CS633 Lecture 02
Line Segments Intersection

Jyh-Ming Lien

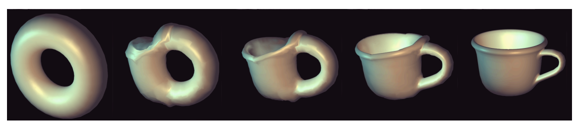
Dept of Computer Science

George Mason University

Based on Chapter 2 of the textbook and Ming Lin's lecture note at UNC

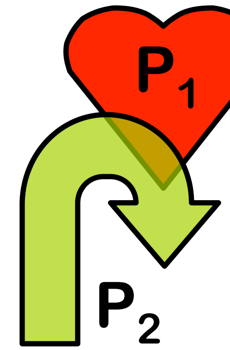
Line Segments Intersection

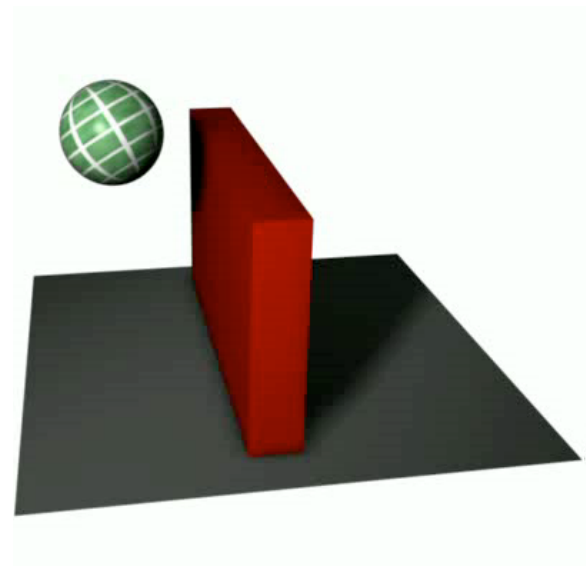
- Driving Applications
  - Geographic information system:
    - the “Map overlay” problems
  - Computer Graphics:
    - Polygon intersection
    - 3D Morphing



  - Modeling:
    - Polygonal Boolean operations

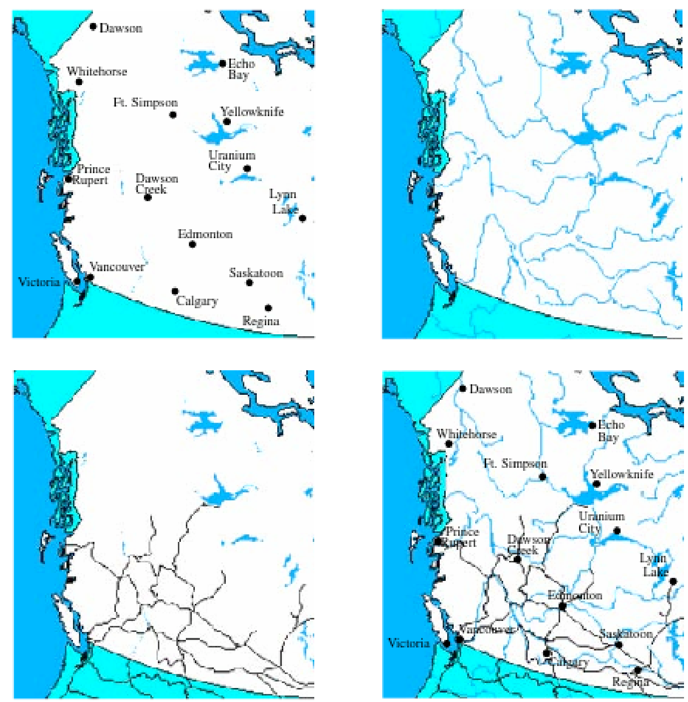
(Constructive Solid Geometry or CSG)





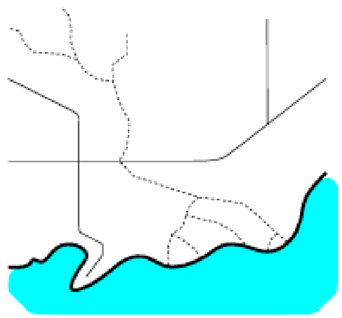
Application 1
Thematic Map Overlay

- GISs split each map into several layers
- Each layer is called a *thematic map*
  - storing one type of information
- Find overlay of several maps to locate interesting junctions
  - Line/curve intersections
  - Region overlapping
  - Point location
  - …



Transform to a Geometric Problem

GIS
Finding the overlay of two maps



- Curves can be approximated by small (line) segments
- Each thematic map can be viewed as a collection of line segments

Computational Geometry
computing all intersection points between the line segments of two sets

To simplify further

Make 2 sets into 1. But, how do we identify the real intersections?

Line Segments Intersection

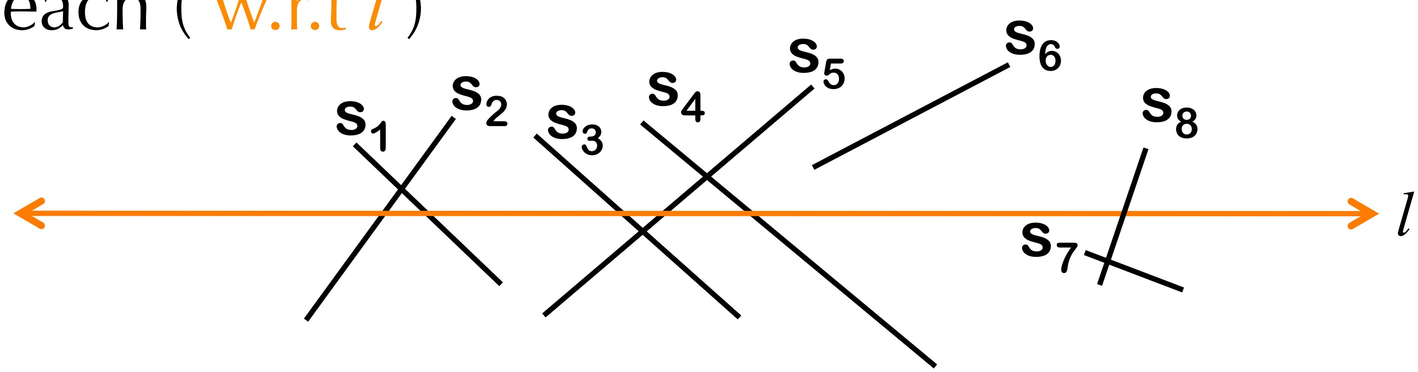
- Problem: Given a set of line segments
- Output: Intersections and for each intersection output the intersecting segments.

Problem Analysis

- Brute Force Approach: O(n^2)
  - Is this the lower bound of the problem?
  - Is this good for our problem? Why?
    - Even there are no intersections, we will spend O(n^2) time
- **Desiderata**: output (intersection) sensitive
- **Observation**: Segments that are close together are the candidates for intersection
  - How do you determine two segments that are close or far away???
    - Can the distance of two segments tell you anything?

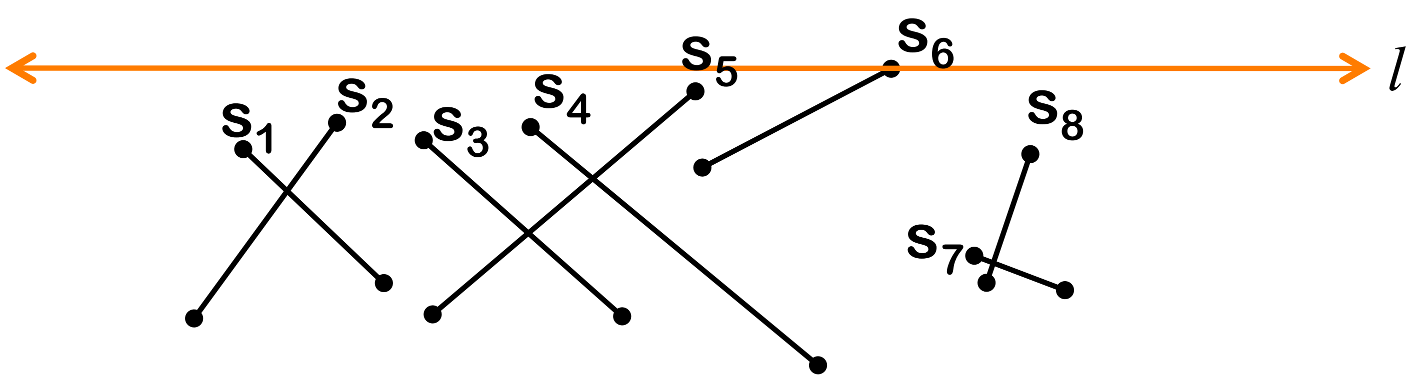
Closeness of Segments

- Draw a line l (horizontal line) find intersections between segments and l,
- Order segments from left to right according to the intersecting point on l
- Now, we know which segments are close to each (w.r.t l)



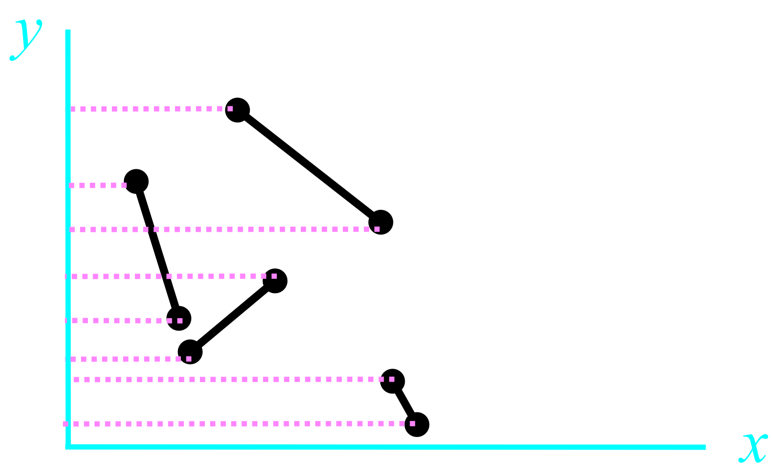
Plane Sweep

- Now if we move the line up and down
  - we should reveal the relationships (closeness) of the line segment across the plane
    - How do you compute the intersections between l and segments efficiently?
    - Do you have to compute the intersections all the time when l sweeps?



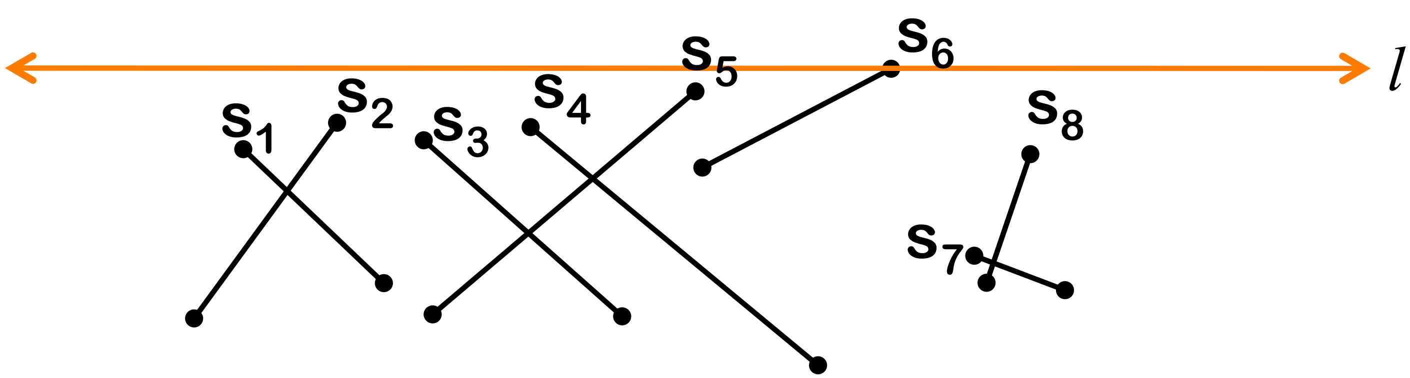
Plane Sweep

- How do you compute the intersections between l and segments efficiently?
  - Project the interval to Y-axis and build a data structure (interval tree) (?)

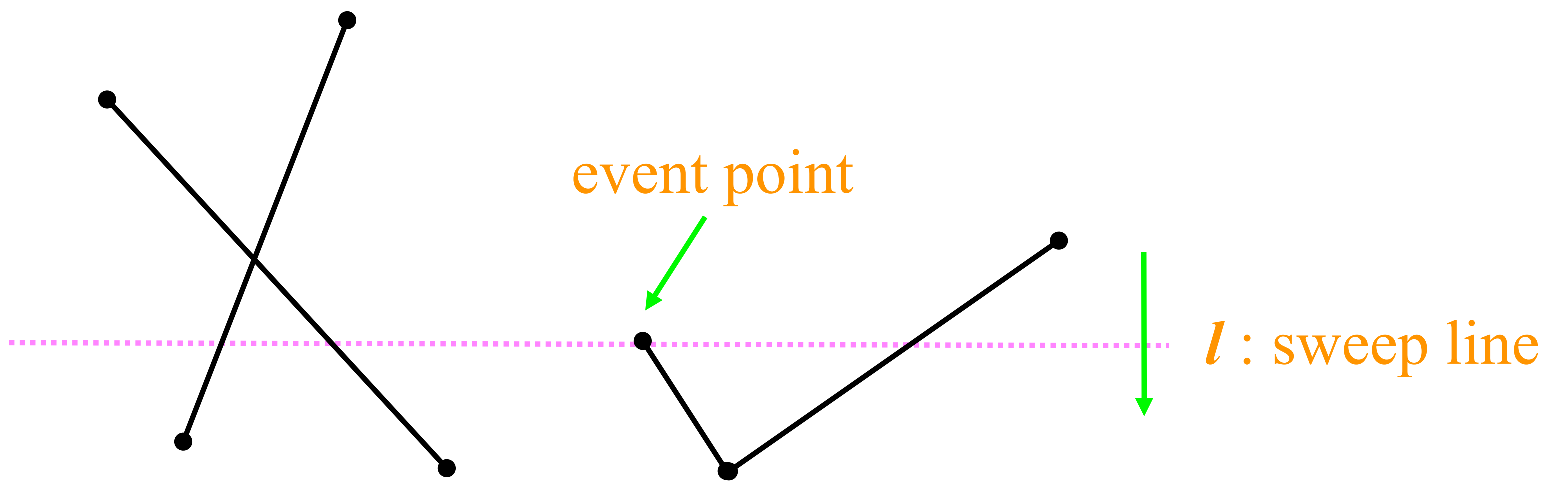


Plane Sweep

- Do you have to compute the intersections all the time when l when move up and down?
  - No!
  - The segment orders only change at the events:
    - End points
    - intersections



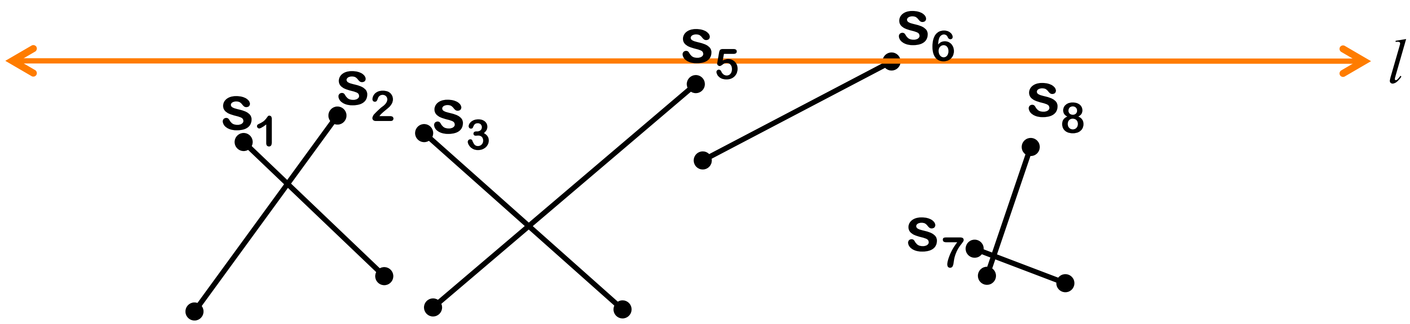
Plane Sweep: Summary



- Status of *l*: the set of segments intersecting *l*
  - Maintain a data structure T so the intersecting segments are sorted from left to right
- Event points: where updates are required

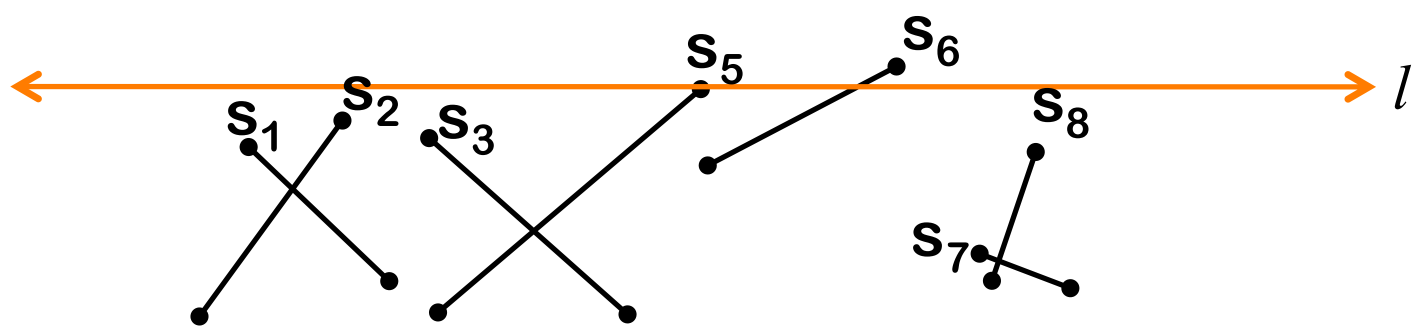
Plane Sweep

- Status of l : (insert S_6 to T)
  - S_6



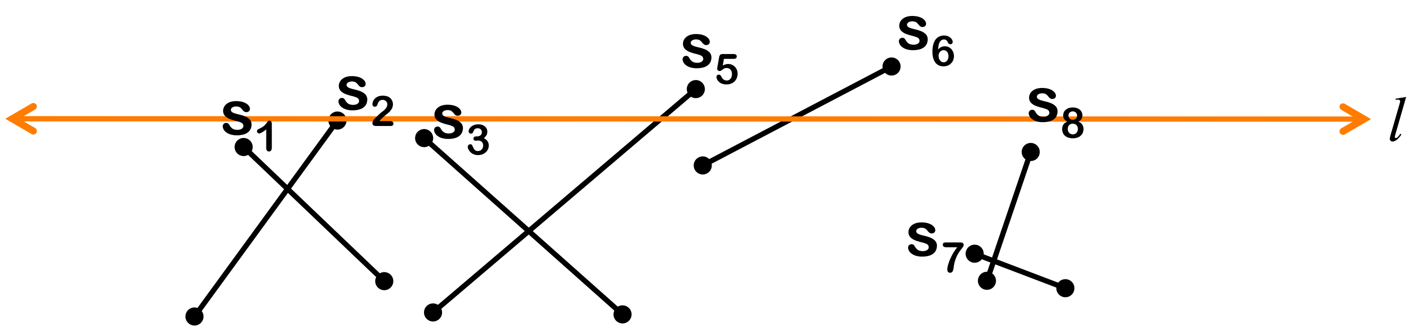
Plane Sweep

- Status of l : (insert S_5 to T)
  - S_5S_6



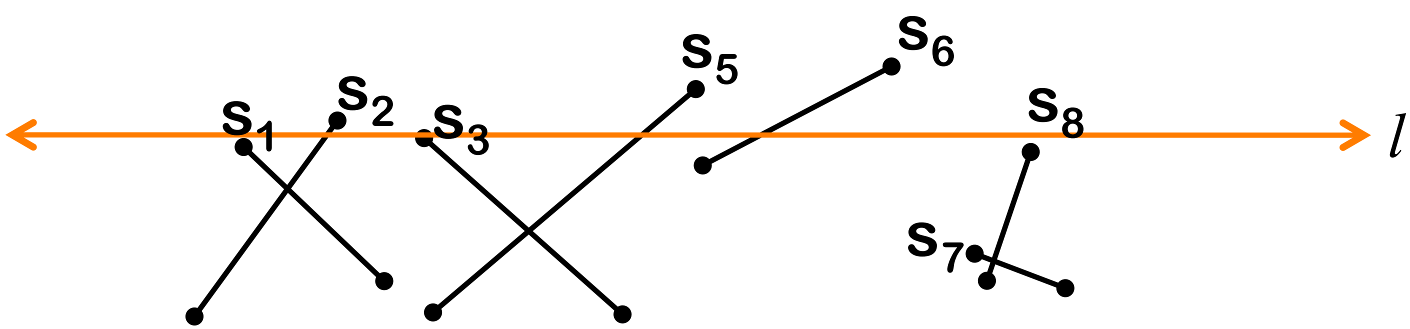
Plane Sweep

- Status of l : (insert S_2 to T)
  - $S_2S_5S_6$



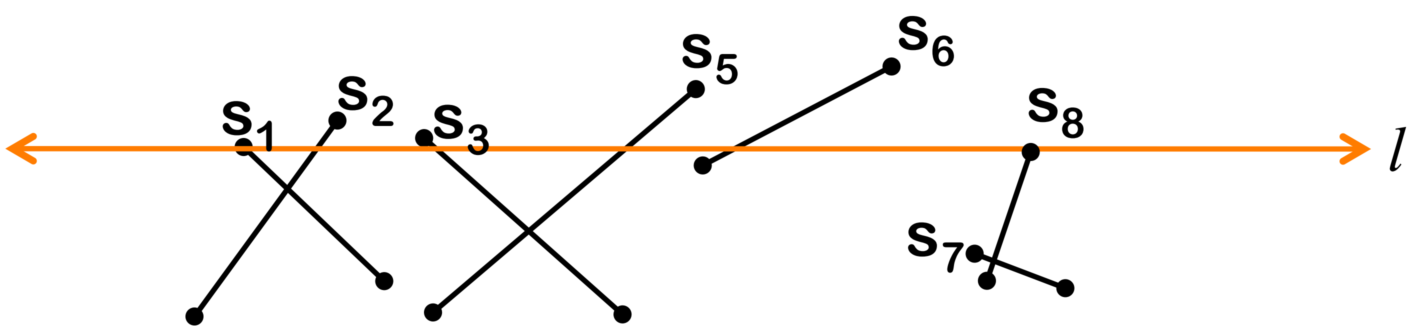
Plane Sweep

- Status of l : (insert S_3 to T)
  - $S_2S_3S_5S_6$



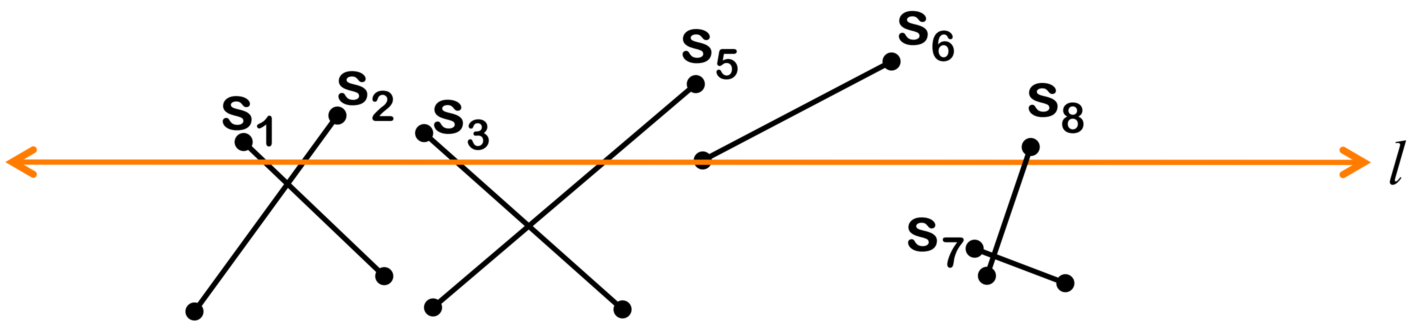
Plane Sweep

- Status of l : (insert S_1 and S_8 to T)
  - $S_1S_2S_3S_5S_6S_8$



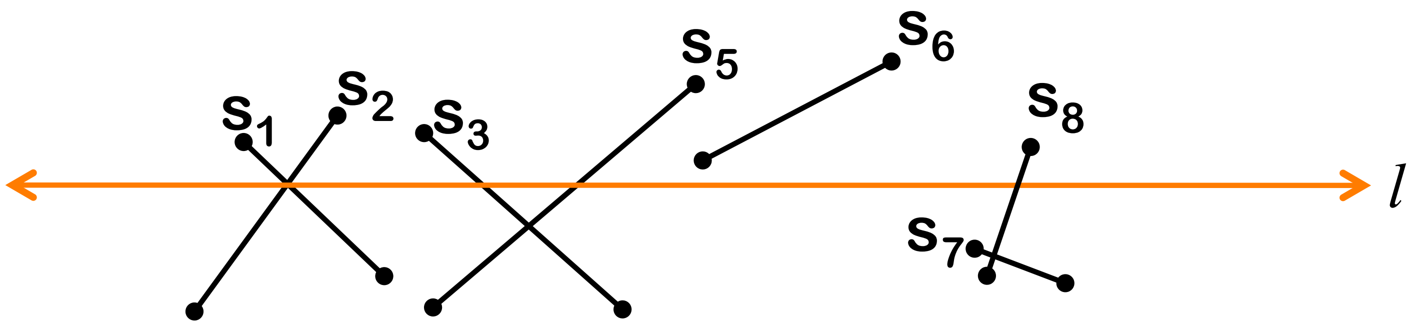
Plane Sweep

- Status of l : (delete S_6 to T)
  - $S_1S_2S_3S_5S_8$



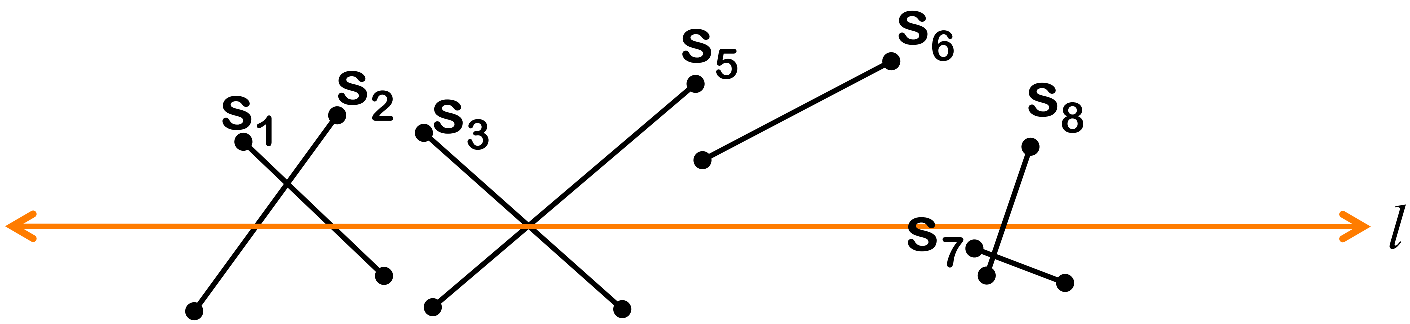
Plane Sweep

- Status of l : (swap S_1 and S_2 in T)
  - $S_2S_1S_3S_5S_8$



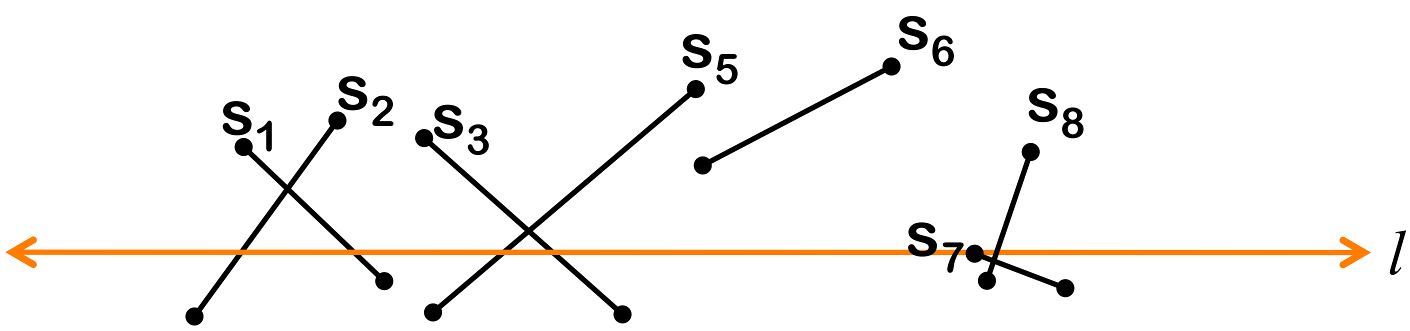
Plane Sweep

- Status of l : (swap S_3 and S_5 in T)
  - $S_2S_1S_5S_3S_8$



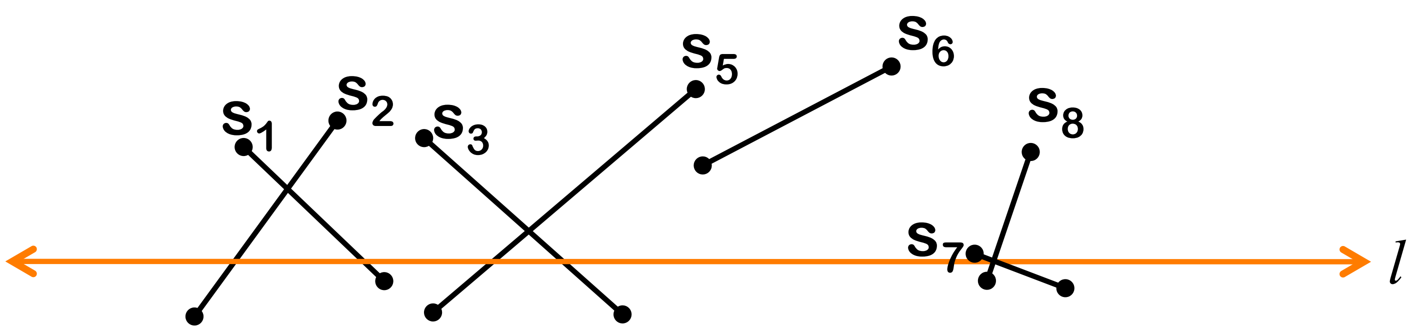
Plane Sweep

- Status of l : (add S_7 to T)
  - $S_2S_1S_5S_3S_7S_8$



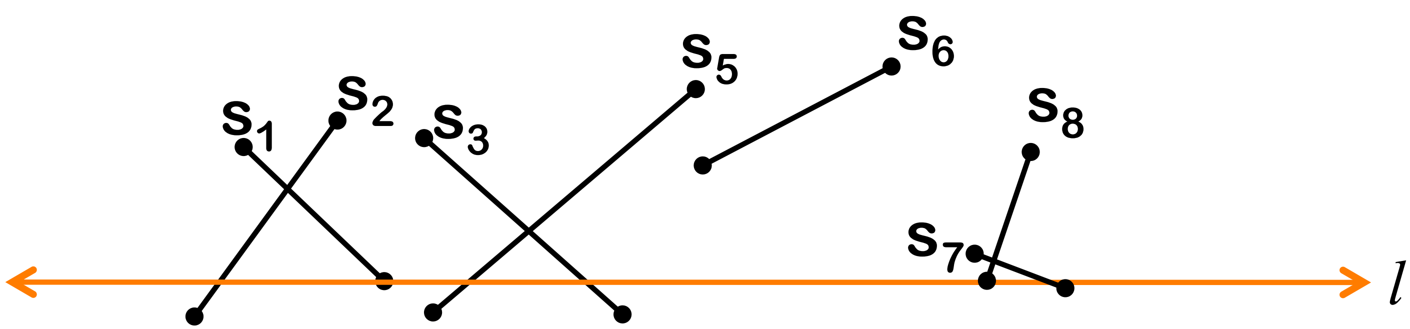
Plane Sweep

- Status of l : (swap S_7 S_8 in T)
  - $S_2S_1S_5S_3S_8S_7$



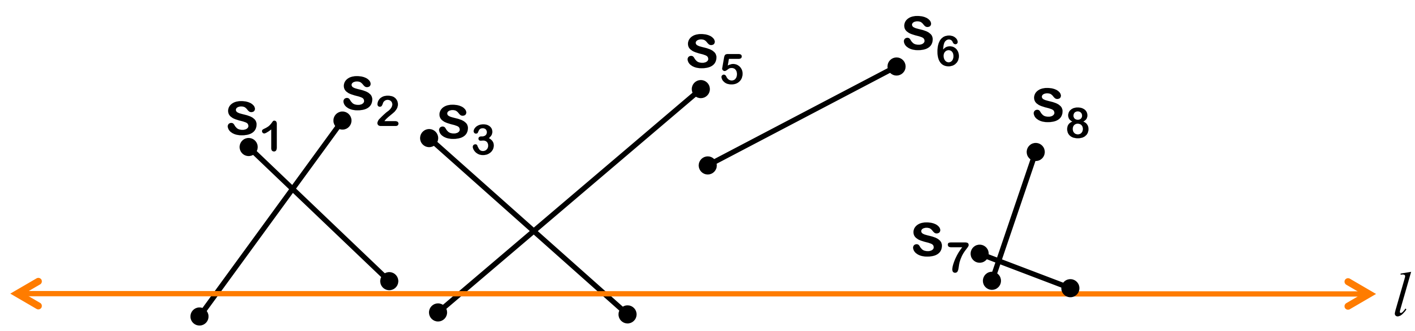
Plane Sweep

- Status of l : (delete S_1 , S_8 from T)
  - $S_2S_5S_3S_7$



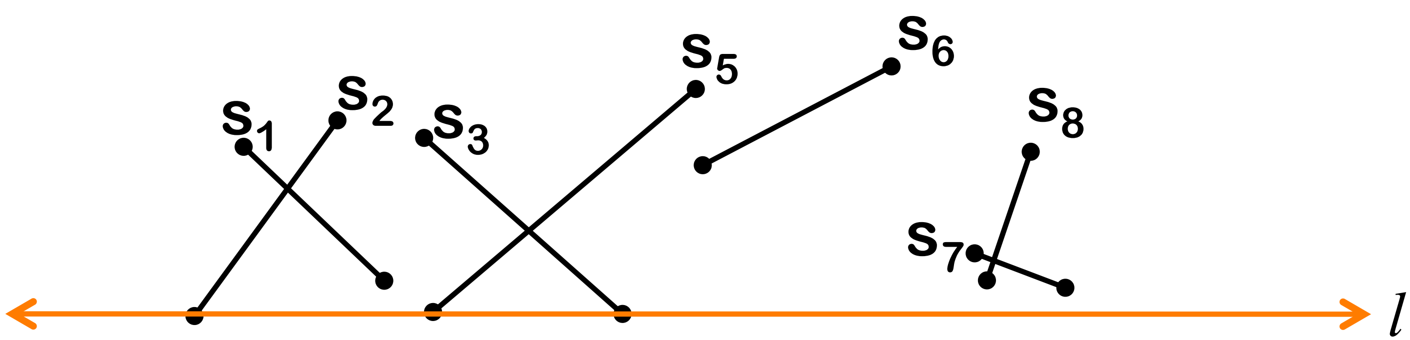
Plane Sweep

- Status of l : (delete S_7 from T)
  - $S_2S_5S_3$



Plane Sweep

- Status of l : (delete S_2 ,S_5 ,S_3 from T)
  - ϕ



Plane Sweep Algorithm
Sketch 1

- Create an event queue Q and add the end points of the segments from top to bottom to the event queue

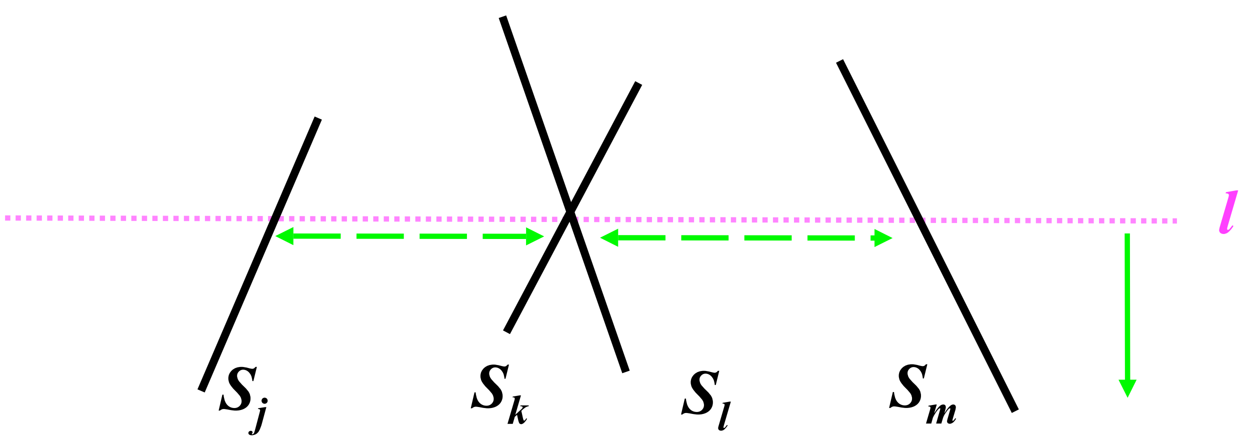
- Create a horizontal sweep line *l* and maintain a sorted list T

- Repeat until no events in Q
  – e ←Q.pop()
  – Place *l* at e
  – Find the segments intersecting the sweep line *l* and store them in T
  – For each pair of adjacent segments in T
    - Check intersection
    - Add intersection to Q

Plane Sweep Algorithm

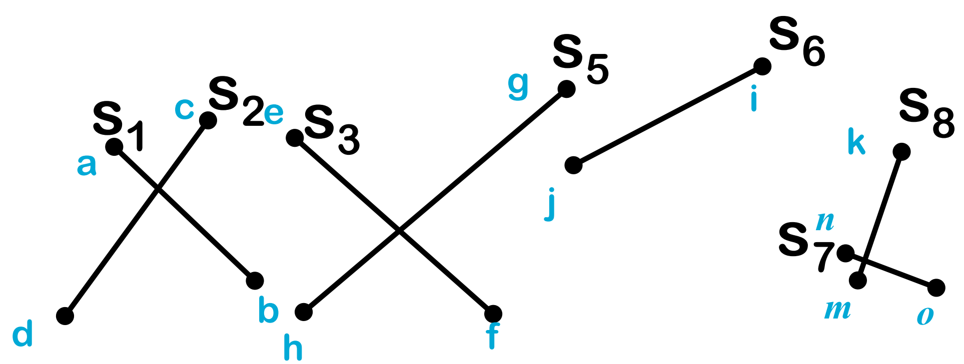
To include the idea of being close in the horizontal direction, only test segments that are adjacent in the horizontal direction --

- Only test each with ones to its left and right
- New “status”: *ordered* sequence of segments



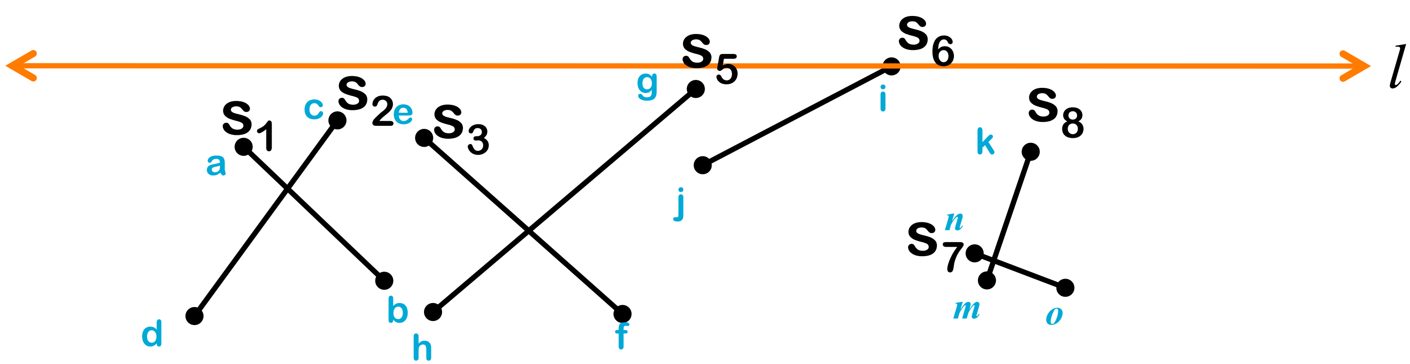
Plane Sweep

- Status of l : (insert S_6 to T)
  - ϕ



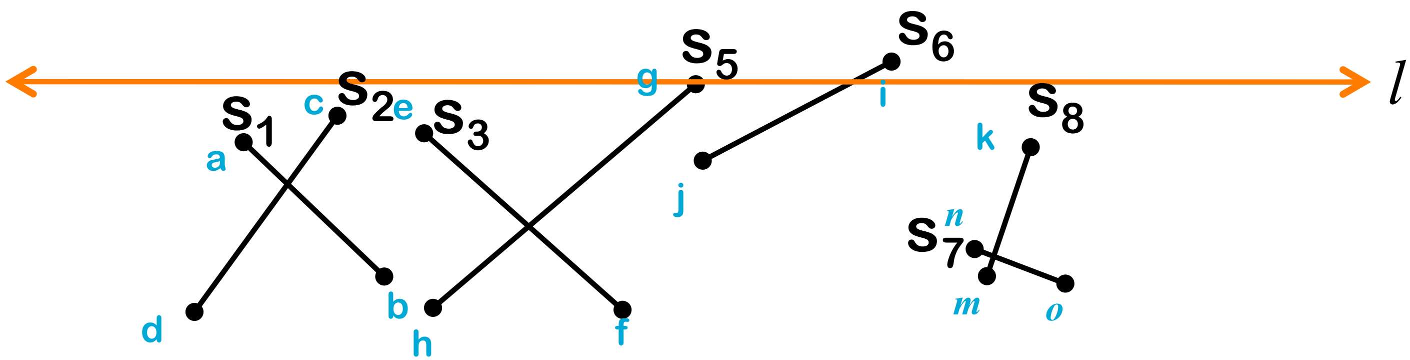
Plane Sweep

- Status of l : (insert S_6 to T)
  - S_6



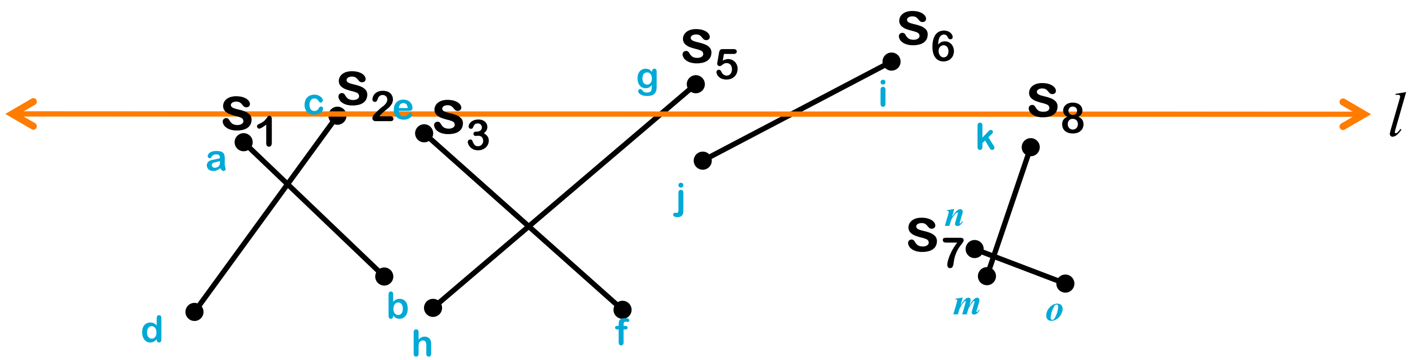
Plane Sweep

- Status of l : (insert S_5 to T)
  - S_5S_6



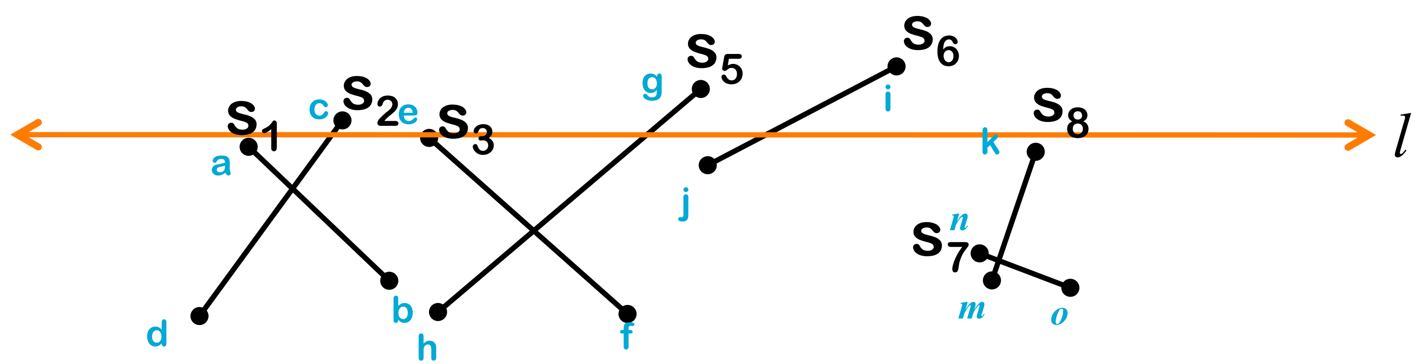
Plane Sweep

- Status of l : (insert S_2 to T)
  - $S_2S_5S_6$



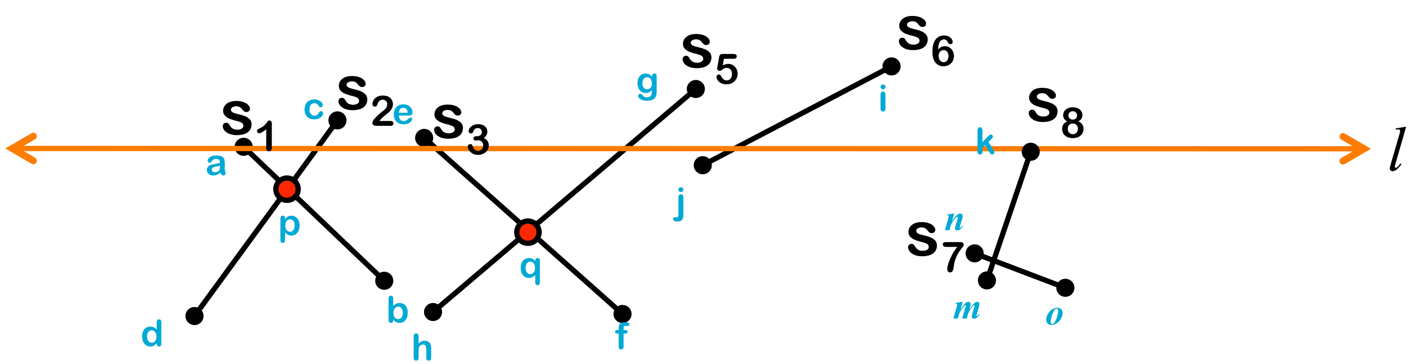
Plane Sweep

- Status of l : (insert S_3 to T)
  - $S_2S_3S_5S_6$



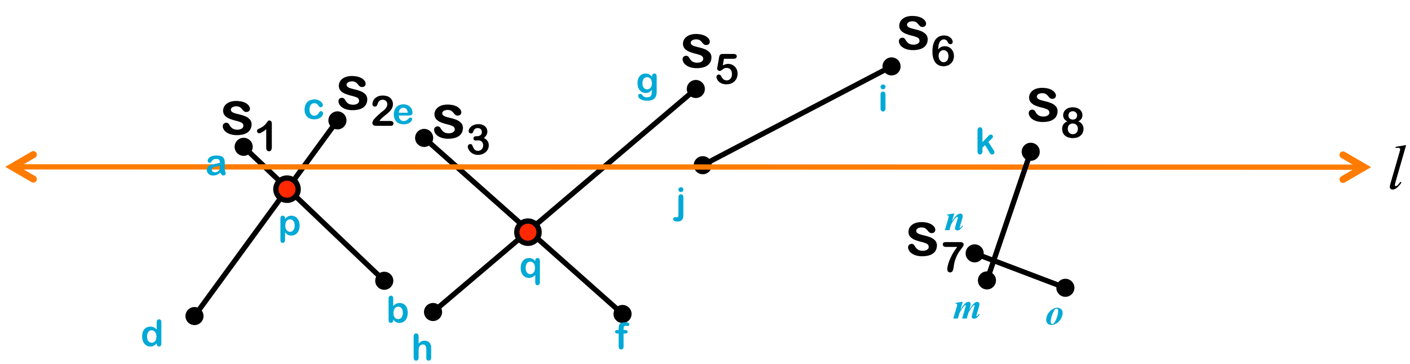
Plane Sweep

- Status of l : (insert S_1 and S_8 to T)
  - $S_1S_2S_3S_5S_6S_8$



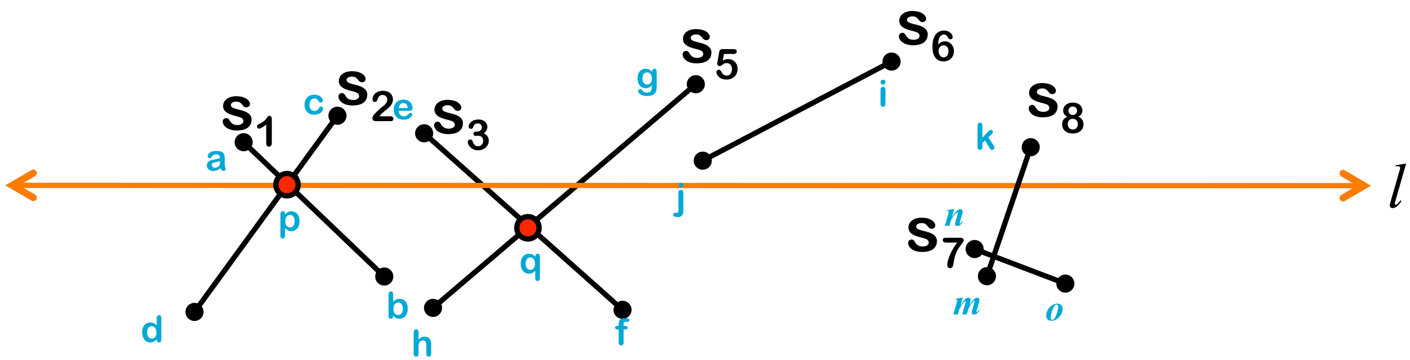
Plane Sweep

- Status of l : (delete S_6 to T)
  - $S_1S_2S_3S_5S_8$



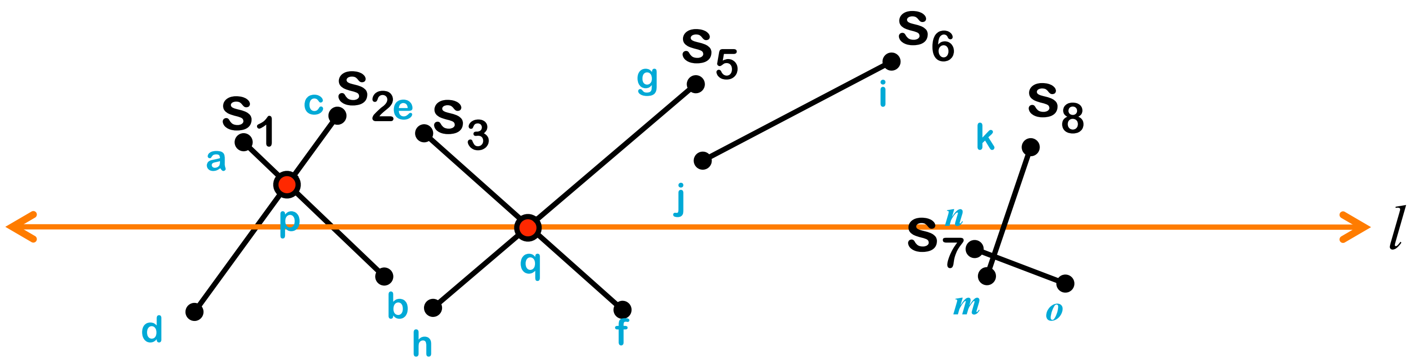
Plane Sweep

- Status of l : (swap S_1 and S_2 in T)
  - $S_2S_1S_3S_5S_8$



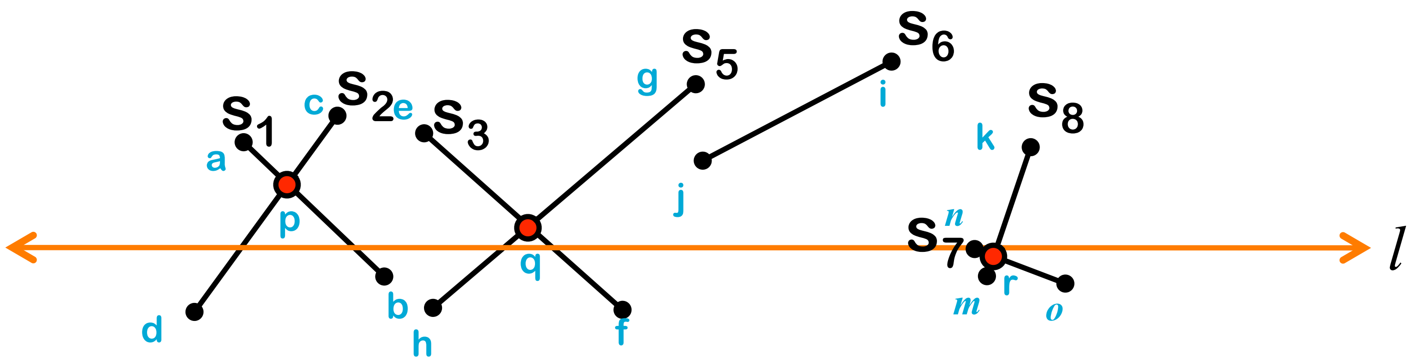
Plane Sweep

- Status of l : (swap S_3 and S_5 in T)
  - $S_2S_1S_5S_3S_8$



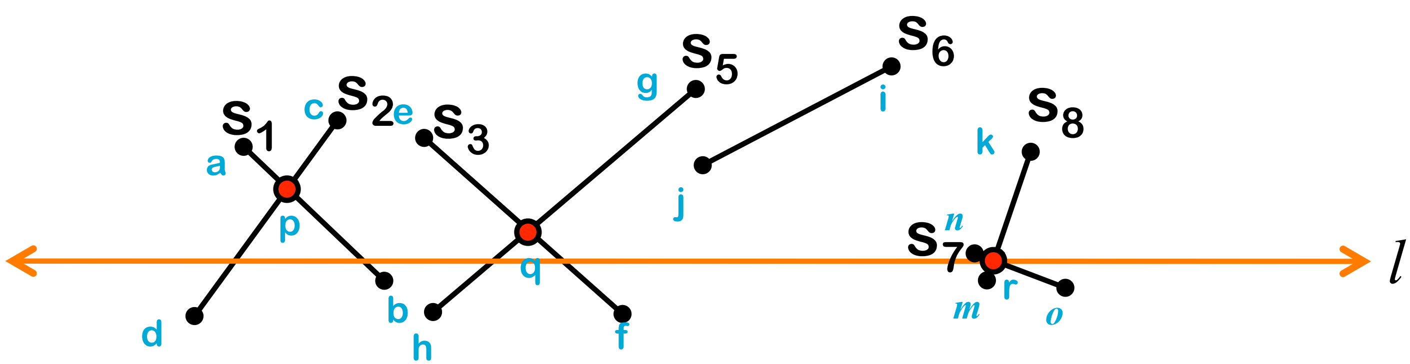
Plane Sweep

- Status of l : (add S_7 to T)
  - $S_2S_1S_5S_3S_7S_8$



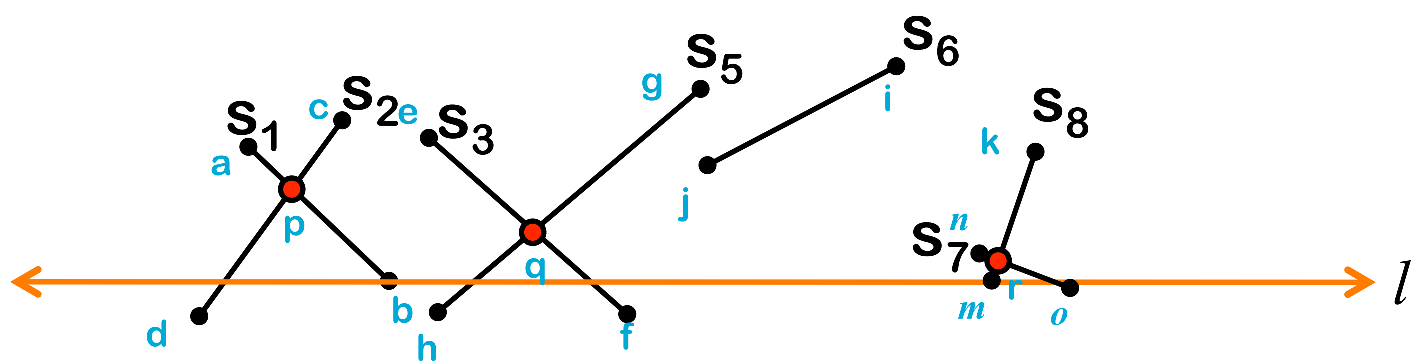
Plane Sweep

- Status of l : (swap S_7 $S_{8\text{ in}}$ T)
  - $S_2S_1S_5S_3S_8S_7$



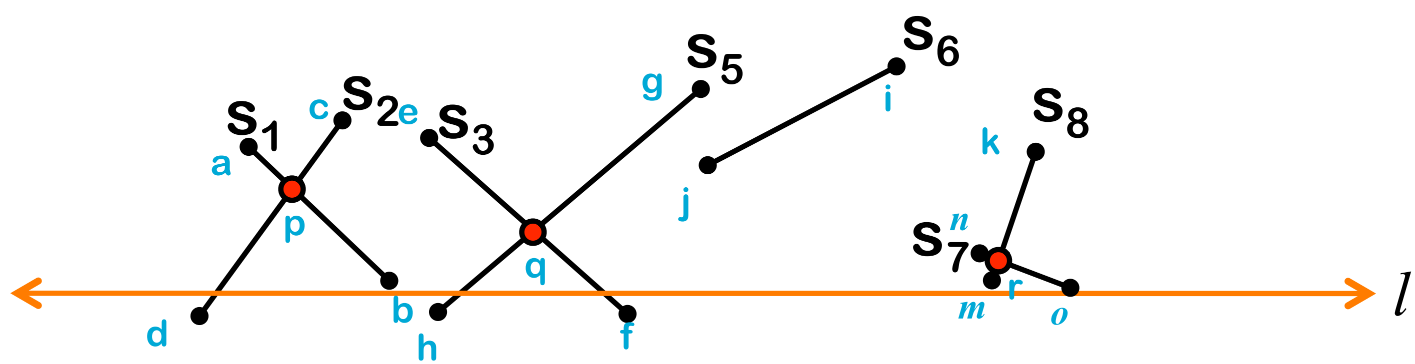
Plane Sweep

- Status of l : (delete S_1 , S_8 from T)
  - $S_2S_5S_3S_7$



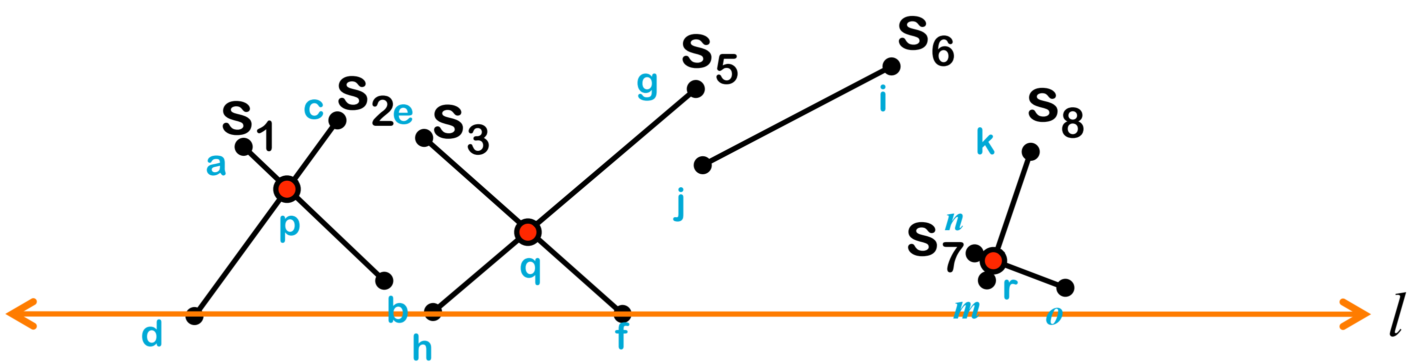
Plane Sweep

- Status of l : (delete S_7 from T)
  - $S_2S_5S_3$



Plane Sweep

- Status of l : (delete S_2 ,S_5 ,S_3 from T)
  - ϕ



Plane Sweep Algorithm
Sketch 1

What's the problem of this algorithm?

- Create an event queue Q and add end points of the segments from top to bottom to the event queue
- Create a horizontal sweep line *l* and maintain a sorted list T
- Repeat until no events in Q
  - e ←Q.pop()
  - Place *l* at e
  - Find the segments intersecting the sweep line *l* and store them in T
  - For each pair of adjacent segments in T
    - Check intersection
    - Add intersection to Q

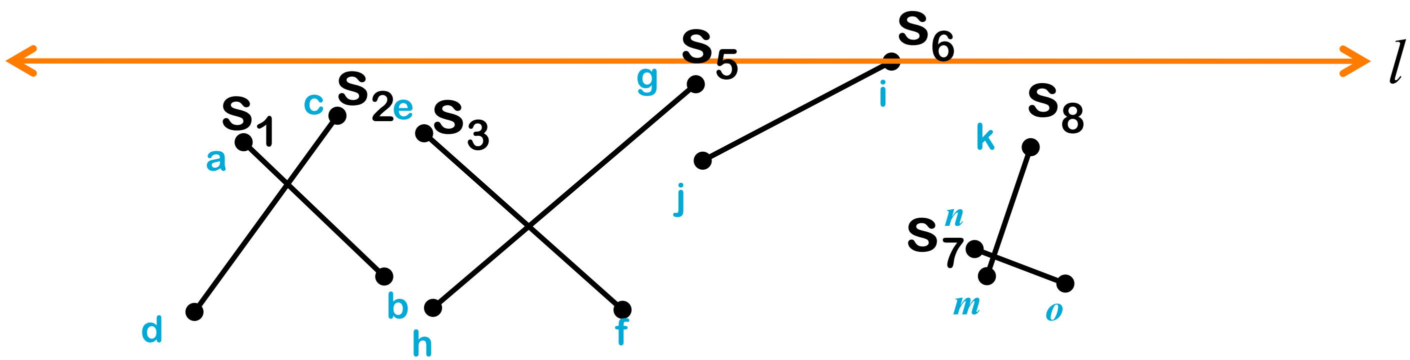
No need to check all pairs!

Plane Sweep Algorithm
Sketch 2

- Create an event queue Q and add end points of the segments from top to bottom to the event queue
- Create a horizontal sweep line *l* and maintain a sorted list T
- Repeat until no events in Q
  - e ←Q.pop()
  - Place *l* at e
  - If e is an upper point of a segment *s*
    - Add *s* to T
    - Check intersection between *s* and *s*′ left and right segments in T
  - If e is a lower point of a segment *s*
    - Remove *s* from T
    - Check intersection between *s*′ left and right segments in T
  - If e is an intersecting point of a set of segments *S*
    - Reorder *S* in T accordingly
    - Check the leftmost segment with the segment on its left
    - Check the rightmost segment with the segment on its right

Plane Sweep

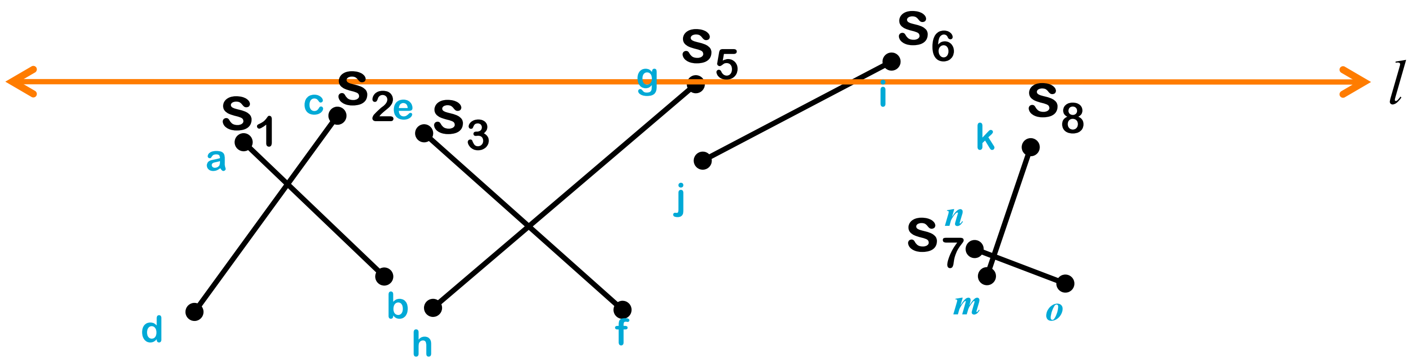
- Status of l : (insert S_6 to T)
  - S_6



Plane Sweep

- Status of l : (insert S_5 to T)
  - S_5S_6

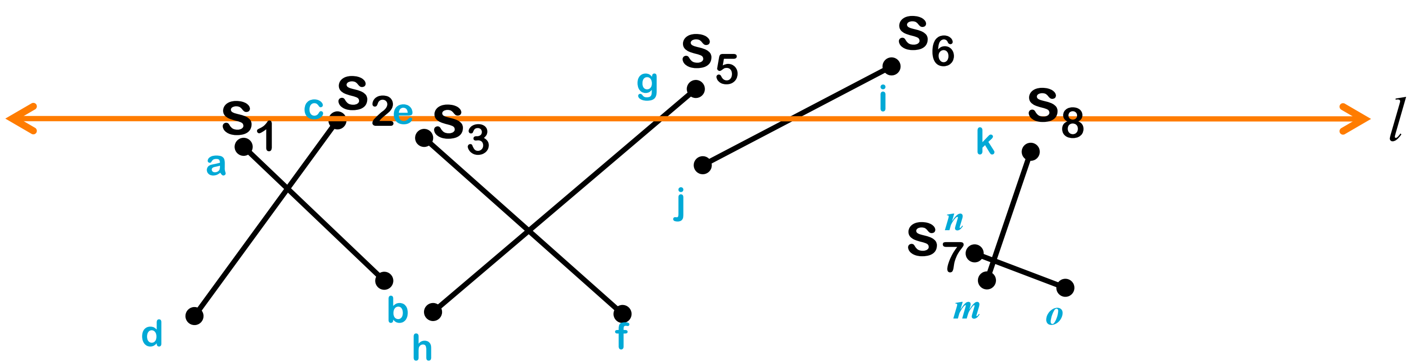
Check intersection: S_5S_6



Plane Sweep

- Status of l : (insert S_2 to T)
  - $S_2S_5S_6$

Check intersection: S_2S_5

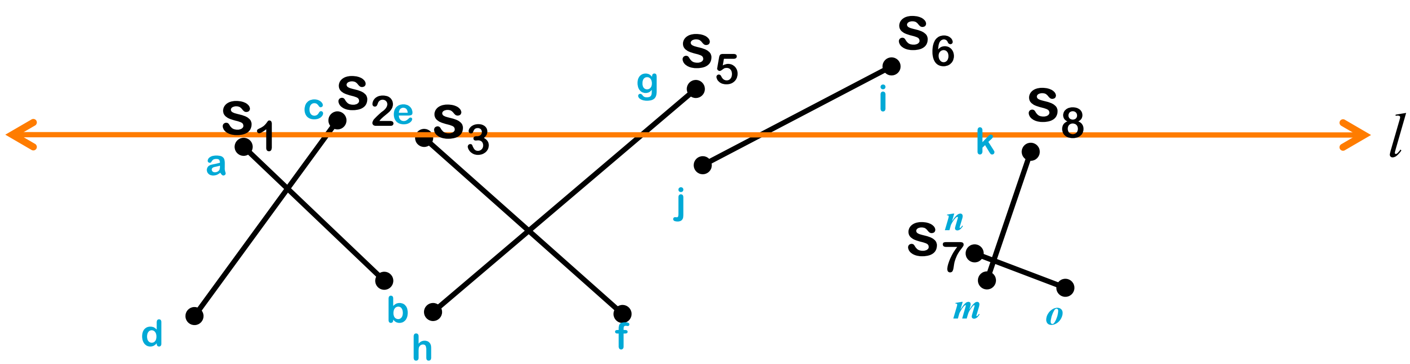


Plane Sweep

- T - Status of l : (insert S_3 to T)
  - $S_2S_3S_5S_6$

Check intersection: S_3S_5

Check intersection: S_3S_2

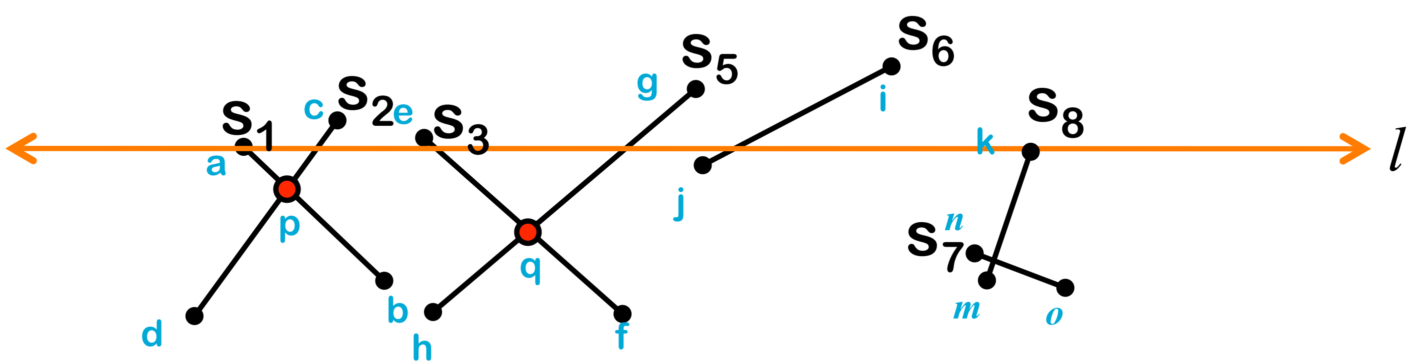


Plane Sweep

- T - Status of l : (insert S_1 and S_8 to T)
  - $S_1S_2S_3S_5S_6S_8$

Check intersection: S_1S_2

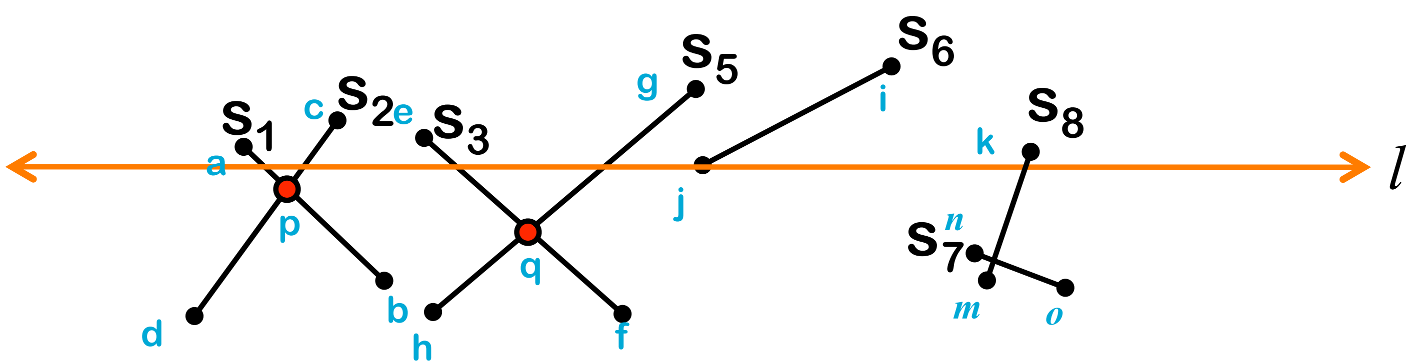
Check intersection: S_8S_6



Plane Sweep

- T - Status of l : (delete S_6 to T)
  - $S_1S_2S_3S_5S_8$

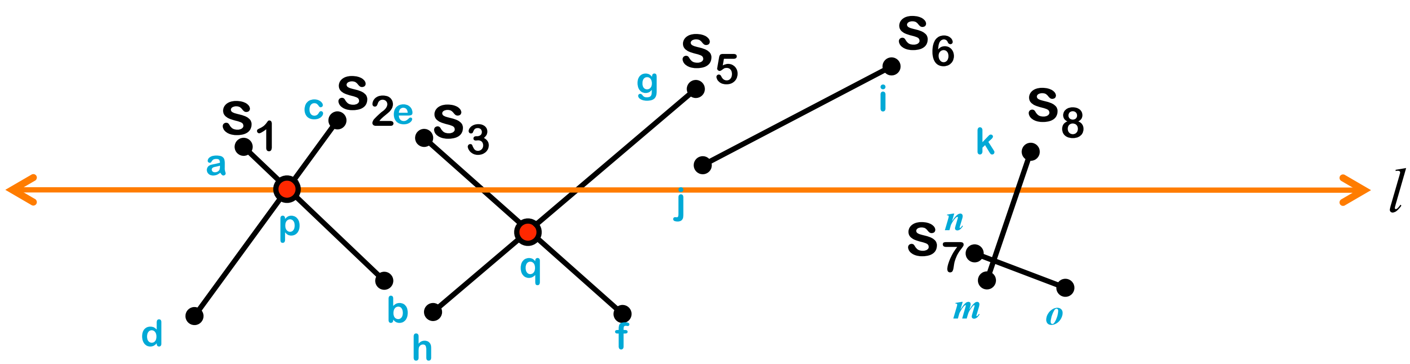
Check intersection: S_5S_8



Plane Sweep

- T - Status of l : (swap S_1 and S_2 in T)
  - $S_2S_1S_3S_5S_8$

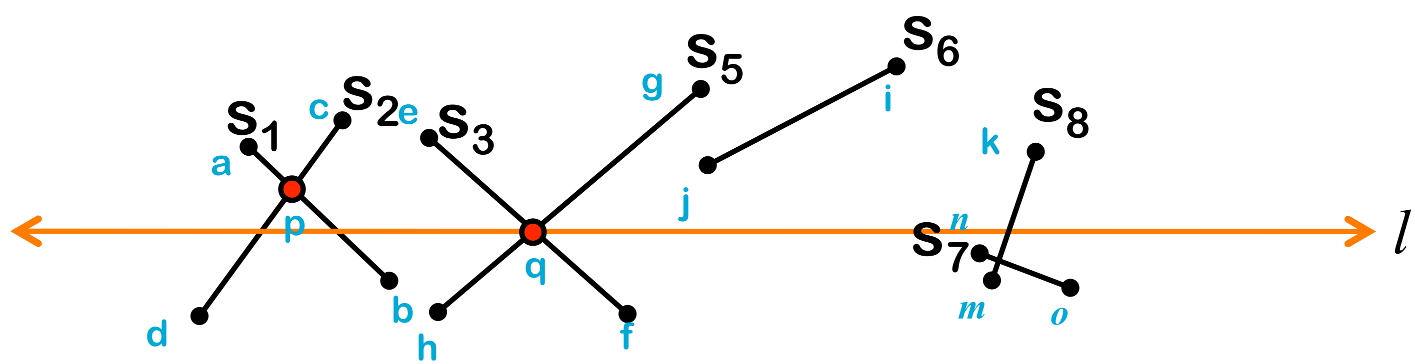
Check intersection: S_1S_3



Plane Sweep

- T - Status of l : (swap S_3 and S_5 in T)
  - $S_2 S_1 S_5 S_3 S_8$

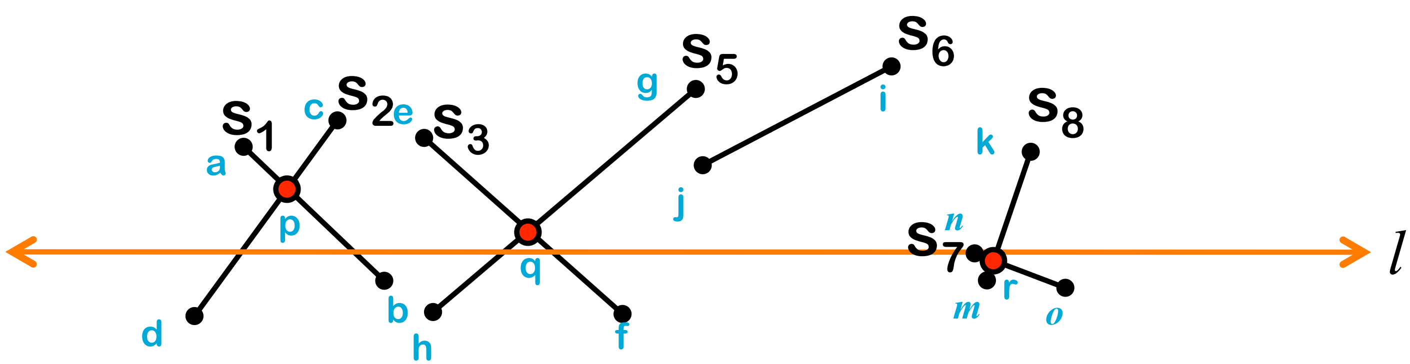
Check intersection: S_1S_5
Check intersection: S_3S_8



Plane Sweep

- T - Status of l : (add S_7 to T)
  - $S_2S_1S_5S_3S_7S_8$

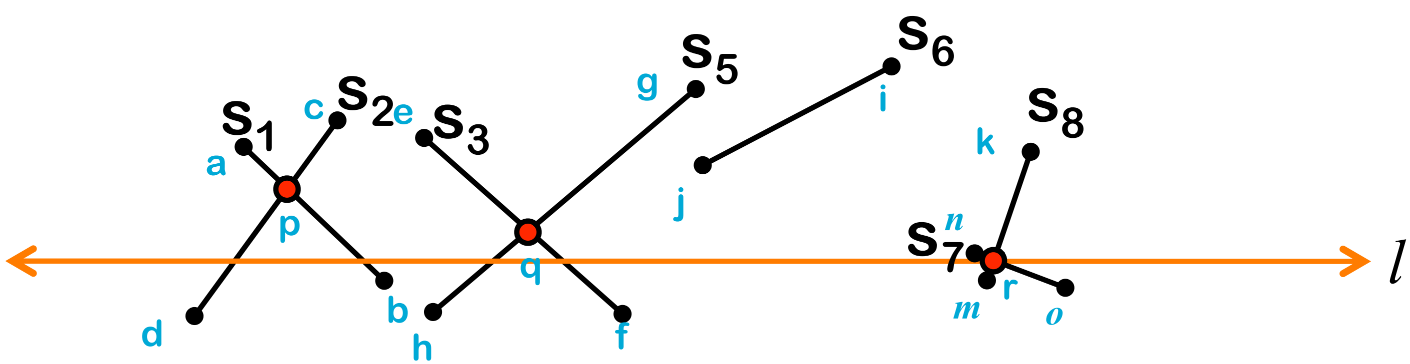
Check intersection: S_7S_3
Check intersection: S_7S_8



Plane Sweep

- T - Status of l : (swap S_7 $S_{8\ \text{in}}$ T)
  - $S_2S_1S_5S_3S_8S_7$

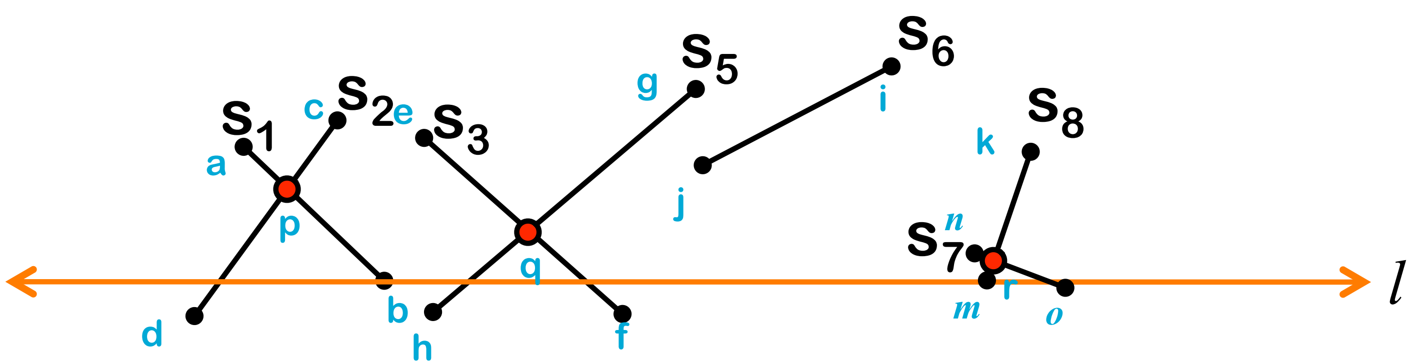
Check intersection: S_8S_3



Plane Sweep

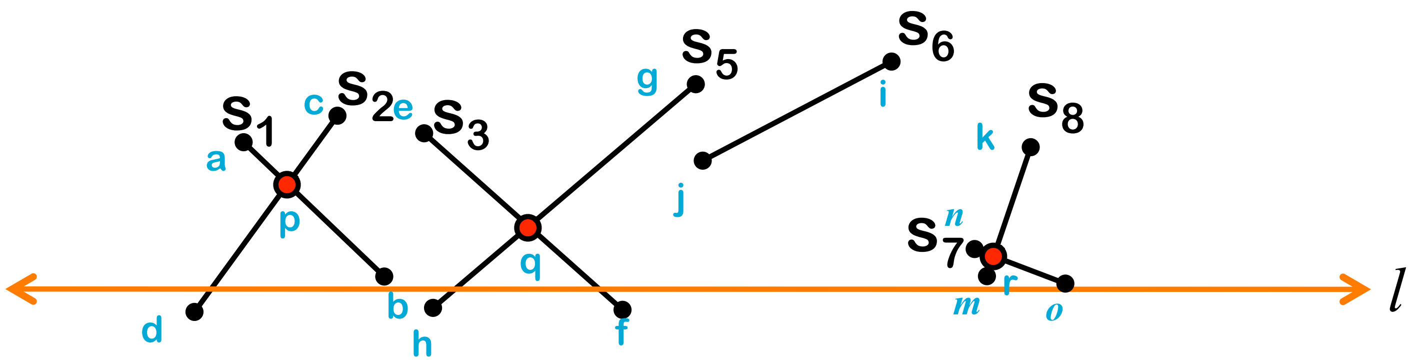
- T - Status of l : (delete S_1 , S_8 from T)
  - $S_2S_5S_3S_7$

Check intersection: S_2S_5



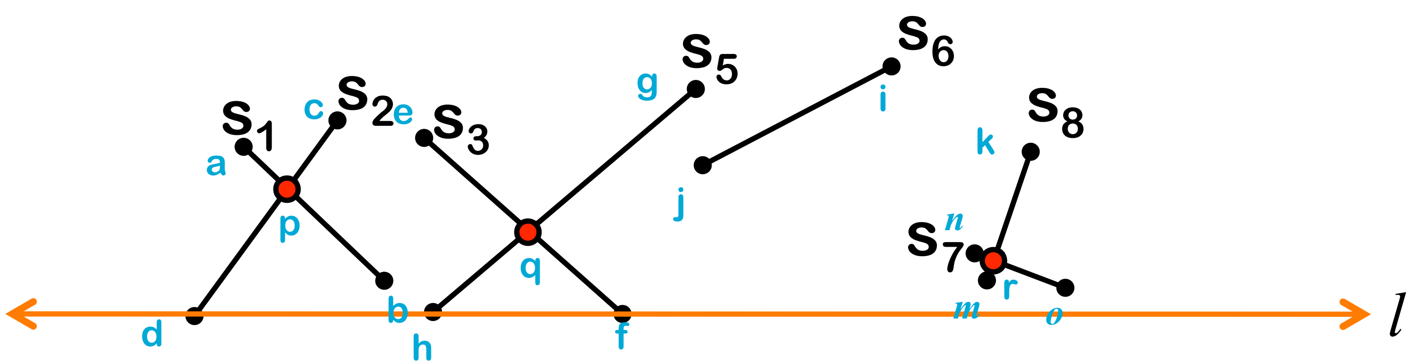
Plane Sweep

- T - Status of l : (delete S_7 from T)
  - $S_2S_5S_3$



Plane Sweep

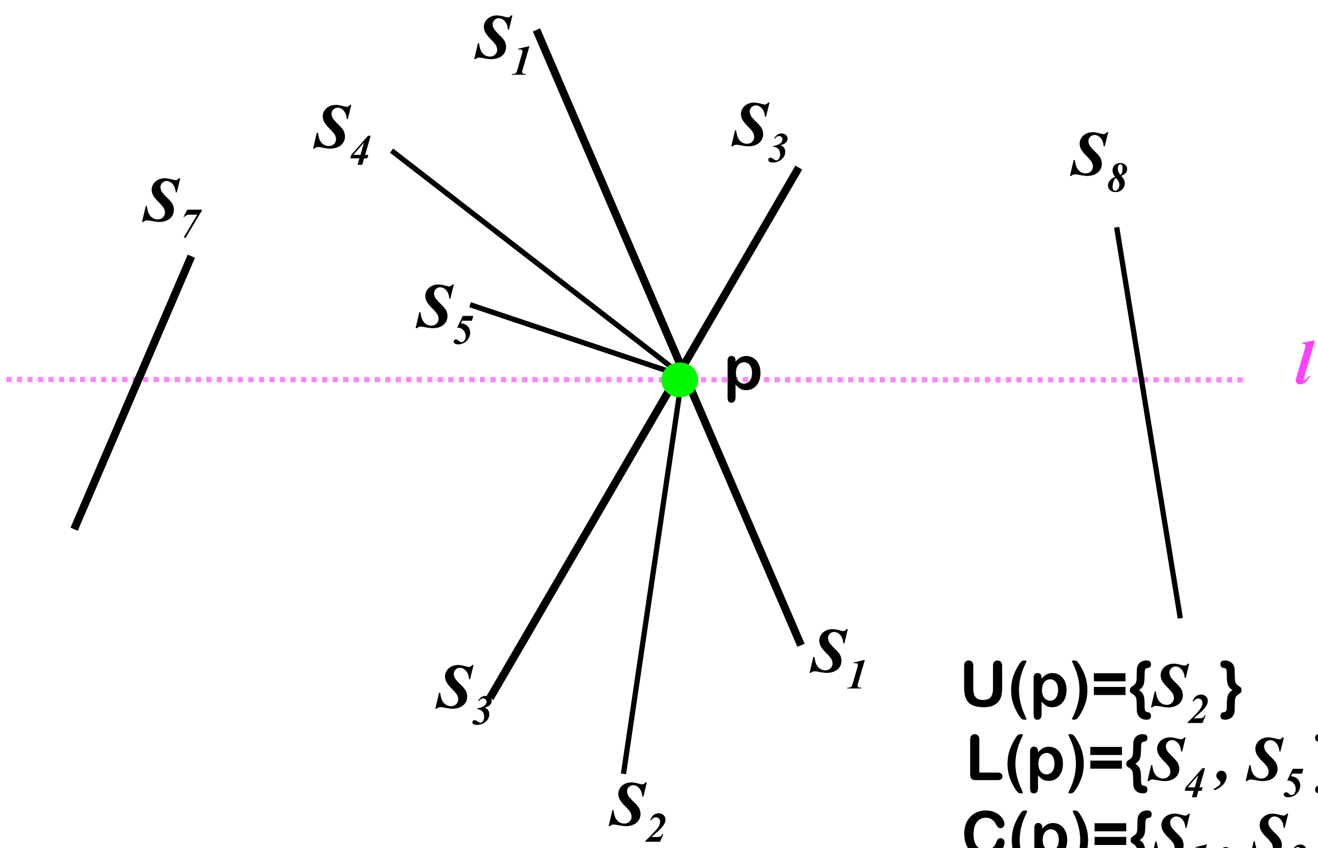
- T - Status of l : (delete S_2 ,S_5 ,S_3 from T)
  - ϕ



Nasty Cases (Degeneracies)

- Horizontal lines
- Overlapping line segments
- Multiple line segments intersect at one single point

Handling Changes in Status



U(p)={S_2}
L(p)={S_4, S_5}
C(p)={S_1, S_3}

HandleEventPoint (p)

[redacted]

See your textbook for detail

Most of these are just some work on bookkeeping

[redacted]

Event Queue Structure

- Event queue requires the following methods
  - remove next event and return it to be treated
  - among 2 events with the same y-coordinate, the one with smaller x-coordinate is returned

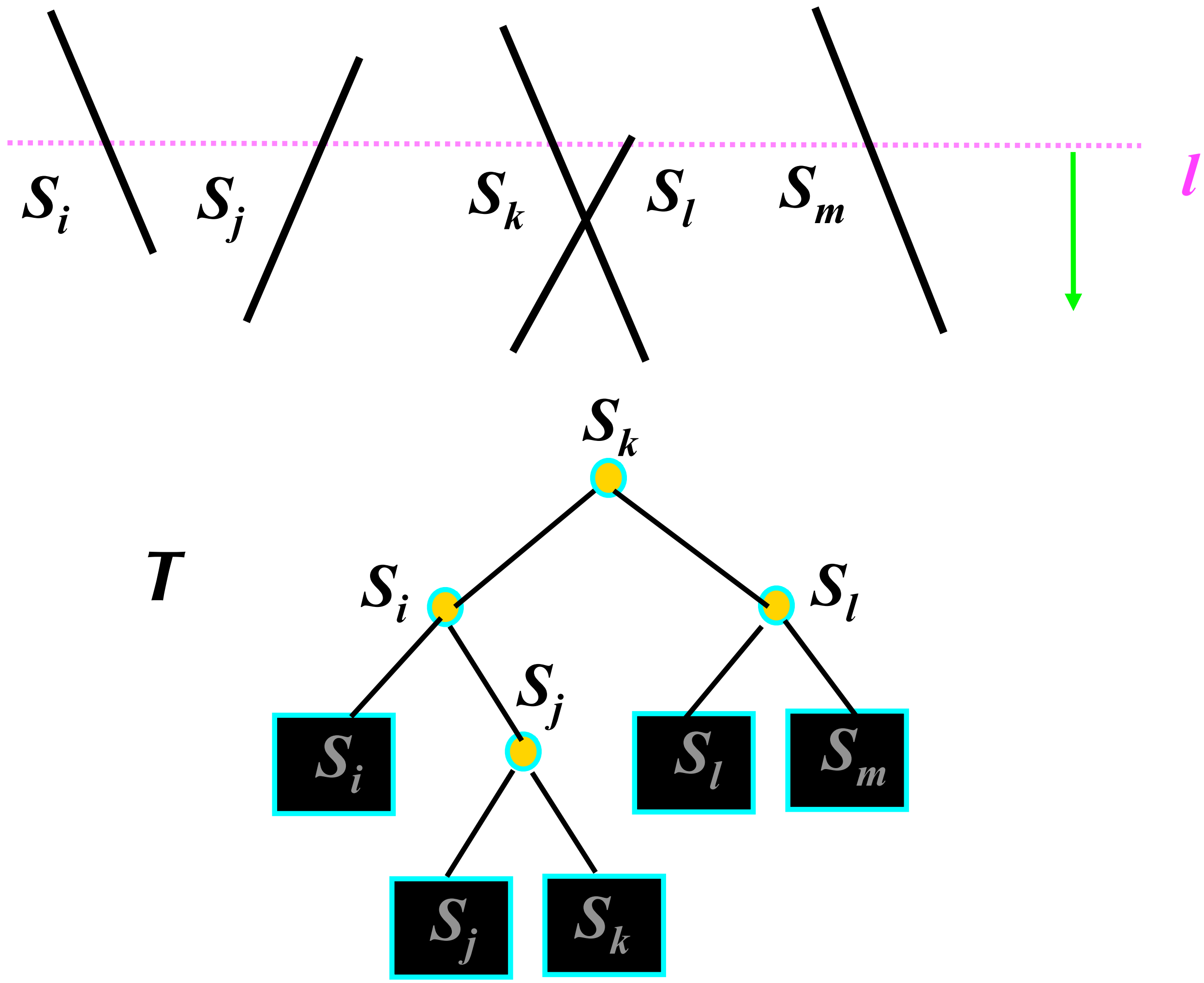
    (left-to-right priority order)
  - allows for insertions & check if it is already there
  - allows 2+ event points to coincide

    (ex) two upper end points coincide

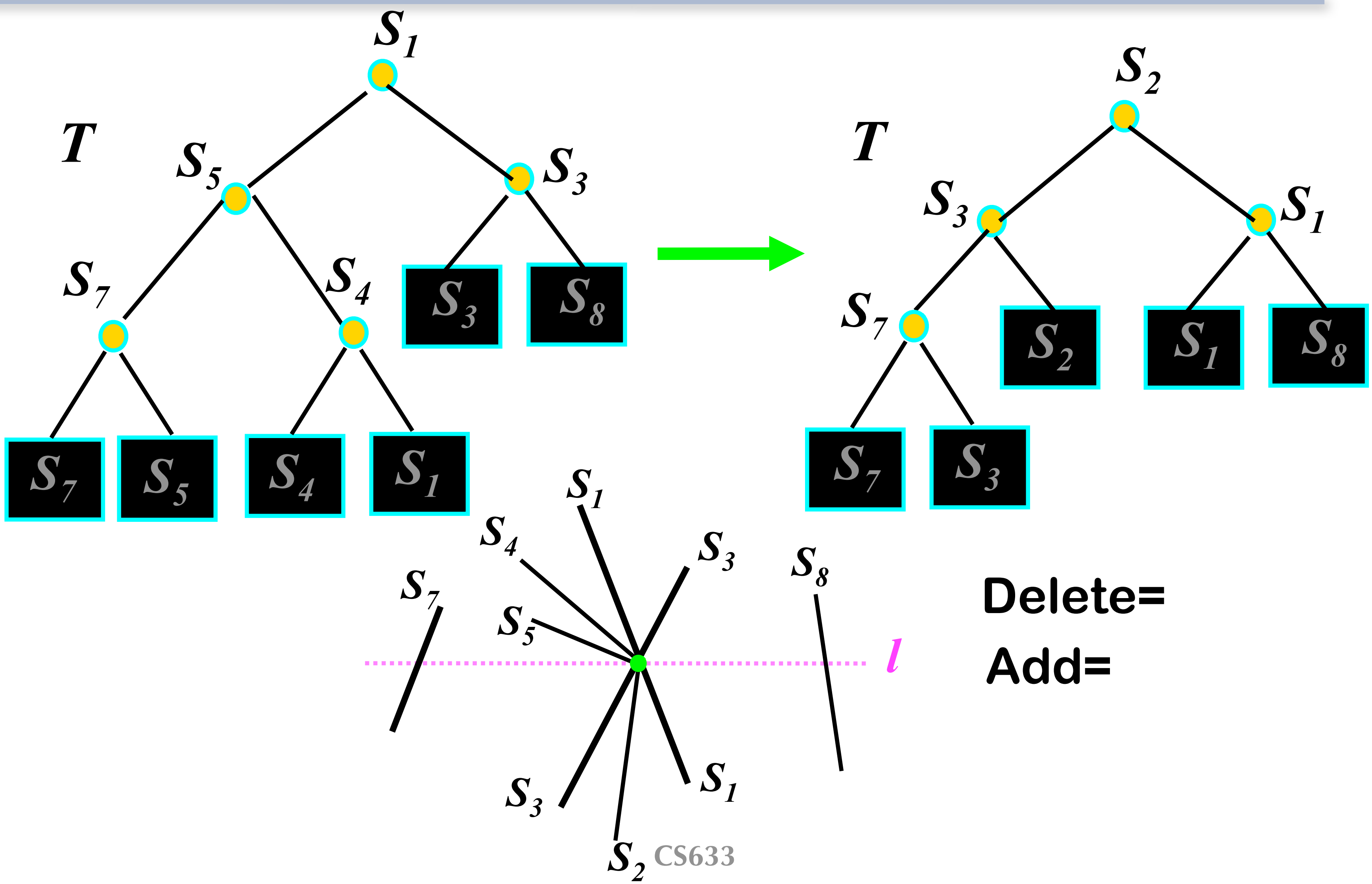
Status Structure, T

- Store the segments in a balanced binary search tree *T* according to their orders
  - both fetching & insertion takes $O(\log m)$ time, where m is the number of events
- Maintain the status of *l* using *T*
  - the left-to-right order of segments on the line $l \leftrightarrow$ the left-to-right order of leaves in *T*
  - segments in internal nodes guide search
  - each update and search takes $O(\log n)$

Status Structure, T



Handling Changes in Status



Delete=

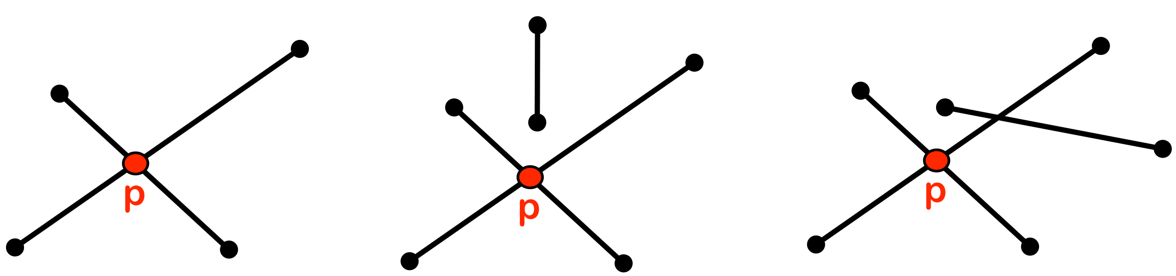
Add=

Algorithm Analysis

- Important property
  - All the intersections above the sweep line must be found
  - Poof: When the sweep line is “Very close” to the intersection, its intersecting line segments must become adjacent!

Algorithm Analysis

- Correctness: Does the algorithm find all intersections? (sketch)
  - Assume there is an intersecting point p that is not found
  - ⇒ The segments intersecting at p never become adjacent when the line sweeps down
  - ⇒ There is no event above p, which makes the segments adjacent
  - ⇒ However, this is not possible.

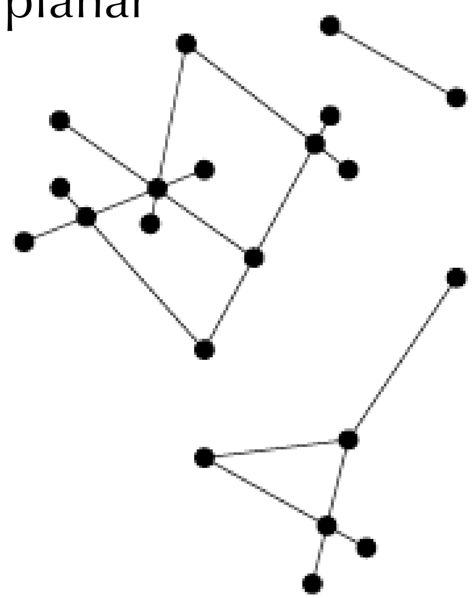


Algorithm Analysis

- Let *S* be a set of n segments in a plane
- All intersections in *S* can be reported in
  - $O(n \log n + k \log n)$ time
    - where k is the size of the output (output includes intersection points and line segments intersecting at the points)
  - $O(n+I)$ space
    - where I is the size of the number of intersections

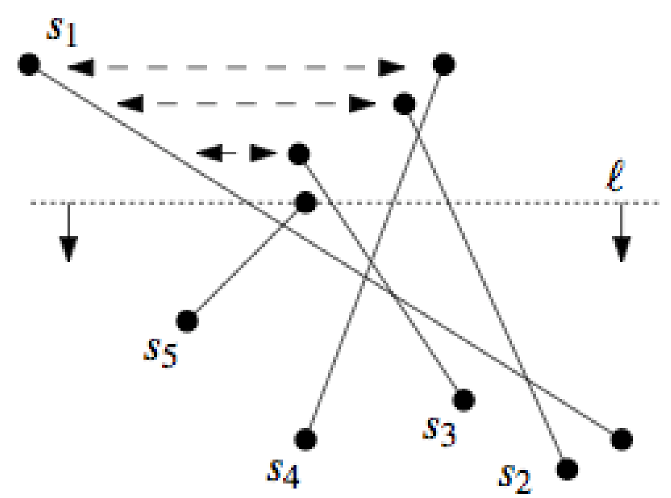
Algorithm Analysis

- With better analysis using Euler's Formula
  - $O(n \log n + I \log n)$ time
    - where I is the size of the number of intersections
    - Let p be all intersections, then $k = \sum_p m(p)$.
    - By treating the segments and intersections as a planar graph, we know $m(p)$=degree(p)
    - Therefore, $k = \sum_p m(p) = \sum_p \text{degree}(p) = 2|E|$.
    - So, how large is |E|, the number of edges in G?



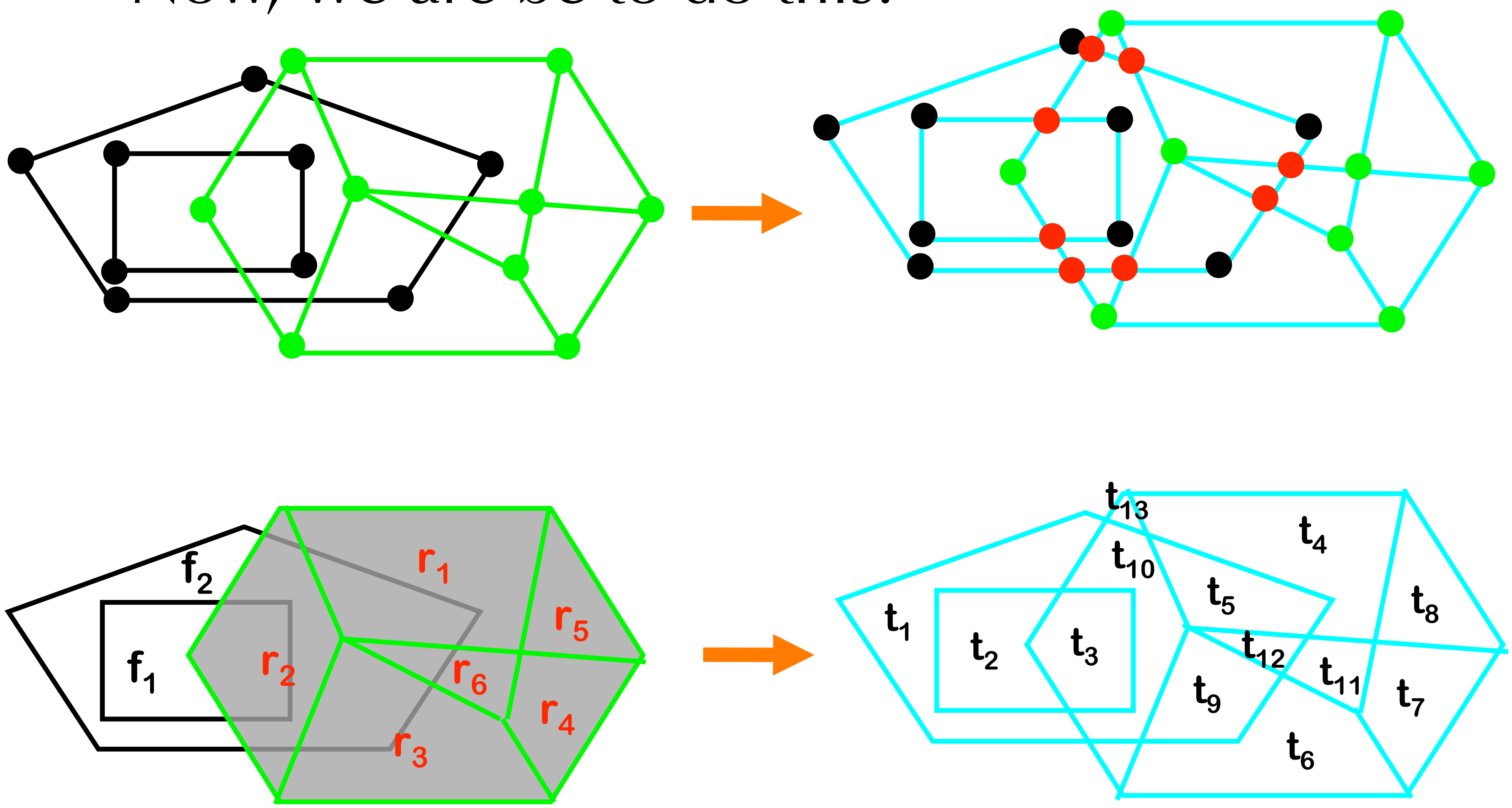
Algorithm Analysis

– *O(n)* space, without storing all events
  - e.g. only store intersection points of pairs of segments that are currently adjacent on the sweep line



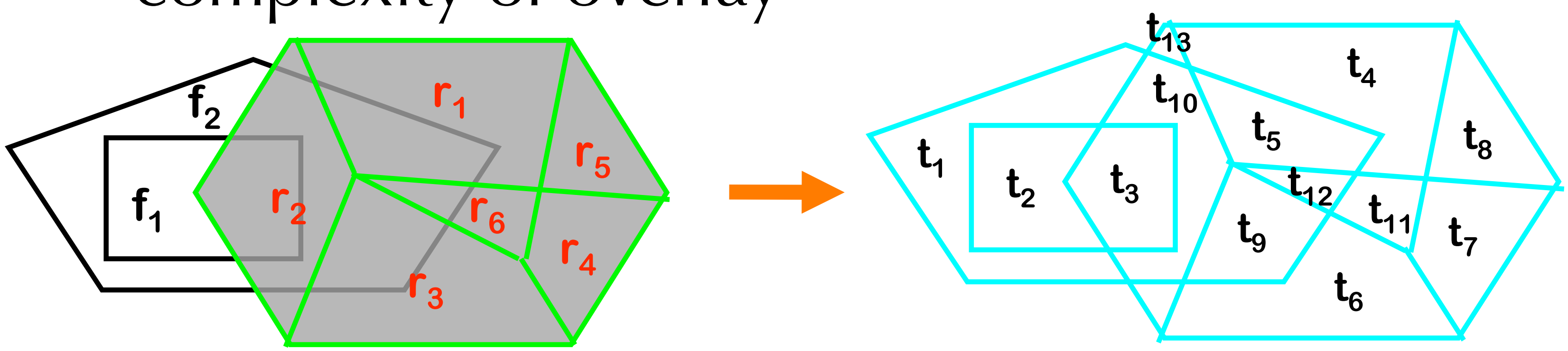
Application 1
Thematic Map Overlay

- Now, we are be to do this:



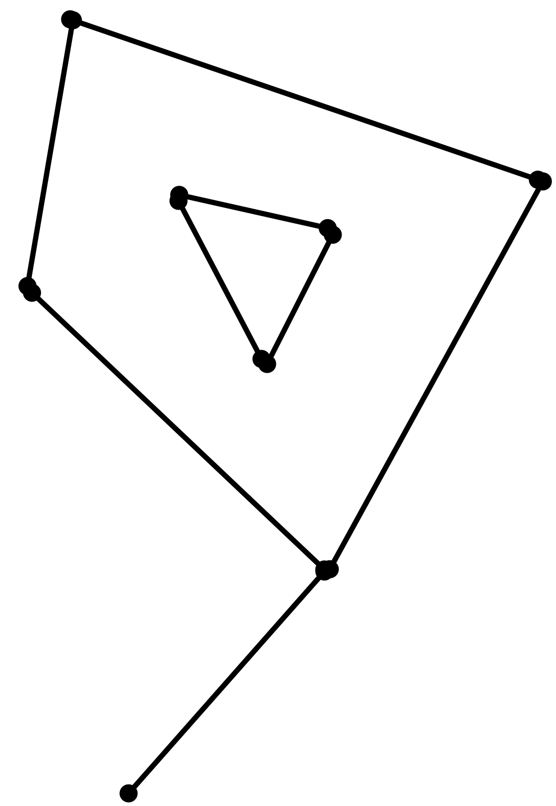
Application 2
Overlay of Subdivisions

- Let S_1, S_2 be two planar subdivisions of complexity n_1 and n_2 respectively; and let $n = n_1 + n_2$
- Overlay of S_1 and S_2 can be constructed in $O(n \log n + k \log n)$ time, where k is the complexity of overlay



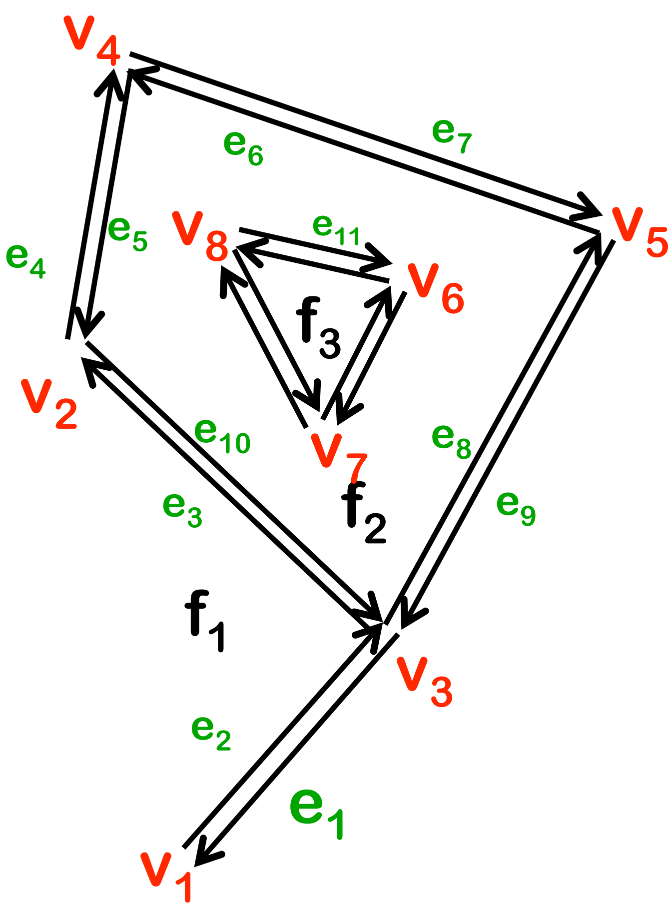
Define a Subdivision: Doubly-Connected Edge List

- 3 records: vertices, faces and "half-edges"
- Vertex:
  - coordinates(v)
  - a ptr to a half-edge
- Face:
  - OuterComponent(f): outer boundary
  - InnerComponent(f): holes boundaries
- Half edge:
  - a ptr to Origin(e)
  - a ptr to a twin-edge
  - ptrs to Next(e) & Prev(e) edges
  - its left IncidentFace(e)



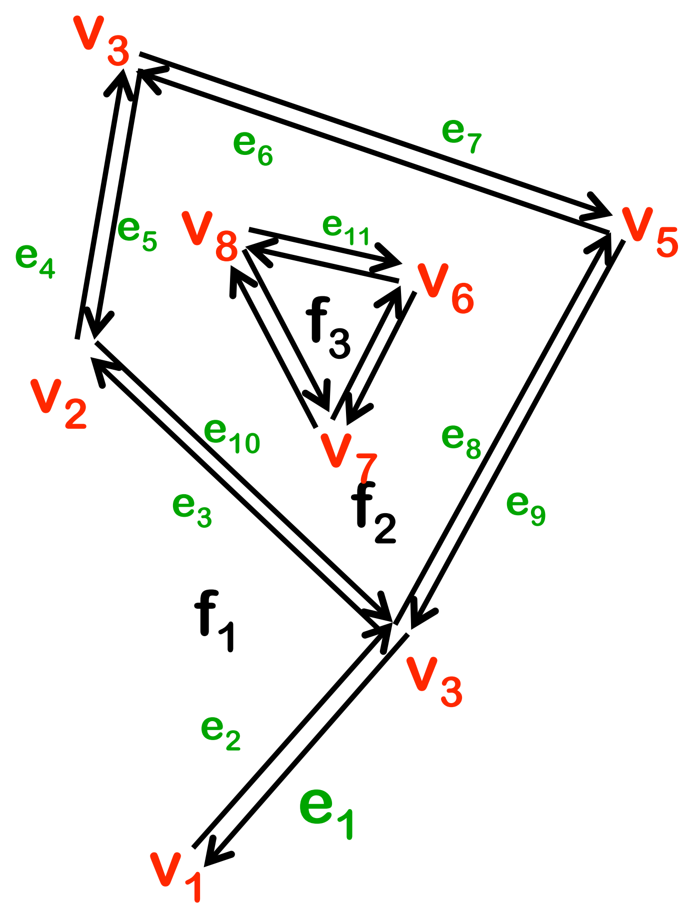
Doubly-Connected Edge List

- v_3:
  - coordinates(v)
  - a ptr to a half-edge e_3
- f_2:
  - OuterComponent(f): e_6
  - InnerComponent(f): e_{11}
- e_1:
  - a ptr to Origin(e): v_3
  - a ptr to a twin-edge: e_2
  - ptrs to Next(e) & Prev(e) edges: e_2 and e_9
  - its left IncidentFace(e): f_1

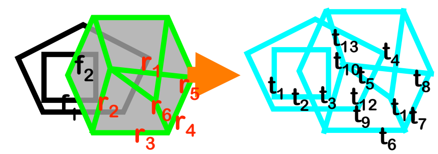


Doubly-Connected Edge List

- How do you find all incident edges of f_1 ?
- How do you find all incident vertices of e_4?
- How do you find all incident edges of v_3 ?
- How do you find all incident faces of v_3 ?

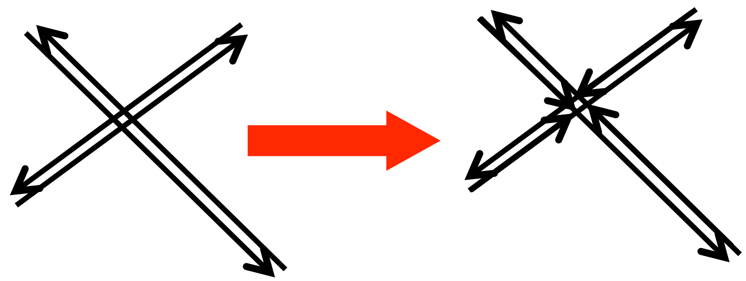


Application 2
Overlay of Subdivisions



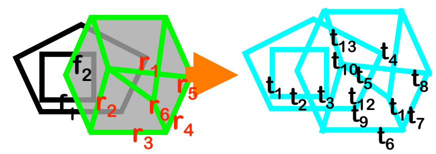
1. Find intersections

2. Update half-edges



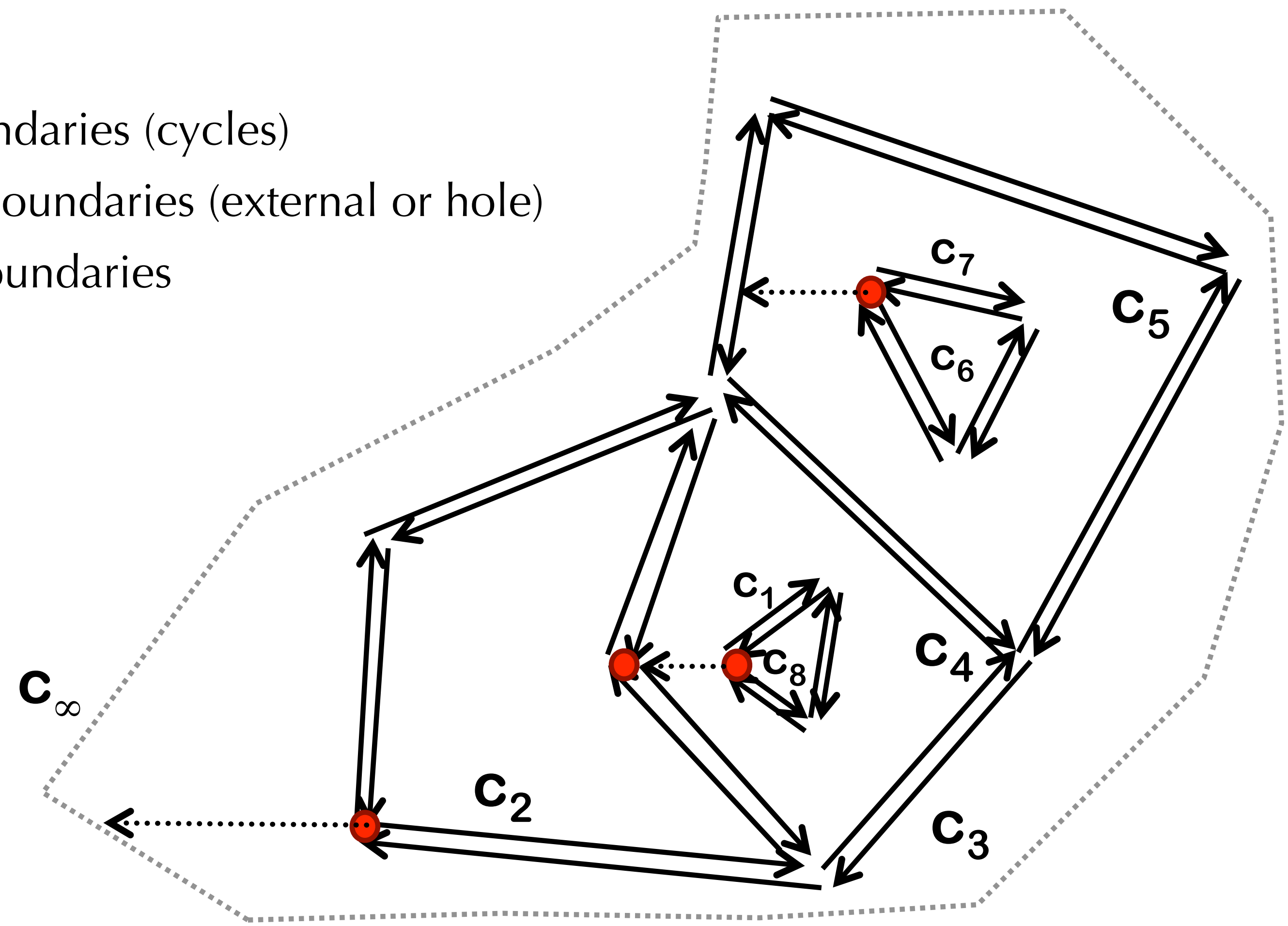
3. Update faces
   1. Find boundaries
   2. Classify boundaries (external or hole)
   3. Group boundaries

Application 2
Overlay of Subdivisions



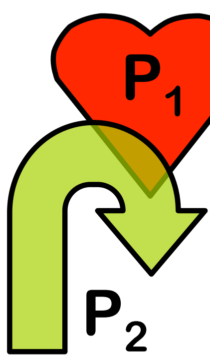
Update faces

1. Find boundaries (cycles)
2. Classify boundaries (external or hole)
3. Group boundaries

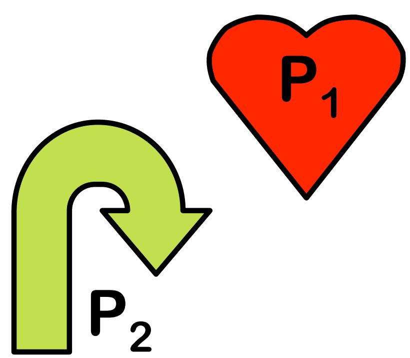


Application 3
Polygon intersection

- Let P_1, P_2 be two polygons, check if they collide with each other in $O(n \log n)$ time



collision



no collision

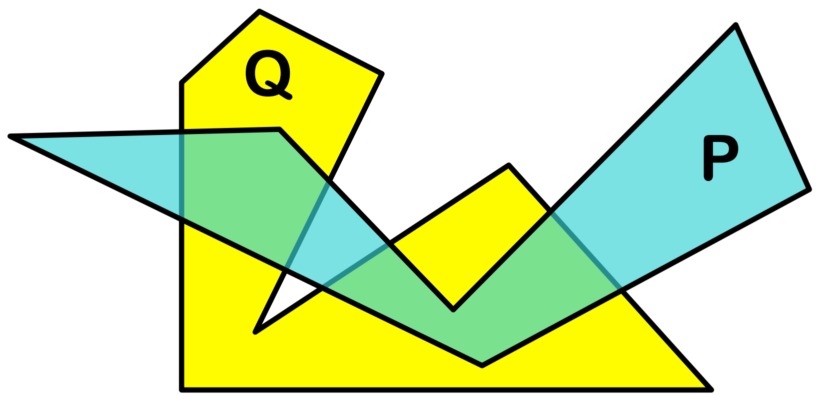
Application 4
Boolean Operations

- Let P_1, P_2 be two polygons with n_1 and n_2 vertices respectively; and let $n = n_1 + n_2$

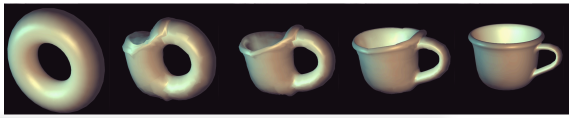
- Their Boolean operations (intersection, union, and difference) can each be computed in $O(n \log n + k \log n)$ time, where k is the complexity of the output

Application 4
Boolean Operations

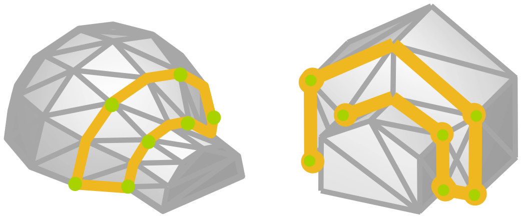
- P-Q
- P∪Q
- P∩Q



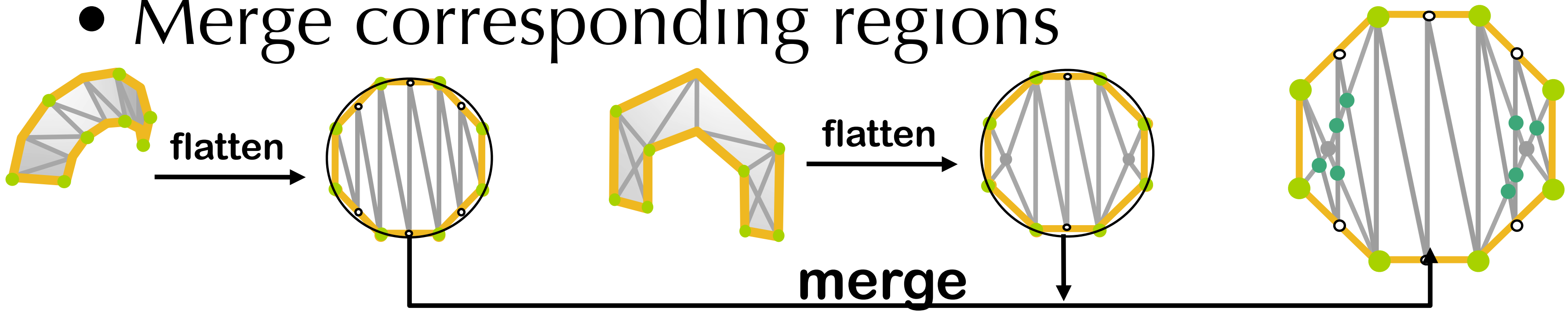
Application 5
3D Morphing



- Compute or specify corresponding regions



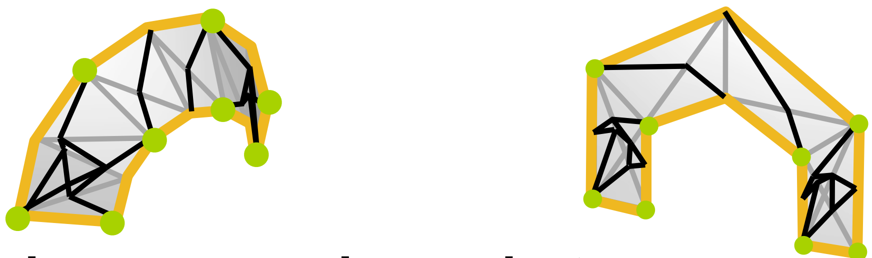
- Merge corresponding regions



Feature-based Surface Decomposition for Polyhedral Morphing; Gregory, Arthur; State, Andrei; Lin, Ming C.; Manocha, Dinesh; Livingston, Mark A.. Proceedings of the Symposium on Computational Geometry. 1999. pp 415-416.

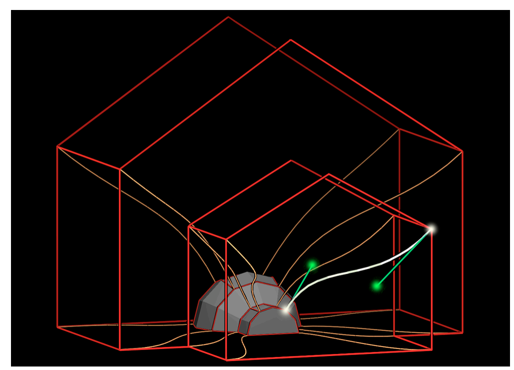
Application
3D Morphing

- Now we have found the correspondences for all points!



- Specify how each point move to its corresponding point

Another example using this technique



Conclusion

- **Line segments intersection**
  - Line sweep paradigm
  - Output sensitive algorithm
- **Doubly-linked edge list**
  - Representing subdivisions
- **Applications**
  - GIS map overlay (lines, regions)
  - 2D collision detection and Boolean operations
  - 3D morphing

Homework Assignment

- Exercise: 2.1, 2.11, 2.14

Next time: Art Gallery problem & Triangulation