following relation holds:

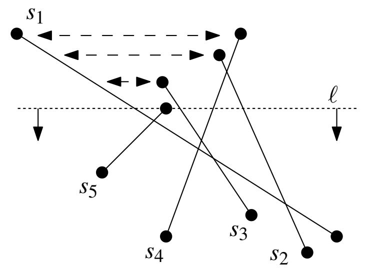
$$n_v - n_e + n_f \geqslant 2.$$

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

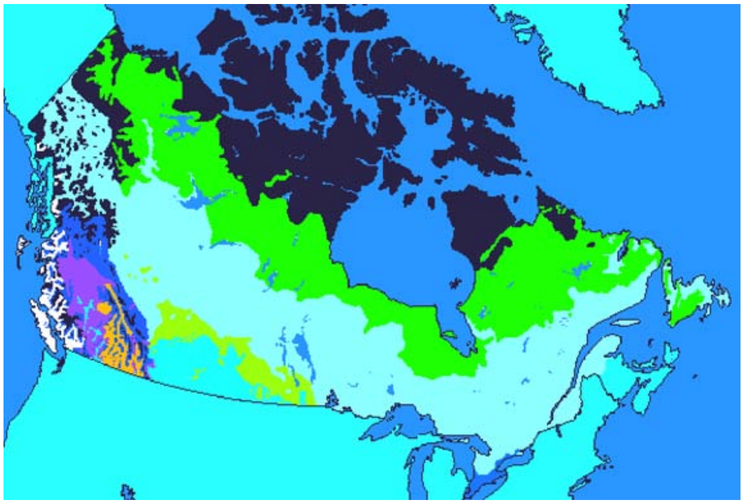
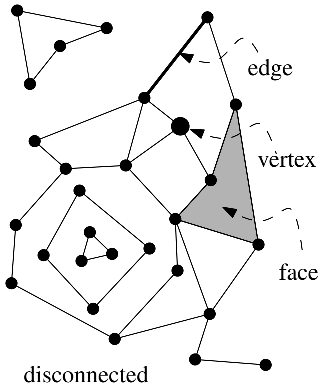


Figure 2.3
Types of forest in Canada

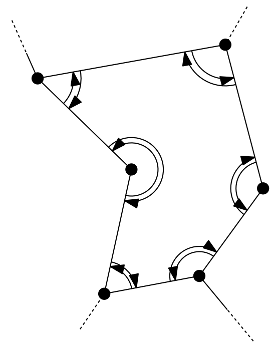
to incorporate structural, topological information: which segments bound a given region, which regions are adjacent, and so on.

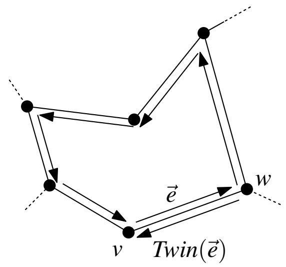


The maps we consider are *planar subdivisions* induced by planar embeddings of graphs. Such a subdivision is *connected* if the underlying graph is connected. The embedding of a node of the graph is called a *vertex*, and the embedding of an arc is called an *edge*. We only consider embeddings where 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.

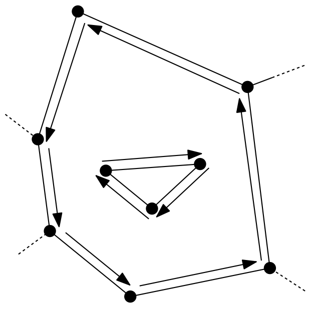
What should we require from a representation of a subdivision? An operation one could ask for is to determine the face containing a given point. This is definitely useful in some applications—indeed, in a later chapter we shall design a data structure for this—but it is a bit too much to ask from a basic 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.





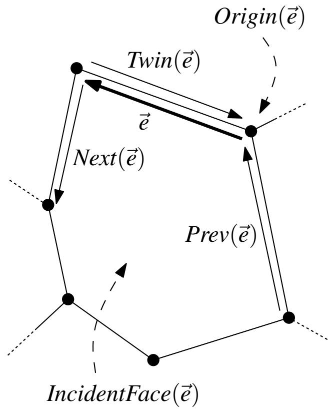
What we just said does not quite hold for the boundaries of holes in a face: if they are traversed in counterclockwise order then the face lies to the right. It will be convenient to orient half-edges such that their face always lies to the same side, so we change the direction of traversal for the boundary 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.



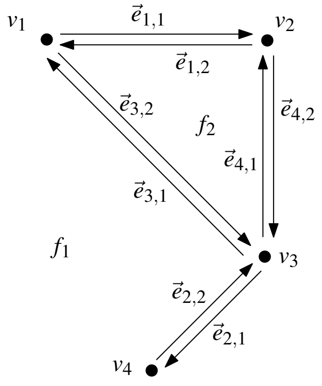
Let's summarize. The doubly-connected edge list consists of three collections of records: one for the vertices, one for the faces, and one for the half-edges. These records store the following geometric and topological information:

- 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

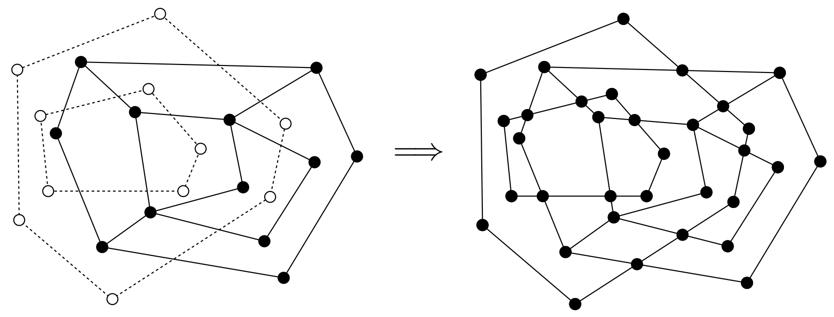


Figure 2.4
Overlaying two subdivisions

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