Planar Subdivisions and Point Location

Carola Wenk

Based on:
Computational Geometry: Algorithms and Applications
and David Mount’s lecture notes
Planar Subdivision

- Let $G=(V,E)$ be an undirected graph.
- $G$ is planar if it can be embedded in the plane without edge crossings.

- A planar embedding (=drawing) of a planar graph $G$ induces a **planar subdivision** consisting of vertices, edges, and faces.
Doubly-Connected Edge List

The doubly-connected edge list (DCEL) is a popular data structure to store the geometric and topological information of a planar subdivision. It contains records for each face, edge, vertex (Each record might also store additional application-dependent attribute information.) It should enable us to perform basic operations needed in algorithms, such as walk around a face, or walk from one face to a neighboring face.

The DCEL consists of:

- For each vertex \( v \), its coordinates are stored in \( \text{Coordinates}(v) \) and a pointer \( \text{IncidentEdge}(v) \) to a half-edge that has \( v \) as its origin.
- Two oriented half-edges per edge, one in each direction. These are called twins. Each of them has an origin and a destination. Each half-edge \( e \) stores a pointer \( \text{Origin}(e) \), a pointer \( \text{Twin}(e) \), a pointer \( \text{IncidentFace}(e) \) to the face that it bounds, and pointers \( \text{Next}(e) \) and \( \text{Prev}(e) \) to the next and previous half-edge on the boundary of \( \text{IncidentFace}(e) \).
- For each face \( f \), \( \text{OuterComponent}(f) \) is a pointer to some half-edge on its outer boundary (null for unbounded faces). It also stores a list \( \text{InnerComponents}(f) \) which contains for each hole in the face a pointer to some half-edge on the boundary of the hole.
Complexity of a Planar Subdivision

• The complexity of a planar subdivision is:
  \#vertices + \#edges + \#faces = n_v + n_e + n_f

• Euler’s formula for planar graphs:
  1) \( n_v - n_e + n_f \geq 2 \)
  2) \( n_e \leq 3n_v - 6 \)

2) follows from 1):

Count edges. Every face is bounded by \( \geq 3 \) edges.
Every edge bounds \( \leq 2 \) faces.
\[
\Rightarrow 3n_f \leq 2n_e \Rightarrow n_f \leq 2/3n_e \\
\Rightarrow 2 \leq n_v - n_e + n_f \leq n_v - n_e + 2/3 n_e = n_v - 1/3 n_e \\
\Rightarrow 2 \leq n_v - 1/3 n_e
\]

• Hence, the complexity of a planar subdivision is \( O(n_v) \), i.e., linear in the number of vertices.
Point Location

• **Point location task:** Preprocess a planar subdivision to efficiently answer point-location queries of the type: Given a point $p=(p_x, p_y)$, find the face it lies in.

• **Important metrics:**
  – Time complexity for preprocessing
    = time to construct the data structure
  – Space needed to store the data structure
  – Time complexity for querying the data structure
**Slab Method**

- **Slab method:**
  Draw a vertical line through each vertex. This decomposes the plane into slabs.

- In each slab, the vertical order of the line segments remains constant.
- If we know in which slab \( p \) lies, we can perform binary search, using the sorted order of the segments in the slab.
- Find slab that contains \( p \) by binary search on \( x \) among slab boundaries.
- A second binary search in slab determines the face containing \( p \).
- Search complexity \( O(\log n) \), but space complexity \( \Theta(n^2) \).
Kirkpatrick’s Algorithm

• Needs a triangulation as input.
• Can convert a planar subdivision with \( n \) vertices into a triangulation:
  – Triangulate each face, keep same label as original face.
  – If the outer face is not a triangle:
    • Compute the convex hull of the subdivision.
    • Triangulate pockets between the subdivision and the convex hull.
    • Add a large triangle (new vertices \( \mathbf{a}, \mathbf{b}, \mathbf{c} \)) around the convex hull, and triangulate the space in-between.

• The size of the triangulated planar subdivision is still \( O(n) \), by Euler’s formula.
• The conversion can be done in \( O(n \log n) \) time.
• Given \( \mathbf{p} \), if we find a triangle containing \( \mathbf{p} \) we also know the (label of) the original subdivision face containing \( \mathbf{p} \).
Kirkpatrick’s Hierarchy

- Compute a sequence $T_0$, $T_1$, ..., $T_k$ of increasingly coarser triangulations such that the last one has constant complexity.
- The sequence $T_0$, $T_1$, ..., $T_k$ should have the following properties:
  - $T_0$ is the input triangulation, $T_k$ is the outer triangle
  - $k \in O(\log n)$
  - Each triangle in $T_{i+1}$ overlaps $O(1)$ triangles in $T_i$

- How to build such a sequence?
  - Need to delete vertices from $T_i$.
  - Vertex deletion creates holes, which need to be re-triangulated.

- How do we go from $T_0$ of size $O(n)$ to $T_k$ of size $O(1)$ in $k=O(\log n)$ steps?
  - In each step, delete a constant fraction of vertices from $T_i$.
- We also need to ensure that each new triangle in $T_{i+1}$ overlaps with only $O(1)$ triangles in $T_i$. 
Vertex Deletion and Independent Sets

When creating $T_{i+1}$ from $T_i$, delete vertices from $T_i$ that have the following properties:

- **Constant degree:**
  Each vertex $v$ to be deleted has $O(1)$ degree in the graph $T_i$.
  - If $v$ has degree $d$, the resulting hole can be re-triangulated with $d-2$ triangles
  - Each new triangle in $T_{i+1}$ overlaps at most $d$ original triangles in $T_i$

- **Independent sets:**
  No two deleted vertices are adjacent.
  - Each hole can be re-triangulated independently.
Independent Set Lemma

**Lemma**: Every planar graph on $n$ vertices contains an independent vertex set of size $n/18$ in which each vertex has degree at most 8. Such a set can be computed in $O(n)$ time.

Use this lemma to construct Kirkpatrick’s hierarchy:

- Start with $T_0$, and select an independent set $S$ of size $n/18$ in which each vertex has maximum degree 8. [Never pick the outer triangle vertices $a$, $b$, $c$.]
- Remove vertices of $S$, and re-triangulate holes.
- The resulting triangulation, $T_1$, has at most $17/18n$ vertices.
- Repeat the process to build the hierarchy, until $T_k$ equals the outer triangle with vertices $a$, $b$, $c$.
- The depth of the hierarchy is $k = \log_{18/17} n$.
Hierarchy Example

Use this lemma to construct Kirkpatrick’s hierarchy:

- Start with $T_0$, and select an independent set $S$ of size $n/18$ in which each vertex has maximum degree 8. [Never pick the outer triangle vertices $a$, $b$, $c$.]
- Remove vertices of $S$, and re-triangulate holes.
- The resulting triangulation, $T_1$, has at most $17/18n$ vertices.
- Repeat the process to build the hierarchy, until $T_k$ equals the outer triangle with vertices $a$, $b$, $c$.
- The depth of the hierarchy is $k = \log_{18/17} n$
Hierarchy Data Structure

Store the hierarchy as a DAG:
- The root is \( T_k \).
- Nodes in each level correspond to triangles \( T_i \).
- Each node for a triangle in \( T_{i+1} \) stores pointers to all triangles of \( T_i \) that it overlaps.

How to locate point \( p \) in the DAG:
- Start at the root. If \( p \) is outside of \( T_k \) then \( p \) is in exterior face; done.
- Else, set \( \Delta \) to be the triangle at the current level that contains \( p \).
- Check each of the at most 6 triangles of \( T_{k-1} \) that overlap with \( \Delta \), whether they contain \( p \). Update \( \Delta \) and descend in the hierarchy until reaching \( T_0 \).
- Output \( \Delta \).
Analysis

• **Query time** is $O(\log n)$: There are $O(\log n)$ levels and it takes constant time to move between levels.

• **Space complexity** is $O(n)$:
  – Sum up sizes of all triangulations in hierarchy.
  – Because of Euler’s formula, it suffices to sum up the number of vertices.
  – Total number of vertices:
    
    $n + \frac{17}{18}n + \left(\frac{17}{18}\right)^2n + \left(\frac{17}{18}\right)^3n + \ldots$
    
    $\leq \frac{1}{1-\frac{17}{18}}n = 18 \, n$

• **Preprocessing time** is $O(n \log n)$:
  – Triangulating the subdivision takes $O(n \log n)$ time.
  – The time to build the DAG is proportional to its size.
Independent Set Lemma

Lemma: Every planar graph on $n$ vertices contains an independent vertex set of size $n/18$ in which each vertex has degree at most 8. Such a set can be computed in $O(n)$ time.

Proof:
Algorithm to construct independent set:
• Mark all vertices of degree $\geq 9$
• While there is an unmarked vertex
  • Let $v$ be an unmarked vertex
  • Add $v$ to the independent set
  • Mark $v$ and all its neighbors
• Can be implemented in $O(n)$ time: Keep list of unmarked vertices, and store the triangulation in a data structure that allows finding neighbors in $O(1)$ time.
Independent Set Lemma

Still need to prove existence of large independent set.

• Euler’s formula for a triangulated planar graph on $n$ vertices:
  \[ \text{\#edges} = 3n - 6 \]

• Sum over vertex degrees:
  \[ \sum_v \text{deg}(v) = 2 \text{\#edges} = 6n - 12 < 6n \]

• **Claim:** At least $n/2$ vertices have degree $\leq 8$.
  **Proof:** By contradiction. So, suppose otherwise.
  \[ \rightarrow n/2 \text{ vertices have degree} \geq 9. \text{ The remaining have degree} \geq 3. \]
  \[ \rightarrow \text{The sum of the degrees is} \geq 9 \frac{n}{2} + 3 \frac{n}{2} = 6n. \text{ Contradiction.} \]

• In the beginning of the algorithm, at least $n/2$ nodes are unmarked. Each picked vertex $v$ marks $\leq 8$ other vertices, so including itself $9$.
  
• Therefore, the while loop can be repeated at least $n/18$ times.

• This shows that there is an independent set of size at least $n/18$ in which each node has degree $\leq 8$. 

\[ \square \]
Summing Up

- Kirkpatrick’s point location data structure needs $O(n \log n)$ preprocessing time, $O(n)$ space, and has $O(\log n)$ query time.
- It involves high constant factors though.

- Next we will discuss a randomized point location scheme (based on trapezoidal maps) which is more efficient in practice.
Trapezoidal map

- **Input:** Set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.
- **Query:** Given point $p$, report the segment directly above $p$.

Create trapezoidal map by shooting two rays vertically (up and down) from each vertex until a segment is hit. [Assume no segment is vertical.]

- **Trapezoidal map** = rays + segments
- Enclose $S$ into bounding box to avoid infinite rays.
- All faces in subdivision are trapezoids, with vertical sides.
- The trapezoidal map has at most $6n+4$ vertices and $3n+1$ trapezoids:
  - Each vertex shoots two rays, so, $2n(1+2)$ vertices, plus 4 for the bounding box.
  - Count trapezoids by vertex that creates its left boundary segment: Corner of box for one trapezoid, right segment endpoint for one trapezoid, left segment endpoint for at most two trapezoids. $\rightarrow 3n+1$
Construction

- Randomized incremental construction
- Start with outer box which is a single trapezoid. Then add one segment $s_i$ at a time, in random order.
Construction

- Let $S_i = \{s_1, ..., s_i\}$, and let $T_i$ be the trapezoidal map for $S_i$.
- Add $s_i$ to $T_{i-1}$.
- Find trapezoid containing left endpoint of $s_i$. [Point location; details later]
- Thread $s_i$ through $T_{i-1}$, by walking through it and identifying trapezoids that are cut.
- “Fix trapezoids up” by shooting rays from left and right endpoint of $s_i$ and trim earlier rays that are cut by $s_i$. 
Analysis

Observation: The final trapezoidal map $T_i$ does not depend on the order in which the segments were inserted.

Lemma: Ignoring the time spent for point location, the insertion of $s_i$ takes $O(k_i)$ time, where $k_i$ is the number of newly created trapezoids.

Proof:

• Let $k$ be the number of ray shots interrupted by $s_i$.
• Each endpoint of $s_i$ shoots two rays
  \[ k_i = k + 4 \] rays need to be processed
• If $k=0$, we get 4 new trapezoids.
• Create a new trapezoid for each interrupted ray shot; takes $O(1)$ time with DCEL
Analysis

Total runtime (without point location): $\sum_{i=1}^{n} k_i$

- Best case: $k_i = O(1)$, so $\sum_{i=1}^{n} k_i = O(n)$.
- Worst case: $k_i = O(i)$, so $\sum_{i=1}^{n} k_i = O(n^2)$.

- Insert segments in *random* order:
  - $\Pi = \{\text{all possible permutations/orders of segments}\}$; $|\Pi| = n!$ for $n$ segments
  - $k_i = k_i(\pi)$ for some random order $\pi \in \Pi$
  - We will show that $E(k_i) = O(1)$
  - $\Rightarrow$ Expected runtime $E(T) = E(\sum_{i=1}^{n} k_i) = \sum_{i=1}^{n} E(k_i) = O(\sum_{i=1}^{n} 1) = O(n)$
Analysis

Theorem: $E(k_i) = O(1)$, where $k_i$ is the number of newly created trapezoids created upon insertion of $s_i$, and the expectation is taken over all segment permutations of $S_i = \{s_1, ..., s_i\}$.

Proof:

• $T_i$ does not depend on the order in which segments $s_1, ..., s_i$ were added.
• Of $s_1, ..., s_i$, what is the probability that a particular segment $s$ was added last?
• $1/i$
• We want to compute the number of trapezoids that would have been created if $s$ was added last.
Analysis

• A trapezoid $\Delta$ depends on $s$ if $\Delta$ would be created by $s$ if $s$ was added last.
• We want to count trapezoids that depend on $s$, and then compute the expectation over all choices of $s$.
• Let $\delta(\Delta,s)=1$, if $\Delta$ depends on $s$. And $\delta(\Delta,s)=0$, otherwise.

- Random variable $k_i(s)=\#\text{trapezoids added when } s \text{ was inserted last in } S_i$.
- $k_i(s)=\sum_{\Delta \in T_i} \delta(\Delta, s)$
- $E(k_i)=\sum_{s \in S_i} k_i(s) P(s) = \frac{1}{i} \sum_{s \in S_i} k_i(s) = \frac{1}{i} \sum_{s \in S_i} \sum_{\Delta \in T_i} \delta(\Delta, s)$
Analysis

- Random variable \( k_i(s) = \# \text{trapezoids added when } s \text{ was inserted last in } S_i \).
- \( k_i(s) = \sum_{\Delta \in T_i} \delta(\Delta, s) \)
- \( E(k_i) = \sum_{s \in S_i} k_i(s) P(s) = \frac{1}{i} \sum_{s \in S_i} k_i(s) = \frac{1}{i} \sum_{s \in S_i} \sum_{\Delta \in T_i} \delta(\Delta, s) \)
- \( = \frac{1}{i} \sum_{\Delta \in T_i} \sum_{s \in S_i} \delta(\Delta, s) \)
- How many segments does \( \Delta \) depend on? At most 4.
- Also, \( T_i \) has \( O(i) \) trapezoids (by Euler’s formula).
- \( E(k_i) = \frac{1}{i} \sum_{\Delta \in T_i} \sum_{s \in S_i} \delta(\Delta, s) = \frac{1}{i} \sum_{\Delta \in T_i} 4 = \frac{1}{i} 4|T_i| = \frac{1}{i} O(i) = O(1) \)
Point Location

- Build a point location data structure; a DAG, similar to Kirkpatrick’s
- DAG has two types of internal nodes:
  - $x$-node (circle): contains the $x$-coordinate of a segment endpoint.
  - $y$-node (hexagon): pointer to a segment
- The DAG has one leaf for each trapezoid.

- Children of $x$-node: Space to the left and right of $x$-coordinate
- Children of $y$-node: Space above and below the segment
- $y$-node is only searched when the query’s $x$-coordinate is within the segment’s span.
- ⇒ Encodes trapezoidal decomposition and enables point location during construction.
Construction

• Incremental construction during trapezoidal map construction.
• When a segment $s$ is added, modify the DAG.
  • Some leaves will be replaced by new subtrees.
• Each old trapezoid will overlap $O(1)$ new trapezoids.
• Each trapezoid appears exactly once as a leaf.

• Changes are highly local.
• If $s$ passes entirely through trapezoid $t$, then $t$ is replaced with two new trapezoids $t'$ and $t''$.
  • Add new $y$-node as parent of $t'$ and $t''$, in order to facilitate search later.
• If an endpoint of $s$ lies in trapezoid $t$, then add an $x$-node to decide left/right and a $y$-node for the segment.
Inserting a Segment

• Insert segment $s_3$. 
Analysis

• **Space:** Expected $O(n)$
  - Size of data structure = number of trapezoids = $O(n)$ in expectation, since an expected $O(1)$ trapezoids are created during segment insertion

• **Query time:** Expected $O(\log n)$

• **Construction time:** Expected $O(n \log n)$ follows from query time

• **Proof** that the query time is expected $O(\log n)$:
  - Fix a query point $Q$.
  - Consider how $Q$ moves through the trapezoidal map as it is being constructed as new segments are inserted.
  - Search complexity = number of trapezoids encountered by $Q$
Query Time

- Let $\Delta_i$ be the trapezoid containing $Q$ after the insertion of $i$th segment.
- If $\Delta_i = \Delta_{i-1}$ then the insertion does not affect $Q$’s trapezoid (E.g., $Q \in B$).
- If $\Delta_i \neq \Delta_{i-1}$ then the insertion deleted $Q$’s trapezoid, and $Q$ needs to be located among the at most 4 new trapezoids.

- $Q$ could fall 3 levels in the DAG.
Query Time

• Let \( X_i \) be the \# nodes on path created in iteration \( i \), and let \( P_i \) be the probability that there exists a node in iteration \( i \), i.e., \( \Delta_i \neq \Delta_{i-1} \).

• The expected search path length is \( E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i) \leq \sum_{i=1}^{n} 3 P_i \) by lin. of expectation and since \( Q \) can drop at most 3 levels.

• **Claim:** \( P_i \leq \frac{4}{i} \).
  - Backwards analysis: Consider deleting segments, instead of inserting.
  - Trapezoid \( \Delta_i \) depends on \( \leq 4 \) segments. The probability that the \( i \)th segment is one of these 4 is \( \leq \frac{4}{i} \).

• The expected search path length is at most

\[
\sum_{i=1}^{n} 3 P_i = \sum_{i=1}^{n} 3 \frac{4}{i} = 12 \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)
\]

Harmonic number