_1$ be a polygon with n_1 vertices and $\mathcal{P}_2$ a polygon with n_2 vertices, and let $n := n_1 + n_2$. Then $\mathcal{P}_1 \cap \mathcal{P}_2$, $\mathcal{P}_1 \cup \mathcal{P}_2$, and $\mathcal{P}_1 \setminus \mathcal{P}_2$ can each be computed in $O(n \log n + k \log n)$ time, where k is the complexity of the output.*

2.5 Notes and Comments

The line segment intersection problem is one of the most fundamental problems in computational geometry. The $O(n \log n + k \log n)$ solution presented in this chapter was given by Bentley and Ottmann [47] in 1979. (A few years earlier, Shamos and Hoey [351] had solved the *detection* problem, where one is only interested in deciding whether there is at least one intersection, in $O(n \log n)$ time.) The method for reducing the working storage from $O(n+k)$ to $O(n)$ described in this chapter is taken from Pach and Sharir [312], who also show that the event list can have size $\Omega(n \log n)$ before this improvement. Brown [77] describes an alternative method to achieve the reduction.

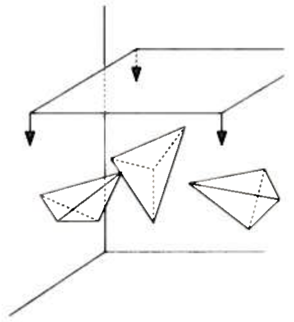
The lower bound for the problem of reporting all line segment intersections is $\Omega(n \log n + k)$, so the plane sweep algorithm described in this chapter is not optimal when k is large. A first step towards an optimal algorithm was taken by Chazelle [88], who gave an algorithm with $O(n \log^2 n / \log \log n + k)$ running time. In 1988 Chazelle and Edelsbrunner [99, 100] presented the first $O(n \log n + k)$ time algorithm. Unfortunately, it requires $O(n+k)$ storage. Later Clarkson and Shor [133] and Mulmuley [288] gave randomized incremental algorithms whose expected running time is also $O(n \log n + k)$. (See Chapter 4 for an explanation of randomized algorithms.) The working storage of these algorithms is $O(n)$ and $O(n+k)$, respectively. Unlike the algorithm of Chazelle and Edelsbrunner, these randomized algorithms also work for computing intersections in a set of curves. Balaban [35] gave the first deterministic algorithm for the segment intersection problem that works in $O(n \log n + k)$ time and $O(n)$ space. It also works for curves.

There are cases of the line segment intersection problem that are easier than the general case. One such case is where we have two sets of segments, say red segments and blue segments, such that no two segments from the same set intersect each other. (This is, in fact, exactly the network overlay problem. In the solution described in this chapter, however, the fact that the segments came from two sets of non-intersecting segments was not used.) This so-called red-blue line segment intersection problem was solved in $O(n \log n + k)$ time

and $O(n)$ storage by Mairson and Stolfi [262] before the general problem was solved optimally. Other optimal red-blue intersection algorithms were given by Chazelle et al. [101] and by Palazzi and Snoeyink [315]. If the two sets of segments form connected subdivisions then the situation is even better: in this case the overlay can be computed in $O(n+k)$ time, as has been shown by Finke and Hinrichs [176]. Their result generalizes and improves previous results on map overlay by Nievergelt and Preparata [293], Guibas and Seidel [200], and Mairson and Stolfi [262].

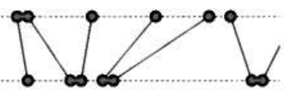
The line segment intersection counting problem is to determine the number of intersection points in a set of n line segments. Since the output is a single integer, a term with k in the time bound no longer refers to the output size (which is constant), but only to the number of intersections. Algorithms that do not depend on the number of intersections take $O(n^{4/3} \log^c n)$ time, for some small constant c [4, 95]; a running time close to $O(n \log n)$ is not known to exist.

Plane sweep is one of the most important paradigms for designing geometric algorithms. The first algorithms in computational geometry based on this paradigm are by Shamos and Hoey [351], Lee and Preparata [250], and Bentley and Ottmann [47]. Plane sweep algorithms are especially suited for finding intersections in sets of objects, but they can also be used for solving many other problems. In Chapter 3 plane sweep solves part of the polygon triangulation problem, and in Chapter 7 we will see a plane sweep algorithm to compute the so-called Voronoi diagram of a set of points. The algorithm presented in the current chapter sweeps a horizontal line downwards over the plane. For some problems it is more convenient to sweep the plane in another way. For instance, we can sweep the plane with a rotating line—see Chapter 15 for an example—or with a pseudo-line (a line that need not be straight, but otherwise behaves more or less as a line) [159]. The plane sweep technique can also be used in higher dimensions: here we sweep the space with a hyperplane [213, 311, 324]. Such algorithms are called space sweep algorithms.



In this chapter we described a data structure for storing subdivisions: the doubly-connected edge list. This structure, or in fact a variant of it, was described by Muller and Preparata [286]. There are also other data structures for storing subdivisions, such as the winged edge structure by Baumgart [40] and the quad edge structure by Guibas and Stolfi [202]. The difference between all these structures is small. They all have more or less the same functionality, but some save a few bytes of storage per edge.

2.6 Exercises



2.1 Let S be a set of n disjoint line segments whose upper endpoints lie on the line $y = 1$ and whose lower endpoints lie on the line $y = 0$. These segments partition the horizontal strip $[-\infty : \infty] \times [0 : 1]$ into $n+1$ regions. Give an $O(n \log n)$ time algorithm to build a binary search tree on the segments