DCEL

Record for each face, edge and vertex:
- Geometric information
- Topological information
- Attribute information

aka Half-Edge Structure
- 3 arrays: Vertices, Edges, Faces, (V,E,F)
- Coordinates are stored only at V.
- Avoid data duplication
- Common operation need to support: Traversing along a line, and find its intersections with all edges and faces.

**On the Agenda**
- The DCEL Data Structure
- Line Segment Intersection
- Plane Sweep
- Euler's Formula
DCEL

- **Vertex record:**
  - Coordinates
  - Pointer to one half-edge that has v as its origin

- **Face record:**
  - Pointer to one half-edge on its boundary

- **Half-edge record:**
  - Pointer to its origin: origin(e)
  - Pointer to its twin half-edge: twin(e)
  - Pointer to the face it bounds: IncFace(e) (face lies to left of e when traversed from origin to destination)
  - Next and previous edge on boundary of IncFace(e): next(e), prev(e)

- **Support for:**
  - Walk around boundary of given face
  - Visit all edges incident to vertex v (how ?)

- **Queries:**
  - Most queries are O(1)

**Line Segment Intersection**

- **Theorem:** Segments \((p_1, p_2)\) and \((p_3, p_4)\) intersect in their interior iff \(p_1 \) and \(p_2\) are on different sides of the line \(p_3p_4\) and \(p_3\) and \(p_4\) are on different sides of the line \(p_1p_2\).

- **Computational robust, but computationally costly.**

- **Special cases:**
  - \(p_4 \parallel p_2\)
  - \(p_3 \parallel p_1\)

**Computing the Intersection**

\[
p(t) = p_1 + (p_2 - p_1)t \quad 0 \leq t \leq 1 \\
q(s) = q_1 + (q_2 - q_1)s \quad 0 \leq s \leq 1
\]

**Question:** What is the meaning of other values of \(s\) and \(t\) ?

Solve (2D) linear vector equation for \(t\) and \(s\):

\[
p(t) = q(s)
\]

check that \(t \in [0, 1]\) and \(s \in [0, 1]\)

**Point in Polygon**

- **Given a polygon \(P\) with \(n\) sides, and a point \(q\), decide whether \(q \in P\).**

- **Solution A:** Count how many times a ray \(r\) originating at \(q\) intersects \(P\). Then \(q \in P\) iff this number is odd.

- **Complexity:** \(O(n)\)

- **Question:** Are there special cases ?
Given $n$ line segments, two questions arise:

- Does any pair intersect? (studied in this class)
- Report all intersections

**Obvious algorithm:** $O(n^2)$ – checking all pairs.

**Line-sweep:** $O(n \log n)$ time determine if there is an intersection.

Applications 1: Checking VLSI design correctness

Need to decide if two components intersect (bug in design).

Applications 2: Finding all junctions and/or intersection points

Applications: Map overlaying: compute all points at which a road crosses a river.

**Computing intersection of two segments**

1. Find the intersection point $p$ of the lines (solving a linear system with two equations).
2. Find if $p$ lies on both segments

**Computing intersection of two segments - cont.**

First check if their bounding boxes of the segments intersect.

Bounding box-the axis-parallel that has the endpoints of the segment as a pair of diagonal vertices.

A quick operation that does not involve division.

Given a set $S$ of numbers, a standard balanced search tree supports:
- $\text{insert}(x, S)$, $\text{delete}(x, S)$, $\text{find}(x, S)$

Each in $O(\log n)$ time (where $n=|S|$).

It can also support the operation $\text{succ}(x, S)$, defined as finding the smallest element of $S$ which is strictly larger than $S$.

**Examples:**
$S = \{10, 20, 30\}$;
$\text{succ}(30, S) = 10$;
$\text{succ}(10, S) = \text{succ}(12, S) = 20$;
$\text{succ}(30, S) = \text{succ}(40, S) = \text{UNDEFINED}$. 
Leftover from Data-structures

```c
Succ(pNODE p, float x){
    p = root;
    x_tmp = INFINITY;  /* x_tmp – temporally value */
    while (p ≠ NULL){
        if (p->key ≤ x) p=p->right;
        else {
            x_tmp = min(x_tmp, p->key);
            p = p->left ;
        }
    }
    return x_tmp;
}
```

Sweep-line algorithm

- Sweep a vertical line from left to right (conceptually replacing x-coordinate with time).
- Maintain the status - a dynamic set S of the segments that intersect the sweep line, ordered (tentatively) by y-coordinate of intersection. (so the lowest segment appears first one the list)
- The status is changed only when
  - new segment is encountered (left endpoints),
  - existing segment finishes (right endpoint)
- Event points are therefore segment endpoints.

The status of the linesweep

The status S is the list of segments that linesweep l intersects (in the order from bottom to top).

Definition: an event happens when l start/stop intersecting a segment.

Note: the status is not changed between events, so l can jump from an event to the next event.

Left endpoint event:

- For a left endpoint of segment s:
  - Add segment s to the status S.
  - Check for intersection between s and its neighbors in S.
- (Will later explain how the neighbors are found)

Example: a is checked for intersection with c and intersection with b.
**Right endpoint event**

For a right endpoint of segment \( s \):
- Delete segment \( s \) from dynamic set \( S \).
- Check for intersection between neighbors \( s \) and its neighbors in \( S \).

**Example:** \( c \) is checked for intersection with \( b \).

![Diagram of Right endpoint event](image)

**Theorem:** If there is an intersection point, the algorithm finds it.

**Proof:** Let \( p \) be the leftmost intersection point.

Consider the last event before (to the left of) \( p \), at which \( c \) or \( b \) are born.

If they are not neighbors on \( l \), it is because another segment, say \( a \), separates between them.

But then either \( a \) has a right endpoint to the left of \( p \) and then \( c \), and \( b \) become neighbors.

Or \( a \) intersects \( b \) or \( c \) at a point to the left of \( p \). A contradiction to \( p \) be the leftmost point.

![Diagram of Theorem](image)

**Maintaining the status \( S \)**

- We maintain \( S \) - the list of segments that intersect in a sorted search tree \( T \), sorted by the order they appear along \( l \).
- When we insert a new segment \( g \), we compare \( y' \), the \( y \)-coordinates of intersections of segments with \( l \).
- Example: Since \( d \) is the root of \( T \), we compute the intersection point of \( f \) with \( d \), call it \( d_f \).
- We compare \( y' \) and \( d_f \) and deduce that \( d_f \) is above \( d \), so it should be inserted into the right subtree.
- Continue recursively.

![Diagram of Maintaining the status S](image)

**Operations on the tree**

- \( \text{Insert}(T,s) \) - Insert the segment \( s \) into the tree \( T \).
- \( \text{Delete}(T,s) \)

\( \text{Above}(T,s) \) - Find the segment just below \( s \).

Example: \( \text{Above}(T,b) = c \) (successor operation in \( T \)).

\( \text{Below}(T,s) \) - Analogous operation (predecessor).

![Diagram of Operations on the tree](image)

**AnySegmentsIntersect(\( S \)) - pseudocode**

Create an empty set \( T \).
Sort endpoints of the segments in \( S \) from left to right.
FOR each endpoint \( p \) in the sorted list of endpoints do {
    IF \( p \) is the left endpoint of a segment \( s \) then
        IF \( \text{Above}(T,s) \) exist and \( \text{Intersects} \) \( s \) or
            \( \text{Below}(T,s) \) exists and intersects then
                Return TRUE (segment intersection)
        ELSEIF \( p \) is the right endpoint of \( s \) then
            IF both \( \text{Above}(T,s) \) and \( \text{Below}(T,s) \) exist
                Delete \( T \).
        \}
    ELSE
        Return "no two segments intersect"
Running time

- There are $2n$ endpoints – $O(n \log n)$ time for sorting
- Each left endpoint event involved
  - Insertion into the tree $O(\log n)$
  - Finding successor/predecessor $O(\log n)$
  - Checking intersection with Above/Below – $O(1)$
- Each right endpoint event involved
  - Deletion from the tree $O(\log n)$
  - Finding successor/predecessor $O(\log n)$
  - Checking intersection between Above/Below – $O(1)$
- Total – $O(n \log n)$

A graph is $k$-connected if between every two vertices there are $k$ edge-disjoint paths.

Planar Graphs

A planar graph is a graph whose vertices and edges can be embedded in $\mathbb{R}^2$ such that its edges do not intersect. Every planar graph can be drawn as a straight-line plane graph.

Triangulation

A triangulation is a straight line plane graph whose faces are all triangles.

A Delaunay triangulation of a set of points is the unique set of triangles such that the circumcircle of any triangle does not contain any other point.

The Delaunay triangulation avoids long and skinny triangles.
Euler Formula

For a connected planar graph:

\[ v + f - e = 2 \]

- \( v \) = \# vertices
- \( f \) = \# faces
- \( e \) = \# edges

**Theorem:** In a triangulation:

1. \( e = f = O(v) \)
2. The average vertex degree is \(~6\).

**Proof:** In such a mesh, \( f = 2e/3 \).

By Euler’s formula: \( v + 2e/3 - e = 2 \) hence \( e = 3(v-2) \) and \( f = 2(v-2) \).

So \( \text{Average(deg)} = 2e/v = 6(v-2)/v \)

\(~6\) for large \( v \).