2 Line Segment Intersection

Thematic Map Overlay

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,

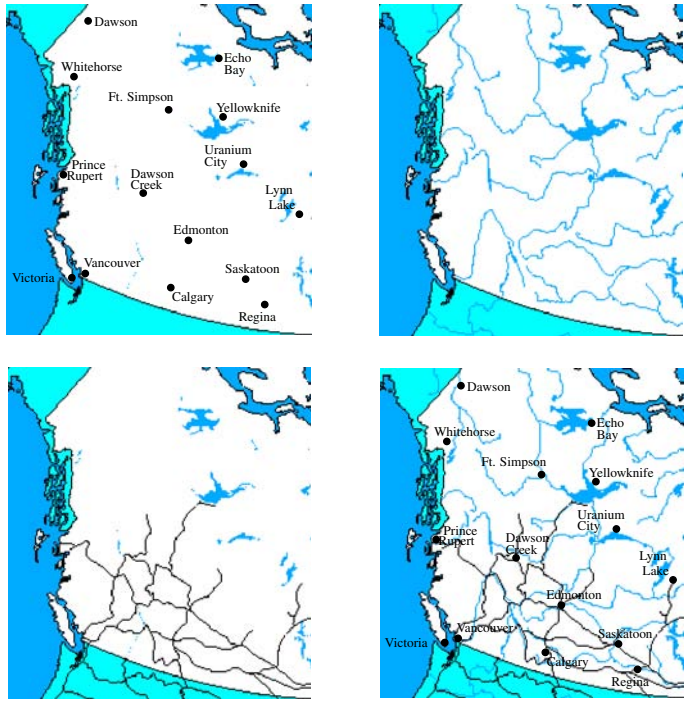


Figure 2.1
Cities, rivers, railroads, and their overlay in western Canada

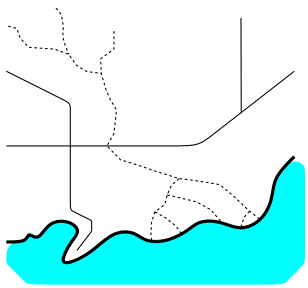
it can still be difficult to spot it on the map. To make maps more readable, geographic information systems split them into several *layers*. Each layer is a thematic map, that is, it stores only one type of information. Thus there will be a layer storing the roads, a layer storing the cities, a layer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

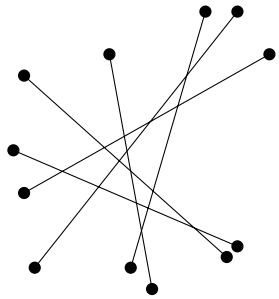
2.1 Line Segment Intersection



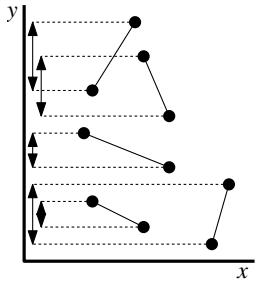
We first study the simplest form of the map overlay problem, where the two map layers are networks represented as collections of line segments. For example, a map layer storing roads, railroads, or rivers at a small scale. Note that curves can be approximated by a number of small segments. At first we 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.



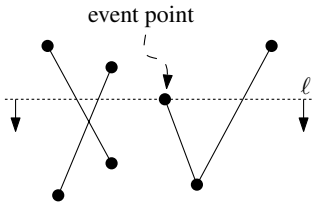
This doesn't seem like a challenging problem: we can simply take each pair of segments, compute whether they intersect, and, if so, report their intersection point. This brute-force algorithm clearly requires $O(n^2)$ time. In a sense this is optimal: when each pair of segments intersects any algorithm 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

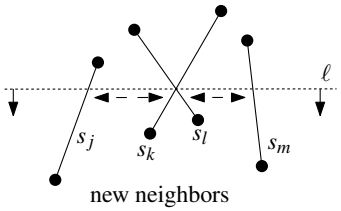
segments. While we sweep the imaginary line, we keep track of all segments intersecting it—the details of this will be explained later—so that we can find the pairs we need.



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.

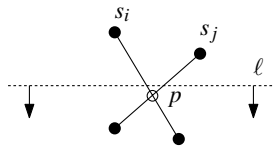


Before trying to turn these ideas into an efficient algorithm, we should convince ourselves that the approach is correct. We have reduced the number of pairs to be tested, but do we still find all intersections? In other words, if two segments s_i and s_j intersect, is there always a position of the 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.

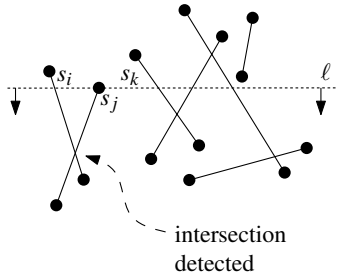
Lemma 2.1 *Let s_i and s_j be two non-horizontal segments whose interiors intersect in a single point p, and assume there is no third segment passing through p. Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.*

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. □

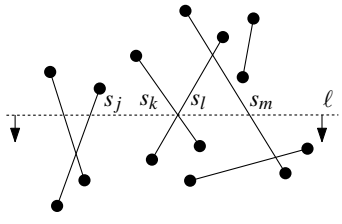


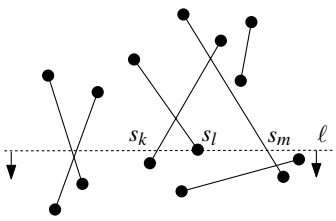
So our approach is correct, at least when we forget about the nasty cases mentioned earlier. Now we can proceed with the development of the plane sweep algorithm. Let's briefly recap the overall approach. We imagine moving a horizontal sweep line ℓ downwards over the plane. The sweep line halts at certain 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.



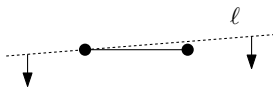


When the event point is the lower endpoint of a segment, its two neighbors now become adjacent and must be tested for intersection. If they intersect below the sweep line, then their intersection point is an event point. (Again, this event could have been detected already.) Assume three segments s_k, 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.



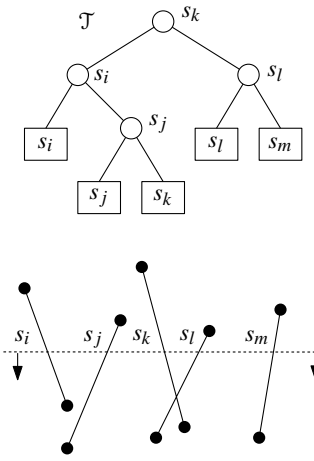
First of all we need a data structure—called the *event queue*—that stores the events. We denote the event queue by $\mathcal{Q}$. We need an operation that removes the next event that will occur from $\mathcal{Q}$, and returns it so that it can be treated. This event is the highest event below the sweep 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.

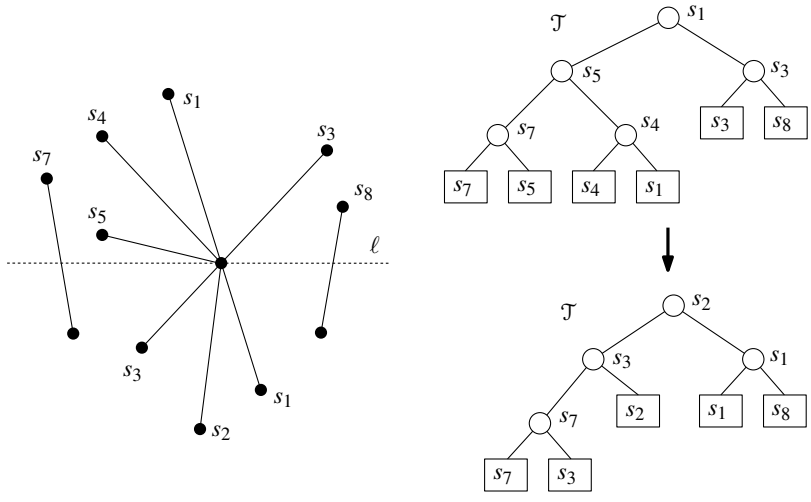


Figure 2.2
An event point and the changes in the status structure

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.

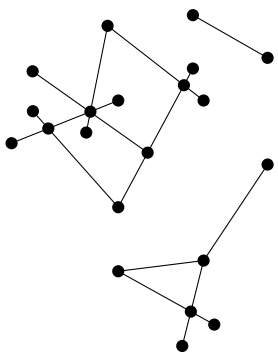
Now assume that p is not an endpoint of a segment. All we need to show is that p will be inserted into $\mathcal{Q}$ at some moment. Note that all segments that are involved have p in their interior. Order these segments by angle around p, and let s_i and s_j be two neighboring segments. Following the 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 $\mathcal{Q}$. □

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:

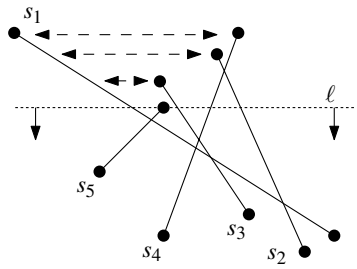
$$n_v - n_e + n_f \geqslant 2.$$

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. □

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