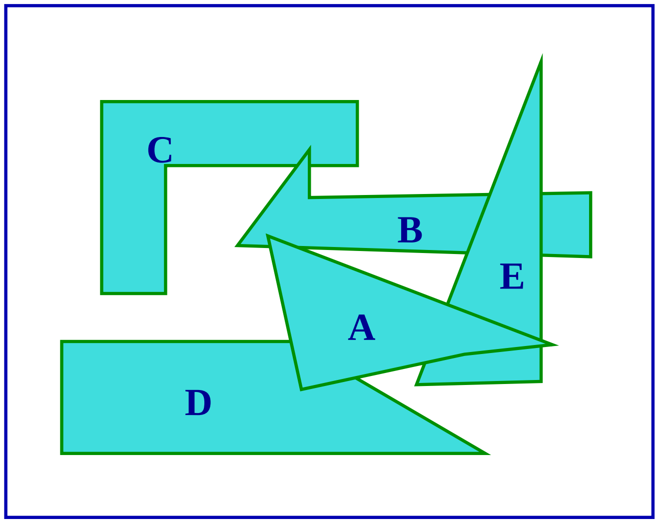
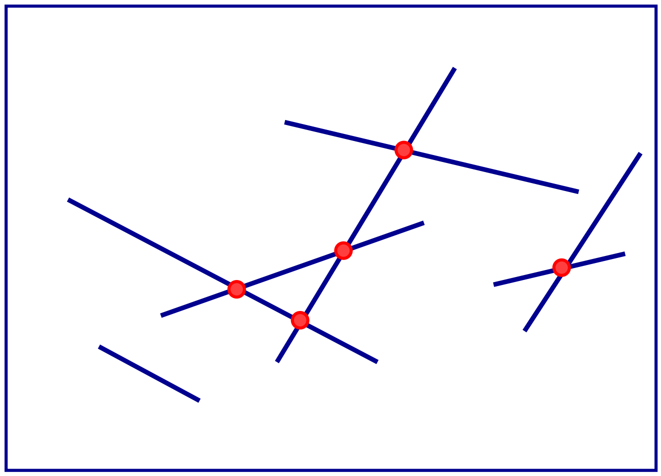
Intersection Problems

- Determine pairs of intersecting objects?



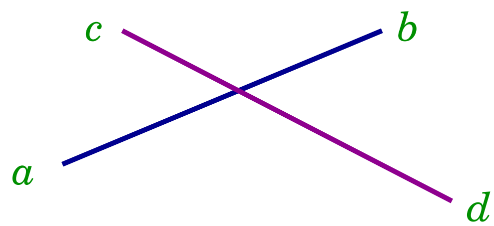
- Complex shapes formed by boolean operations: intersect, union, diff.
- Collision detection in robotics and motion planning.
- Visibility, occlusion, rendering in graphics.
- Map overlay in GISs: e.g. road networks on county maps.

Line Segment Intersection



- **The most basic problem: intersections among line segments in R^2.**
- **General enough to capture polygons, 2D projections of 3D scenes.**
- **Naive algorithm: Check all pairs. $O(n^2)$.**
- **If k intersections, then ideal will be $O(n \log n + k)$ time.**
- **We will describe a $O((n+k)\log n)$ solution. Also introduce a new technique : plane sweep.**

Primitive Operation



- **How to decide if two line segments ab and cd intersect?**
- **Write the equations of each segment in parametric form:**

$$\begin{aligned} p(s) &= (1-s)a + sb \qquad \textbf{for } 0 \le s \le 1 \\ q(t) &= (1-t)c + sd \qquad \textbf{for } 0 \le t \le 1 \end{aligned}$$

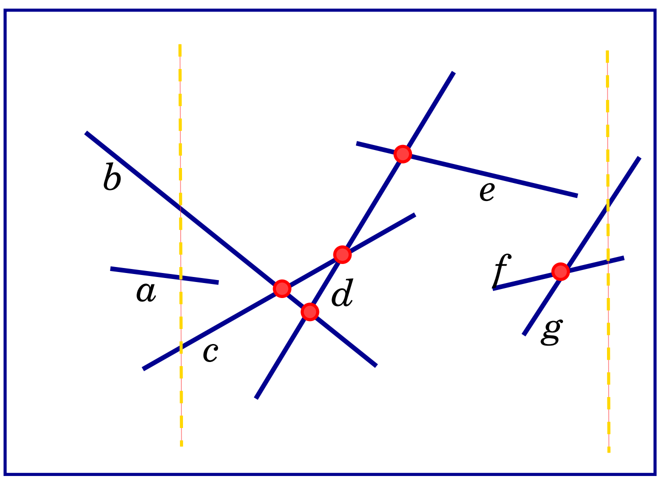
- **An intersection occurs if for some values of s, t, we get $p(s) = q(t)$.**
- **In terms of x, y , we get:**

$$\begin{aligned} (1-s)a_x + sb_x &= (1-t)c_x + td_x \\ (1-s)a_y + sb_y &= (1-t)c_y + td_y \end{aligned}$$

- **Solve for s, t and see if they lie in $[0, 1]$.**

Plane Sweep Algorithm

- **Input $S = \{s_1, s_2, \ldots, s_n\}$; each segment given by pair of endpoints.**
- **Report all intersecting segment pairs.**
- **We move an imaginary vertical line from left to right.**
- **Maintain vertical order of segments intersecting the sweep line; order changes only at discrete times.**

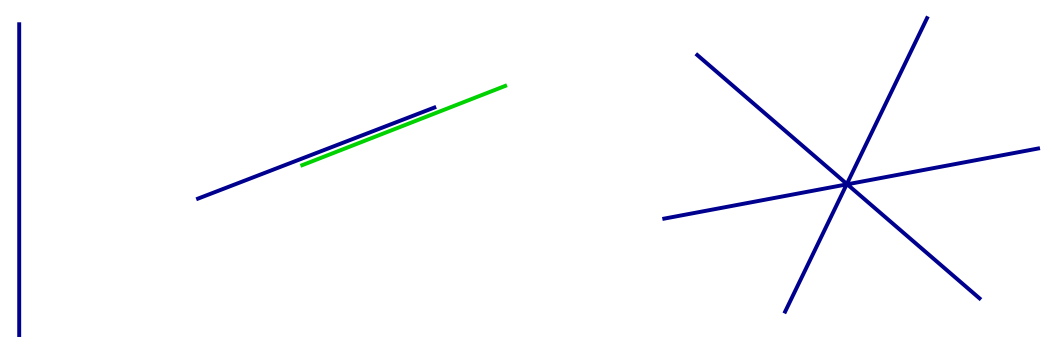


- **Intersections among S inferred by looking at localized information along sweep line.**

Simplifying Assumptions

- **In order to avoid dealing with technical special cases, which obscure the main ideas, we assume:**

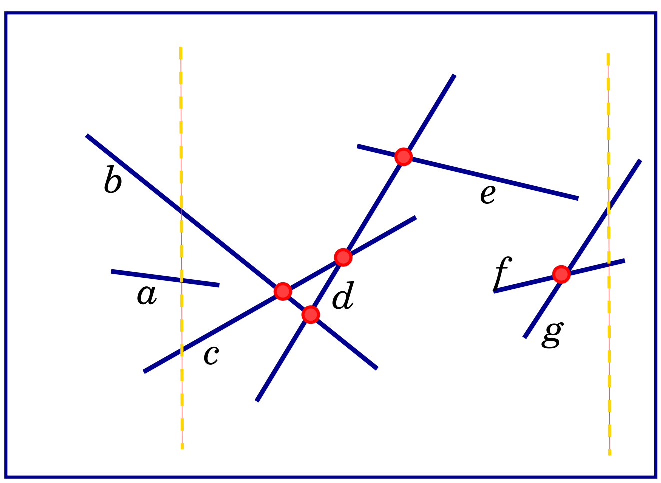
  1. **No segment is vertical.**
  2. **Any two segments intersect in at most one point.**
  3. **No three or more lines intersect in a common point.**



Data Structures

- **Sweep Line Status:** **Maintain the segments intersecting the sweep line ℓ in sorted order, from top to bottom.**

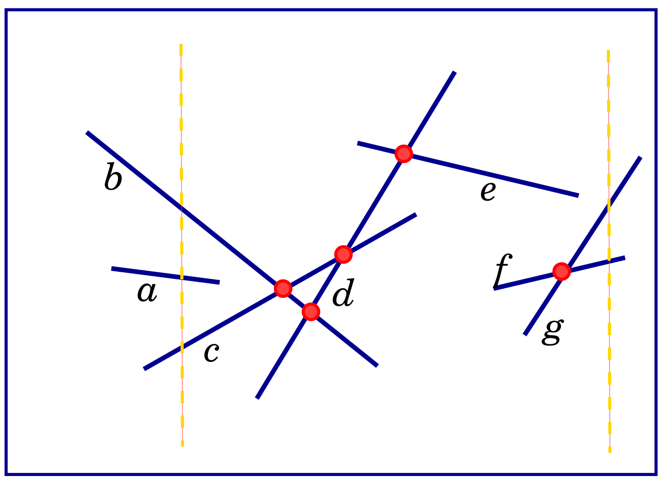
1. **Balanced binary tree.**
2. **Insert, delete, search in $O(\log n)$.**
3. **The choice of the key? The y-position of $s \cap \ell$ changes as ℓ moves.**
4. **Use "variable" key, the equation of the line: $y = mx + c$.**
5. **Plugging in x fixes y coordinate.**
6. **All order-comparisons among segments done for a fixed x-position of ℓ.**



Data Structures

- **Event Queue:** **Events represent instants when sweep line order changes.**

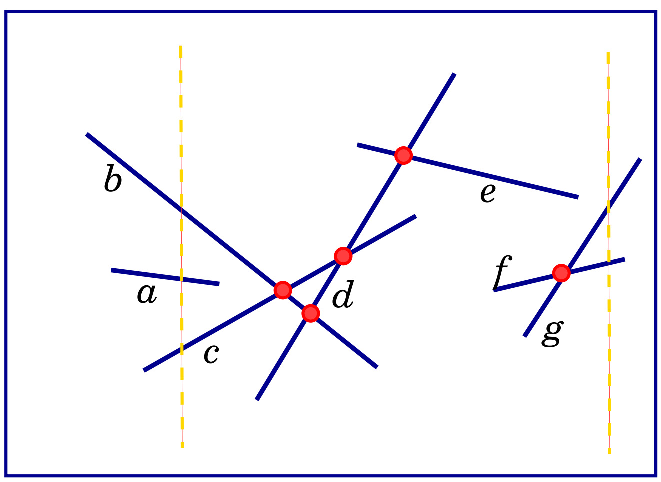
1. **While the y-coordinates of segments along ℓ change continuously, their ordering changes only at discrete steps.**



2. **Order changes when a segment begins, a segment ends, or two segments intersect.**
3. **Segments begin/end events known in advance; the intersection events generated dynamically.**
4. **Maintain events in x-sorted order, in a balanced binary tree.**

What's the Idea?

- **The algorithm requires knowing the intersection points (for event queue).**
- **But that's whole problem we are trying to solve!**
- **We don't need all intersections up front; only before the sweep line reaches them.**

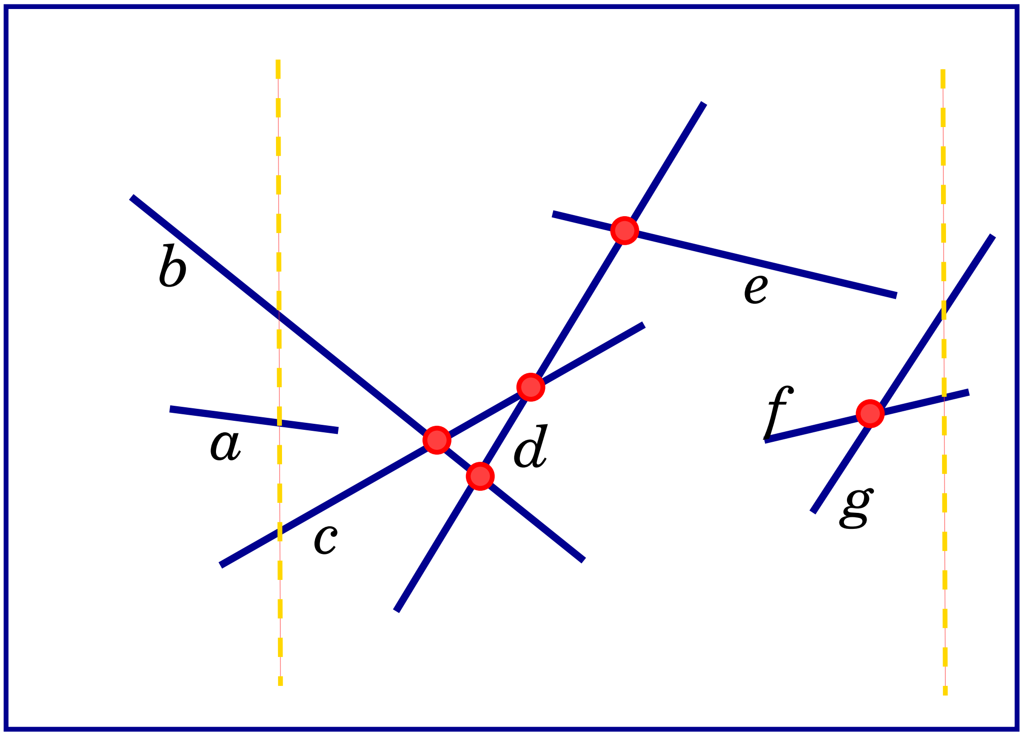


- **Plane sweep's idea is to maintain only the "most immediate" intersections.**
- **At any time, the Event Queue schedules only those intersections that are between two neighboring segments in the sweep line order.**

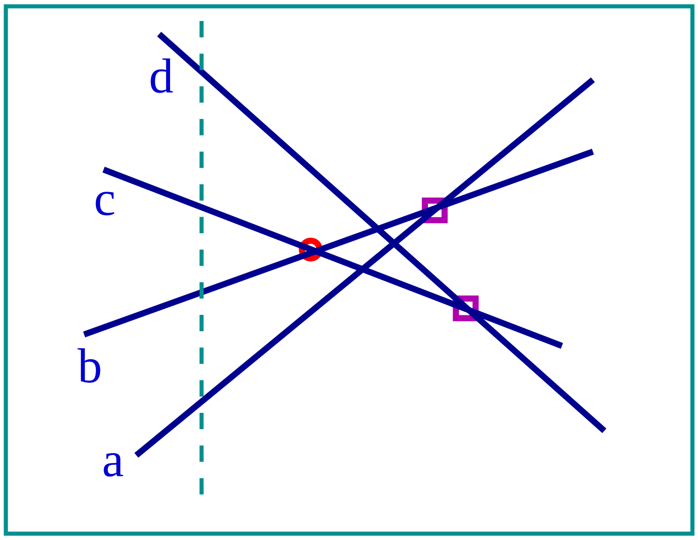
Algorithm

1. **Initialize Event Queue with endpoints of S, in sorted order.**
2. **While queue non-empty, extract the next event. Three cases:**
3. **[Left endpoint of a segment s_i]**
   - **Insert s_i into sweep line status tree;**
   - **If s_i intersects its above or below neighbors, add those intersections to Event Queue.**
4. **[Right endpoint of a segment s_i]**
   - **Delete s_i from sweep line status tree;**
   - **If s_i's neighbors intersect, add that intersection to Event Queue.**
5. **[Intersection of s_i and s_j]**
   - **Swap the order of s_i and s_j;**
   - **Delete intersection events involving s_i and s_j from the Event Queue.**
   - **Possibly add new intersection events between s_i, s_j and their new neighbors.**

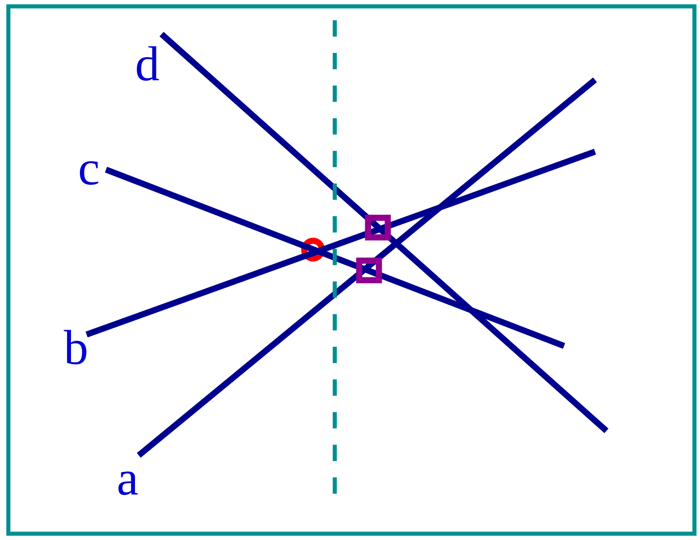
Illustration



Illustration



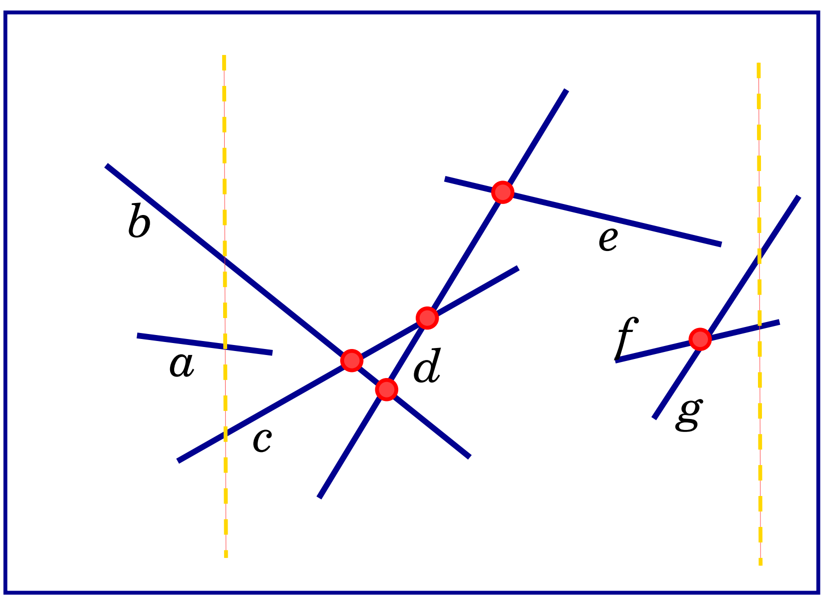
d
c
b
a
Sweep Line Status



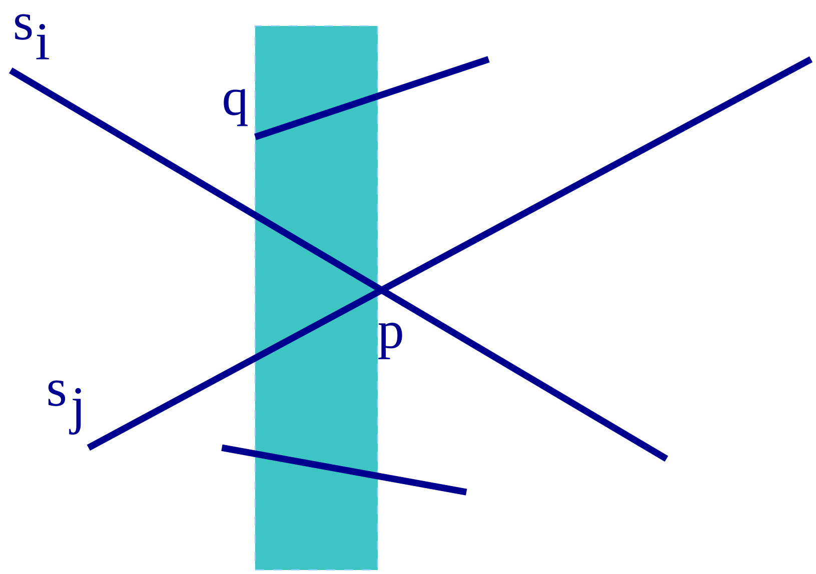
d
b
c
a
Sweep Line Status

Correctness

1. Algorithm only checks intersections between segments that are adjacent along sweep line at some point.
2. The algorithm obviously doesn't report false intersections.
3. But can it miss intersections?
4. No. If segments s_i and s_j intersect at point p, then s_i and s_j are neighbors just before the sweep line reaches p.

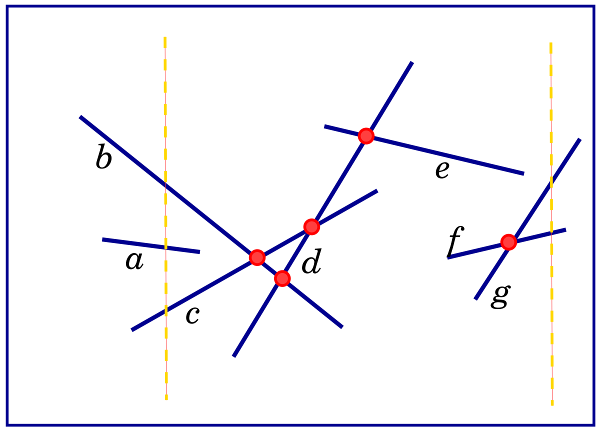


Proof



- No three or more segments intersect at one point, so only s_i and s_j intersect at p.
- For sweep line placed just before p, there cannot be any segment between s_i and s_j; otherwise, there must be another event before p.
- Let q be the event before p. Then, the order of segments along sweep line after q and before p must remain unchanged.
- Thus, s_i and s_j are adjacent in the sweep line status tree when p is processed.

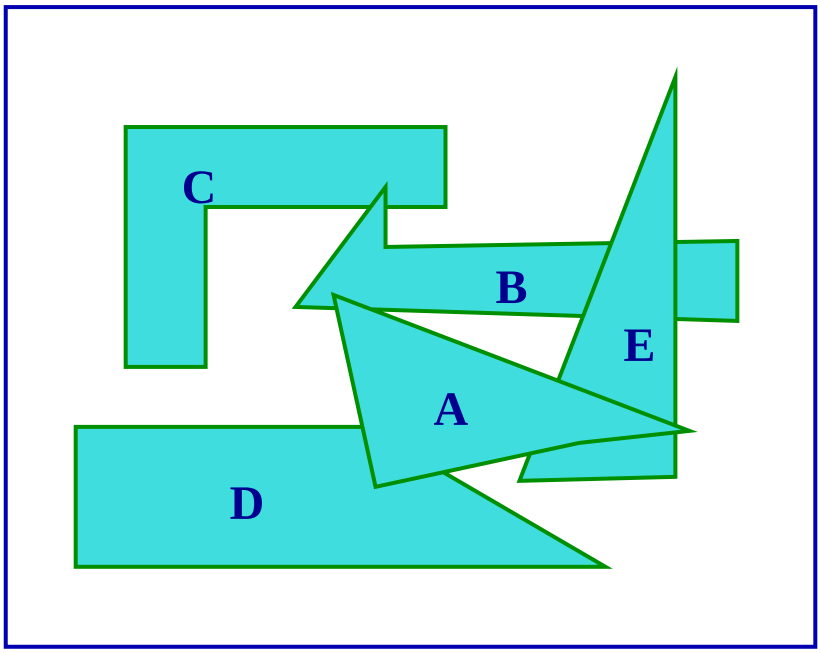
Complexity



- **Number of events processed is $2n + k$.**
- **Number of events scheduled and descheduled can be larger.**
- **But each intersection processing creates at most 2 new events, and deletes at most 2 old events, so $O(k)$ events handled.**
- **Handling an event require $O(1)$ changes to the status tree, and $O(1)$ insert/delets in Event Queue.**
- **Thus, processing cost per event is $O(\log n)$.**
- **Time complexity is $O((n + k)\log(n + k))$.**

Subdivision Representation

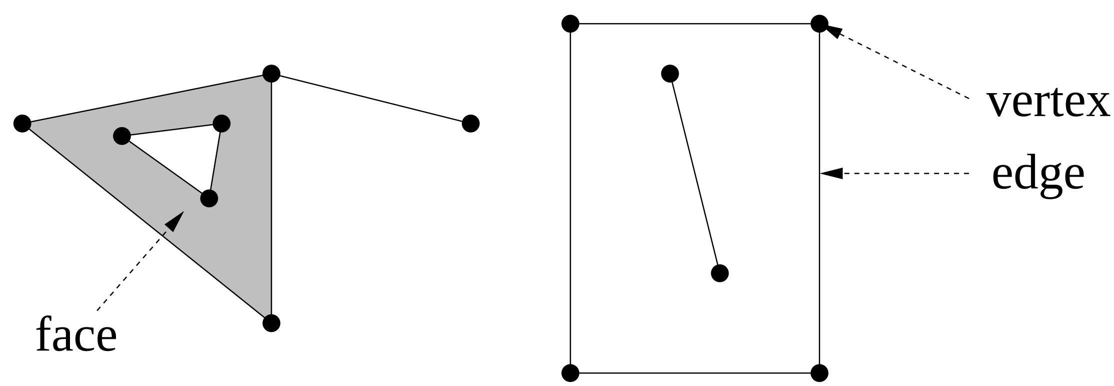
- **How do we organize a planar subdivision for easy access to useful information?**



- **E.g. how to tell that objects** A, B, E **create a hole? Which edge bound that hole?**
- **The planar subdivision, or planar straight line graph, is the embedding of a geometric graph.**

Planar Subdivision

- A natural partition of the plane into regions (faces), bounded by cycles of line segments (edges), with points as their endpoints (vertices).
- Using a general topological notation, these are also called 2- 1-, and 0-dimensional faces.



- Planar subdivision are quite important: triangulations, Voronoi diagrams etc.
- Develop a data structure that provides primitives like "list the edges bounding a face", "list the edges that surround a vertex."

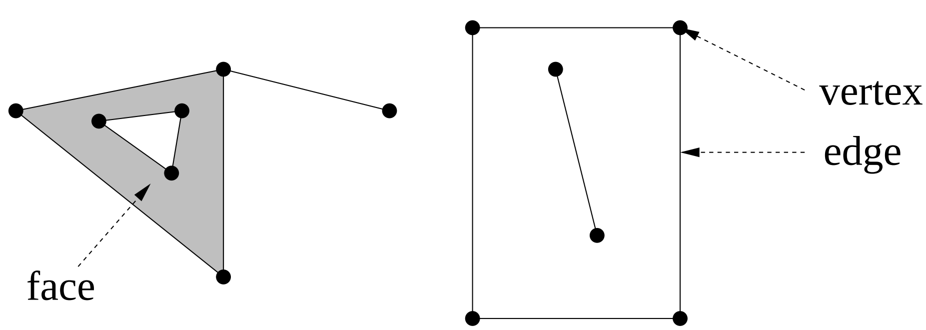
Planar Graphs

- **Planar subdivisions have attractive properties because they are embeddings of planar graphs.**
- **A graph is planar if it can be drawn in the plane so that no two edges cross. An embedding is any such drawing.**
- **One of the most famous properties of planar graphs is the Euler's Formula, linking the number of vertices, edges, and faces.**

$$V - E + F = 2$$

- **If the graph has multiple, C, disconnected pieces, then**

$$V - E + F - C = 1$$

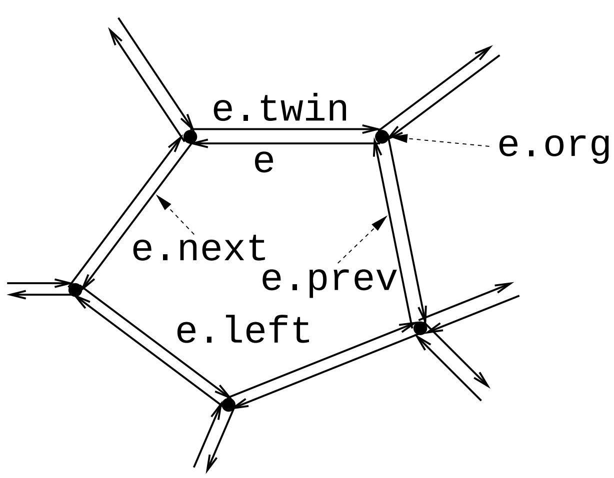


DCEL Representation

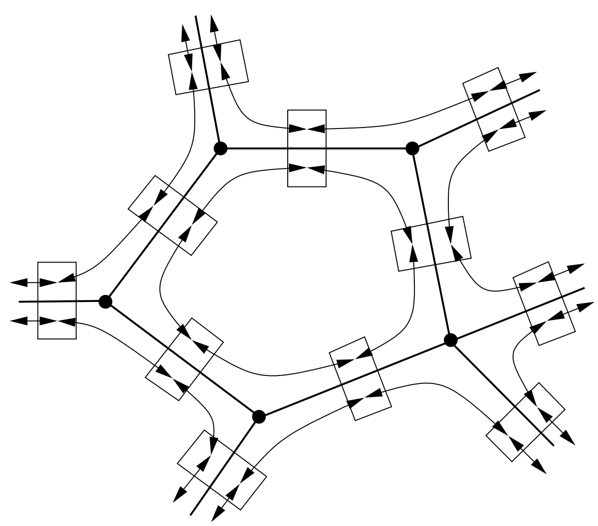
- **Euler's formula can be used to show that**
  $$E \leq 3V - 6 \qquad F \leq 2V - 4.$$
- **Thus, the number of vertices, edges, and faces are all linearly related.**
- **DCEL (Doubly Connected Edge List) is one of the most commonly used representations. For simplicity, we now assume faces do not have holes in them.**
- **DCEL is an edge-based structure, which links together the three sets of records: vertex records, edge records, and face records.**

Details

1. A vertex v stores its coordinates, and a pointer to one of its incident edges, v.inc-edge, for which v is the origin.
2. Each undirected edge split into two oppositely directed edges, called twins, each pointing to the other.
3. Directed edge e points to its origin vertex, e.org. Note that e.dest = e.twin.org
4. Edge e also has a pointer to its left face, e.left. Finally, e.next and e.prev point to the next and previous edges of this face, in counterclockwise order.



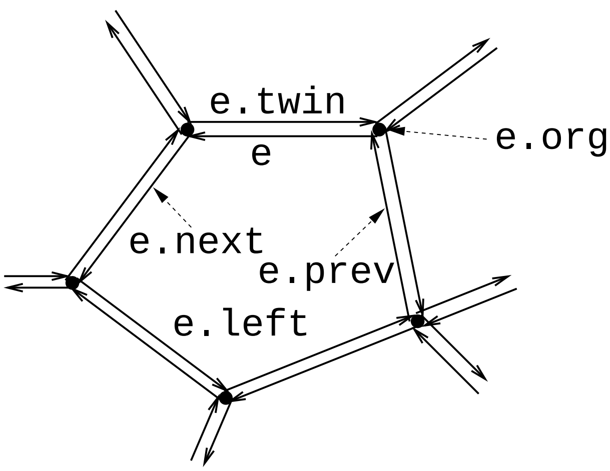
DCEL


Alternative view

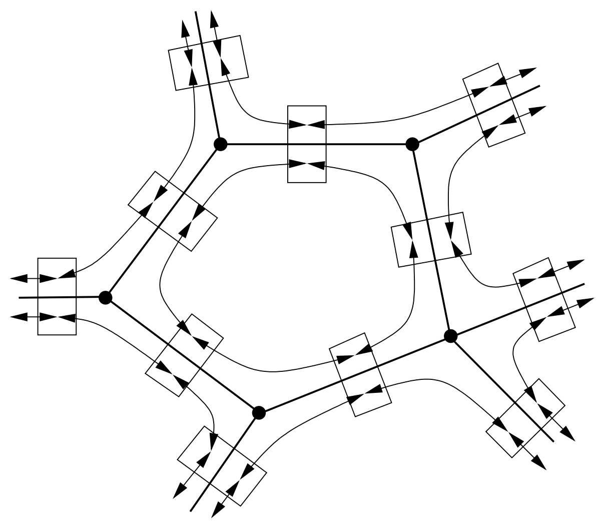
Example

1. Each face f stores a pointer, **f.inc-edge**, to one of its edges. (Read the book for complete description of how to handle various special cases.)

2. Figure shows two ways to visualize DCEL.



DCEL



Alternative view

3. How do we use this data structure?

4. Consider overlaying two subdivisions. Or, constructing the segment intersection subdivision.

Merging Subdivisions

1. **Two subdivisions S_1 and S_2, stored as DCELs. Compute the DCEL of $S_1 \cup S_2$.**
2. **Use plane sweep to find segment intersections, and update DCEL.**
3. **Many small but tedious details. Book describes them. I will explain the most interesting case: intersection event.**
4. **Merge** (a_1, b_1)
   - **Create new vertex v at the intersection point.**
   - **Split the two intersecting edges. Let a_2 and b_2 be the new edge pieces.**
   - **Link them together** $\mathrm{Splice}(a_1, a_2, b_1, b_2)$**.**

Split and Splice

- **Split(edge &a1, edge &a2)**

  a2 = new edge(v, a1.dest());
  a2.next = a1.next; a1.next.prev = a2;
  a1.next = a2; a2.prev = a1; a1t = a1.twin; a2t = a2.twin;
  a2t.prev = a1t.prev; a1t.prev.next = a2t;

  a1t.prev = a2t; a2t.next = a1t;

- **Splice(&a1, &a2, &b1, &b2)**

  a1.next = b2; b2.prev = a1; b2t.next = a2; a2.prev = b2t;
  a2t.next = b1t; b1t.prev = a2t; b1.next = a1t; a1t.prev = b1;