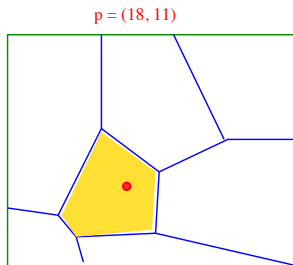


Point Location

- Preprocess a planar, polygonal subdivision for point location queries.



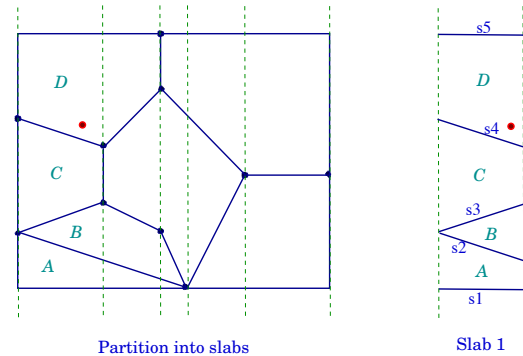
- Input is a subdivision S of complexity n , say, number of edges.
- Build a data structure on S so that for a query point $p = (x, y)$, we can find the face containing p fast.
- Important metrics: space and query complexity.

Subhash Suri

UC Santa Barbara

The Slab Method

- Draw a vertical line through each vertex. This decomposes the plane into slabs.
- In each slab, the vertical order of line segments remains constant.



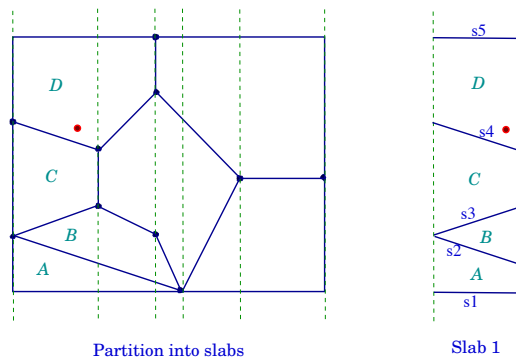
- If we know which slab $p = (x, y)$ lies, we can perform a binary search, using the sorted order of segments.

Subhash Suri

UC Santa Barbara

The Slab Method

- To find which slab contains p , we perform a binary search on x , among slab boundaries.
- A second binary search in the slab determines the face containing p .



- Thus, the search complexity is $O(\log n)$.
- But the space complexity is $\Theta(n^2)$.

Subhash Suri

UC Santa Barbara

Optimal Schemes

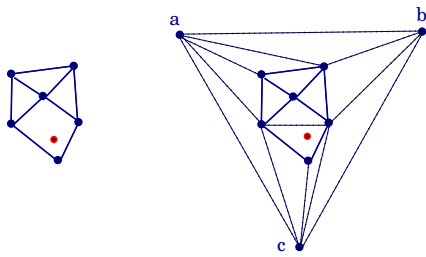
- There are other schemes (kd -tree, quad-trees) that can perform point location reasonably well, they lack theoretical guarantees. Most have very bad worst-case performance.
- Finding an optimal scheme was challenging. Several schemes were developed in 70's that did either $O(\log n)$ query, but with $O(n \log n)$ space, or $O(\log^2 n)$ query with $O(n)$ space.
- Today, we will discuss an elegant and simple method that achieved optimality, $O(\log n)$ time and $O(n)$ space [D. Kirkpatrick '83].
- Kirkpatrick's scheme however involves large constant factors, which make it less attractive in practice.
 - Later we will discuss the use of persistent data structures to obtain a practical and almost optimal solution.

Subhash Suri

UC Santa Barbara

Kirkpatrick's Algorithm

- Start with the assumption that planar subdivision is a **triangulation**.
- If not, triangulate each face, and label each triangular face with the same label as the original containing face.
- If the outer face is not a triangle, compute the convex hull, and triangulate the pockets between the subdivision and CH.
- Now put a large triangle abc around the subdivision, and triangulate the space between the two.

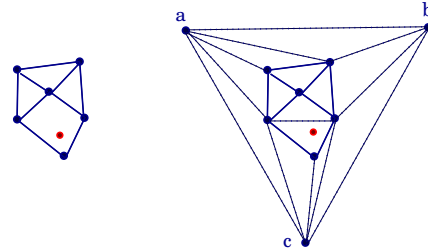


Subhash Suri

UC Santa Barbara

Modifying Subdivision

- By Euler's formula, the final size of this triangulated subdivision is still $O(n)$.
- This transformation from S to triangulation can be performed in $O(n \log n)$ time.



- If we can find the triangle containing p , we will know the original subdivision face containing p .

Subhash Suri

UC Santa Barbara

Hierarchical Method

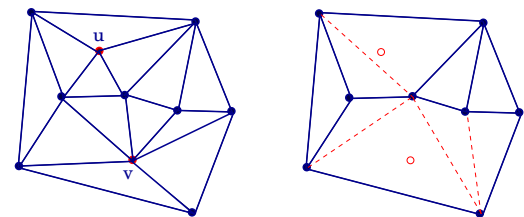
- Kirkpatrick's method is hierarchical: produce a sequence of increasingly coarser triangulations, so that the last one has $O(1)$ size.
- Sequence of triangulations T_0, T_1, \dots, T_k , with following properties:
 1. T_0 is the initial triangulation, and T_k is just the outer triangle abc .
 2. k is $O(\log n)$.
 3. Each triangle in T_{i+1} overlaps $O(1)$ triangles of T_i .
- Let us first discuss how to construct this sequence of triangulations.

Subhash Suri

UC Santa Barbara

Building the Sequence

- Main idea is to delete some vertices of T_i .
- Their deletion creates **holes**, which we re-triangulate.



Vertex deletion and re-triangulation

- We want to go from $O(n)$ size subdivision T_0 to $O(1)$ size subdivision T_k in $O(\log n)$ steps.
- Thus, we need to delete a **constant fraction** of vertices from T_i .
- A critical condition is to ensure each new triangle in T_{i+1} overlaps with $O(1)$ triangles of T_i .

Subhash Suri

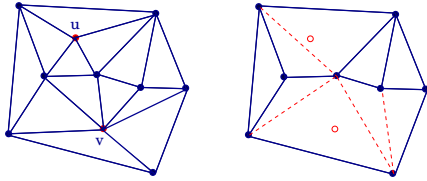
UC Santa Barbara

Independent Sets

- Suppose we want to go from T_i to T_{i+1} , by deleting some points.
- Kirkpatrick's choice of points to be deleted had the following two properties:

[Constant Degree] Each deletion candidate has $O(1)$ degree in graph T_i .

- If p has degree d , then deleting p leaves a hole that can be filled with $d - 2$ triangles.
- When we re-triangulate the hole, each new triangle can overlap at most d original triangles in T_i .

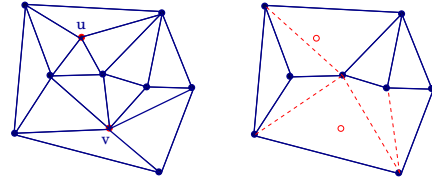


Vertex deletion and re-triangulation

Independent Sets

[Independent Sets] No two deletion candidates are adjacent.

- This makes re-triangulation easier; each hole handled independently.



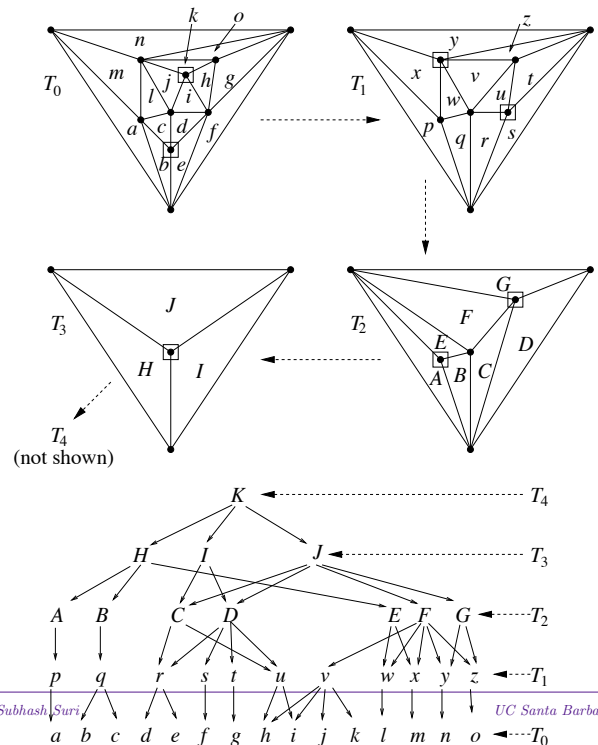
Vertex deletion and re-triangulation

I.S. 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. The set can be found in $O(n)$ time.

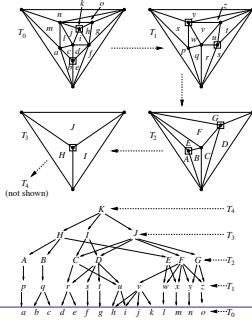
- We prove this later. Let's use this now to build the triangle hierarchy, and show how to perform point location.
- Start with T_0 . Select an ind set S_0 of size $n/18$, with max degree 8. Never pick a, b, c , the outer triangle's vertices.
- Remove the vertices of S_0 , and re-triangulate the holes.
- Label the new triangulation T_1 . It has at most $\frac{17}{18}n$ vertices. Recursively build the hierarchy, until T_k is reduced to abc .
- The number of vertices drops by $17/18$ each time, so the depth of hierarchy is $k = \log_{18/17} n \approx 12 \log n$

Illustration



The Data Structure

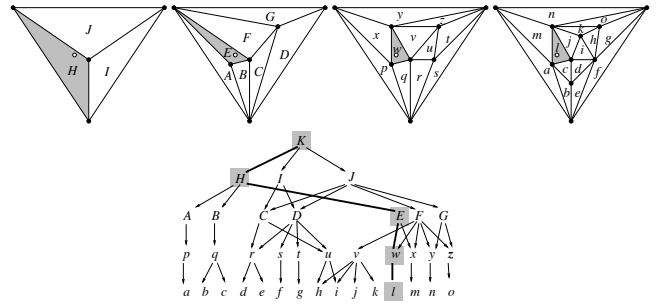
- Modeled as a DAG: the root corresponds to single triangle T_k .
- The nodes at next level are triangles of T_{k-1} .
- Each node for a triangle in T_{i+1} has pointers to all triangles of T_i that it overlaps.
- To locate a point p , start at the root. If p outside T_k , we are done (exterior face). Otherwise, set $t = T_k$, as the triangle at current level containing p .



Subhash Suri

UC Santa Barbara

The Search

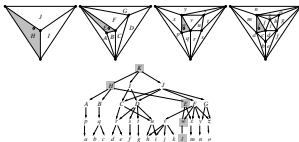


- Check each triangle of T_{k-1} that overlaps with t —at most 6 such triangles. Update t , and descend the structure until we reach T_0 .
- Output t .

Subhash Suri

UC Santa Barbara

Analysis



- Search time is $O(\log n)$ —there are $O(\log n)$ levels, and it takes $O(1)$ time to move from level i to level $i - 1$.
- Space complexity requires summing up the sizes of all the triangulations.
- Since each triangulation is a planar graph, it is sufficient to count the number of vertices.
- The total number of vertices in all triangulations is

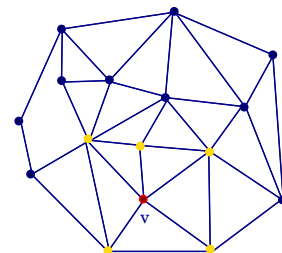
$$n \left(1 + (17/18) + (17/18)^2 + (17/18)^3 + \dots \right) \leq 18n.$$
- Kirkpatrick structure has $O(n)$ space and $O(\log n)$ query time.

Subhash Suri

UC Santa Barbara

Finding I.S.

- We describe an algorithm for finding the independent set with desired properties.
- Mark all nodes of degree ≥ 9 .
- While there is an unmarked node, do
 - Choose an unmarked node v .
 - Add v to IS.
 - Mark v and all its neighbors.
- Algorithm can be implemented in $O(n)$ time—keep unmarked vertices in list, and representing T so that neighbors can be found in $O(1)$ time.



Subhash Suri

UC Santa Barbara

I.S. Analysis

- Existence of large size, low degree IS follows from Euler's formula for planar graphs.

- A triangulated planar graph on n vertices has $e = 3n - 6$ edges.

- Summing over the vertex degrees, we get

$$\sum_v \deg(v) = 2e = 6n - 12 < 6n.$$

- We now claim that at least $n/2$ vertices have degree ≤ 8 .

- Suppose otherwise. Then $n/2$ vertices all have degree ≥ 9 . The remaining have degree at least 3. (Why?)

- Thus, the sum of degrees will be at least $9\frac{n}{2} + 3\frac{n}{2} = 6n$, which contradicts the degree bound above.

- So, in the beginning, at least $n/2$ nodes are unmarked. Each chosen v marks at most 8 other nodes (total 9 counting itself.)

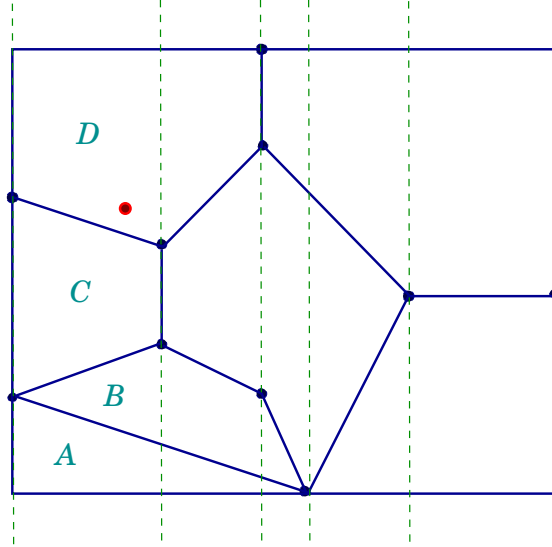
- Thus, the node selection step can be repeated at least $n/18$ times.

- So, there is a I.S. of size $\geq n/18$, where each node has degree ≤ 8 .

1 Point Location using Persistent Search Trees

In this section we describe another approach to the point-location problem, based on a fully-functional, or “persistent”, representation of sets. We obtain an $O(\log n)$ query time and $O(n \log n)$ space solution. So it is not optimal in terms of space. This can be made optimal by a more sophisticated way to make data structures persistent.

Let’s go back to the approach of dividing the polygonal subdivision into slabs. With the representation described earlier, doing a point location query took only $O(\log n)$ time. The problem was that the data structure took too much space.



Partition into slabs

Let’s examine this problem a little more closely. The reason that the space is potentially quadratic in n is that there is the possibility that a long horizontal segment can be divided up into many pieces, one for each slab. For example, the segment separating regions A and B is divided into three parts.

If there were some way that we could avoid having to store that redundantly in multiple slabs, we have a chance of controlling the space blow-up. This is the approach we take here.

With this in mind, let’s examine the difference between consecutive slabs. Call the two slabs S_i and S_{i+1} . The differences between the two slabs occur when there is a vertex v of the subdivision along their boundary. To convert S_i into S_{i+1} we delete from S_i the segments incident on v from the left, and we insert into S_i the segments incident on v from the right. We do this for all the vertices on the boundary between the two slabs.

The total number of insertions and deletions that occur in the entire diagram is twice the number of segments, or $O(n)$. It turns out that using functional programming we can incur a space cost of $O(\log n)$ per insertion or deletion.

1.1 Fully Functional Ordered Sets

A slab is represented by a set of non-vertical segments. For each non-vertical we're going to keep the equation of its line. So for segment i we keep $m_i x + b_i$, where m_i is the slope and b_i is the intercept. This representation will allow us to determine which of two segments is higher (at a particular x value).

A slab is a set of segments. We're going to store this in a balanced binary search tree. It will be a fully functional implementation. This means, for example, that if we insert a segment a into a set S , it returns a new set S' . The original set S remains the same.

Our set data structure will support the following operations:

Insert(S, a): Insert a segment a into a set S . Return the result.

Delete(S, a): Delete a segment a from the set S . Return the result.

Lookup(S, p): Given a point p and a set of segments S , return a segment from S that neighbors the region containing p .

Fully functional implementations of sets based on balanced binary search trees are standard features of functional programming languages such as Haskell, Ocaml, and SML. Furthermore, it's easy to create fully functional implementations in any language. You just have to maintain the discipline of never modifying any fields of a node. Instead you create a new node initialized with the values you want in each field.

Because the sets are stored as a tree, the way lookup works is that it returns the last segment touched when searching down the tree for a segment containing point p .

The space used by Insert and Delete is $O(\log n)$. This happens automatically in a functional implementation of sets using balanced binary search trees. (You can also think of it as copying the path from the root to a node that changes.)

1.2 Building the Data Structure

Start out with an empty slab S_0 . Process the vertices v_0, \dots, v_k in left-to-right order. To process vertex v_i we take slab S_i and delete the segments connecting to it on the left and insert the ones that connect to it on the right. After this, we have slab S_{i+1} . The space used by this is $O(n \log n)$.

As we do this we build an array A , sorted by x , with pointers to the slabs that we constructed in this scan. So given x we can find in $O(\log n)$ time the slab containing x .

We're also going to build a dictionary D which keeps, for each non-vertical segment in the diagram, the name of the region above it and below it.

1.3 Doing Point Location

Given a point $p = (x, y)$ here is how we find the region containing p .

First we find the slab containing p by doing binary search in the array A . Say it's in slab S_i . Now we do a lookup of p in S_i . This gives us a segment s bounding the region containing point p . Now we determine if the point p is above or below s . So we can then lookup in the dictionary D the name of the region containing the point p .

This process is $O(\log n)$ time.

1.4 Achieving $O(n)$ Space

Optimal space can be achieved by using the “fat node” method of making data structures persistent. It’s described in this paper by Sarnak and Tarjan.

www.link.cs.cmu.edu/15859-f07/papers/point-location.pdf

The details are beyond the scope of this course, but feel free to talk to me about how it works.