Line Segment Intersection

Line Segment Intersection

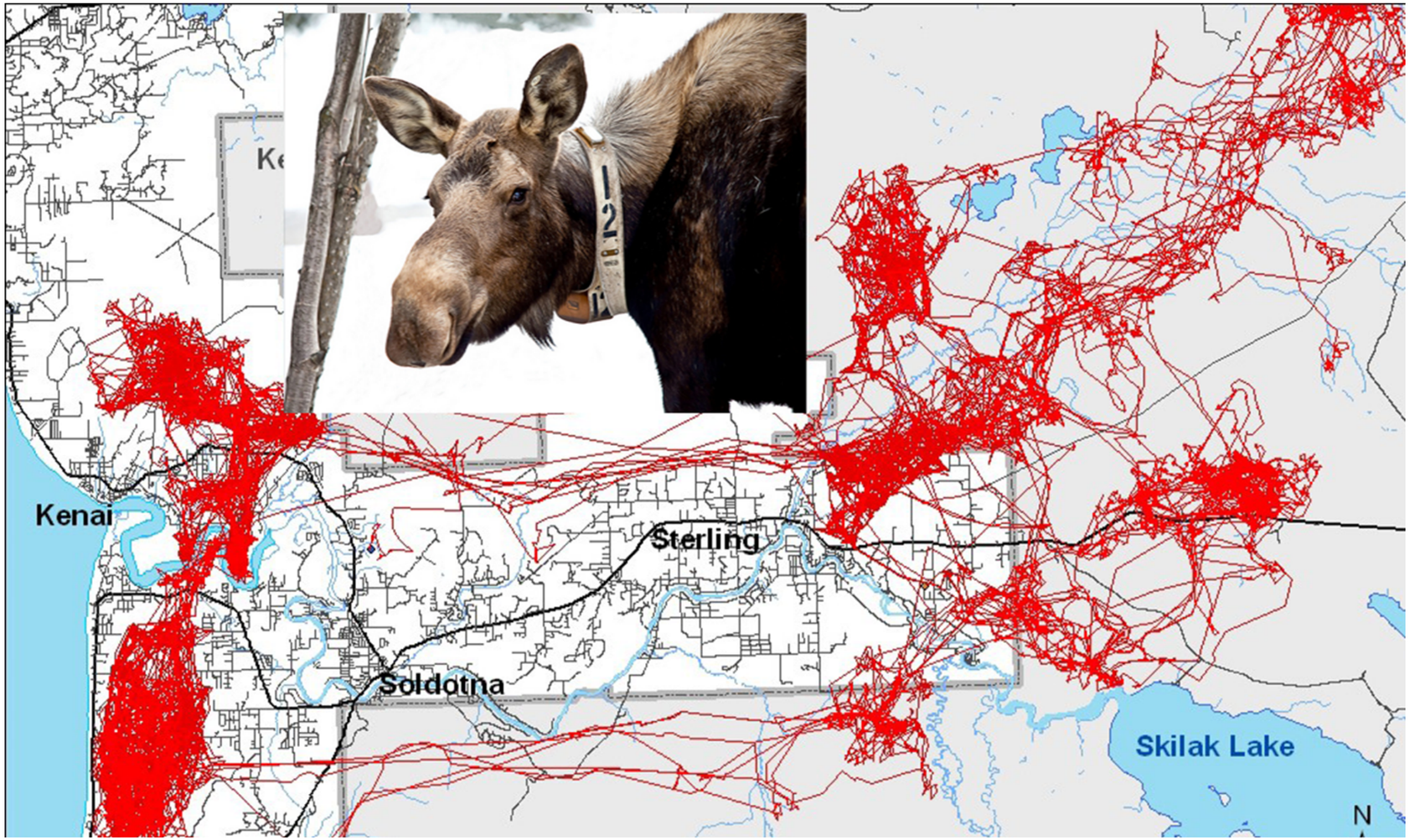
Algorithmic Problem: Computing intersections in a set of line segments

Algorithmic Technique: Plane Sweep

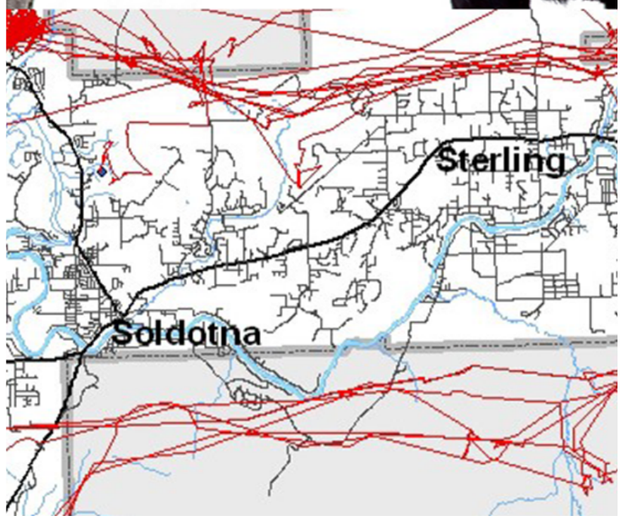
Data Structure: How to store a planar subdivision

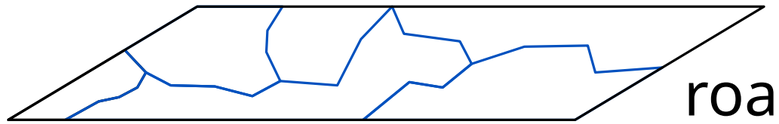
Motivation

Where did the moose cross the road?



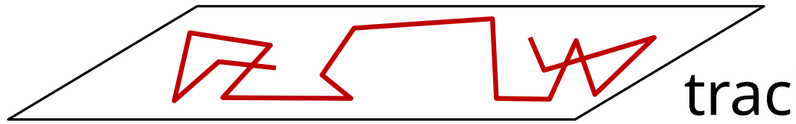
Motivation: Map overlay





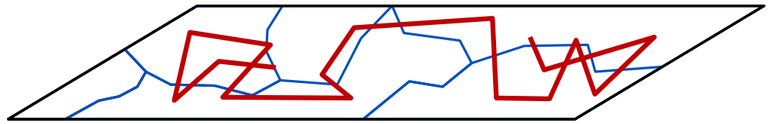
road network

+



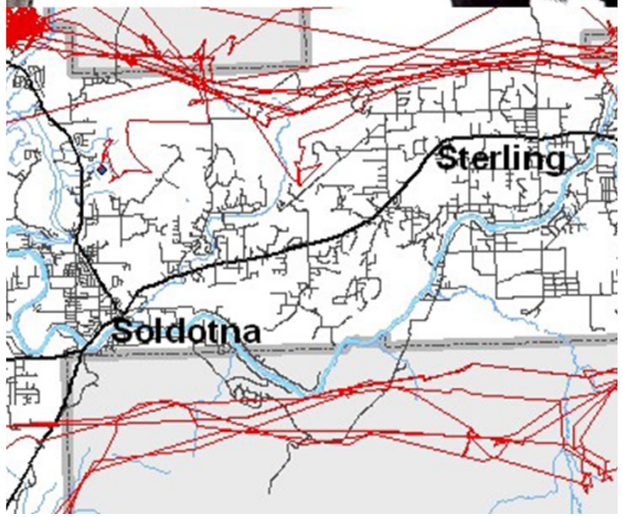
track

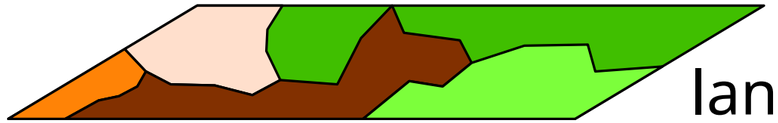
=



combination of layers

Motivation: Map overlay



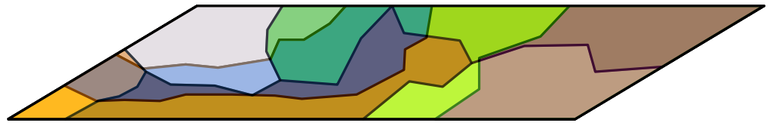


land use

+

rain fall

=

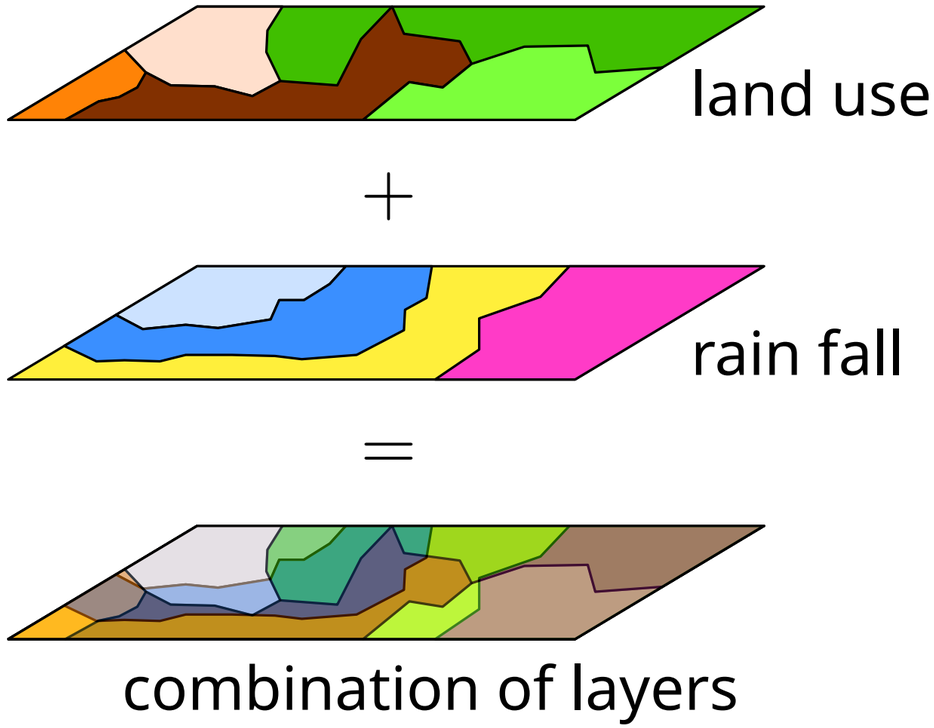


combination of layers

Motivation: Map overlay

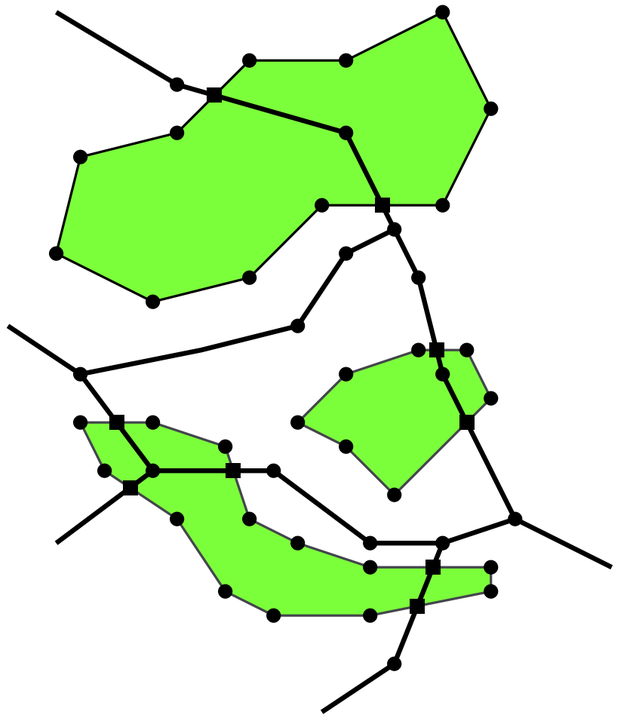
In a geographic information system (GIS) data is stored in separate layers

A layer stores information about particular theme, like land cover, road network, municipality boundaries, …



Motivation: Map overlay

Map overlay: combination of two (or more) map layers

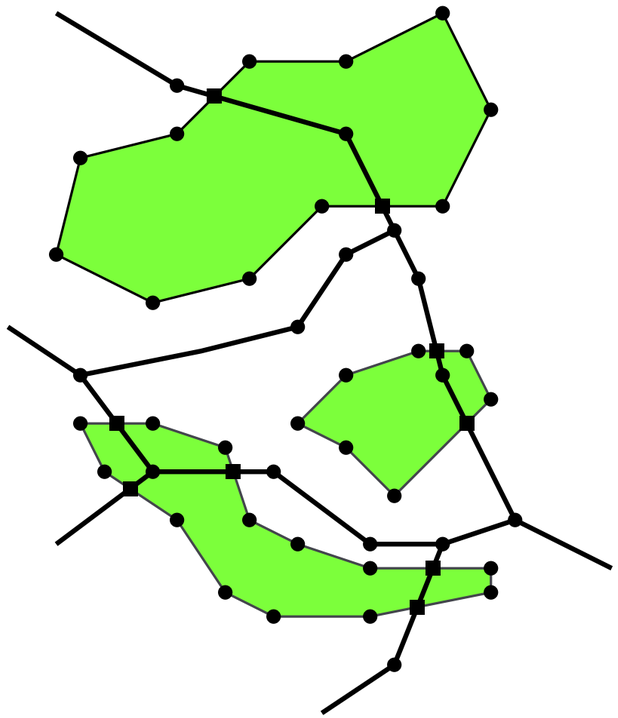


Motivation: Map overlay

Map overlay: combination of two (or more) map layers

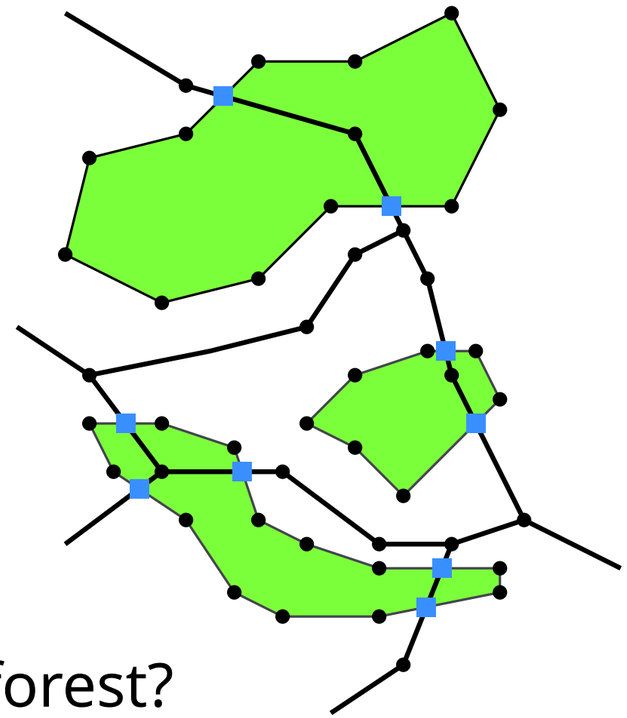
Example:

What is the total length of roads through forests?



Motivation: Map overlay

Map overlay: combination of two (or more) map layers



Example:

What is the total length of roads through forests?

First step:

Where does the road enter/exit the forest?

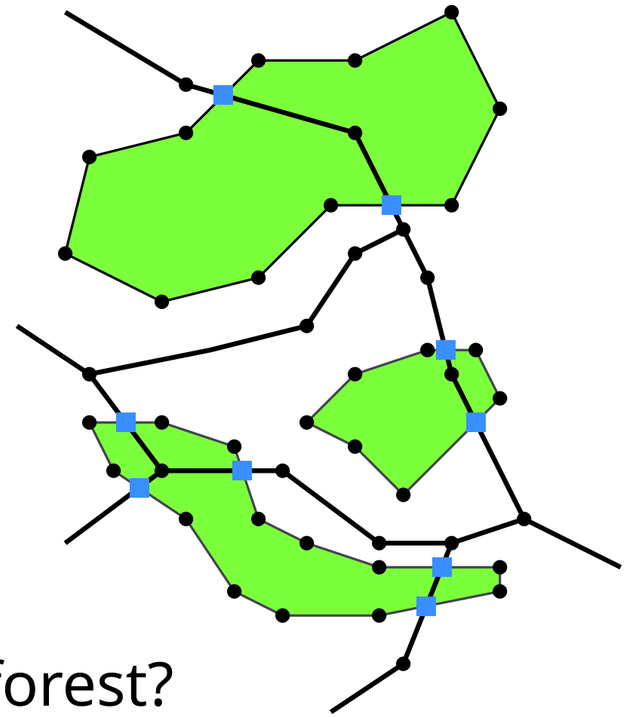
$\rightarrow$ intersections of a set of line segments

Motivation: Map overlay

Map overlay: combination of two (or more) map layers

Example:

What is the total length of roads through forests?



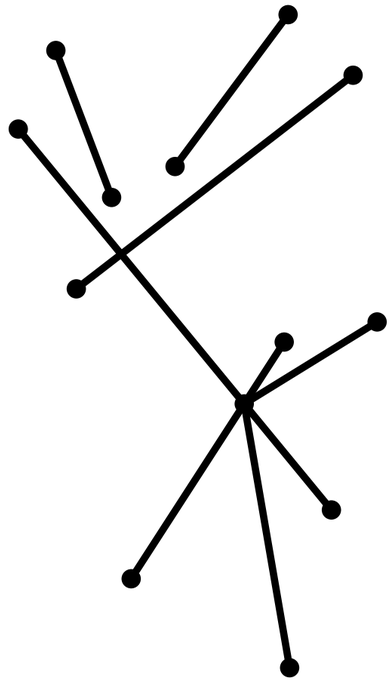
First step:

Where does the road enter/exit the forest?

→ intersections of a set of line segments

Problem Statement

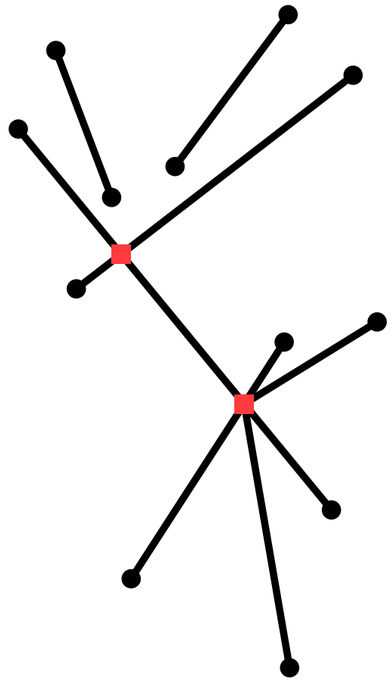
Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane



Problem Statement

Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane

Find:
- all intersection points of two or more line segments
- for each intersection: the intersecting segments

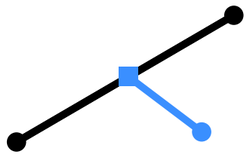


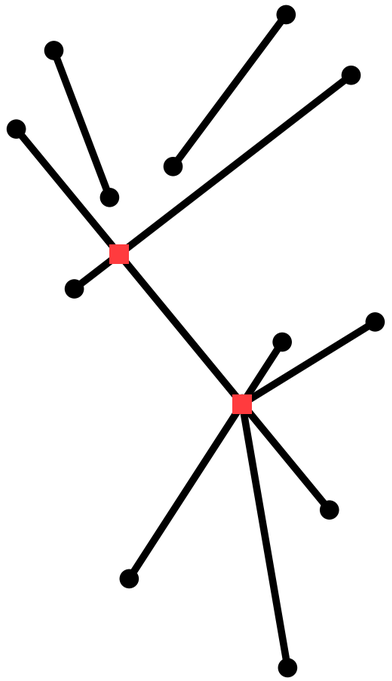
Problem Statement

Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane

Find:
- all intersection points of two or more line segments
- for each intersection: the intersecting segments

Def: line segments are **closed** sets

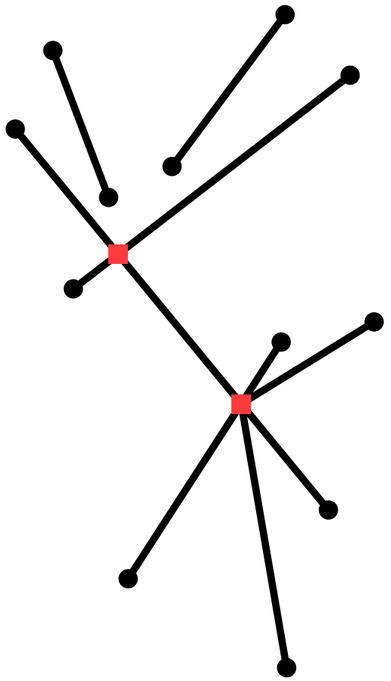




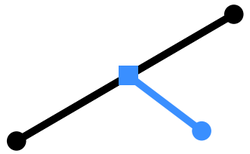
Problem Statement

Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane

Find:
- all intersection points of two or more line segments
- for each intersection: the intersecting segments



Def: line segments are **closed** sets



Discussion:
- simple solution?
- worst-case optimality?

A simple, optimal algorithm?

Algorithm FINDINTERSECTIONS(S)

Input: set S of line segments in the plane

Output: set of intersection points among the segments in S

```
1: for each pair of line segments e_i, e_j ∈ S do
2:     if e_i and e_j intersect then
3:         report their intersection point
```

A simple, optimal algorithm?

Algorithm FINDINTERSECTIONS(S)

Input: set S of line segments in the plane

Output: set of intersection points among the segments in S

```
1: for each pair of line segments e_i, e_j ∈ S do
2:     if e_i and e_j intersect then
3:         report their intersection point
```

running time: $\Theta(n^2)$

correctness: all possibilities checked

Quiz

Is this algorithm wost-case optimal?

Algorithm FINDINTERSECTIONS(S)

Input: set S of line segments in the plane

Output: set of intersection points among the segments in S

```
1: for each pair of line segments e_i, e_j ∈ S do
2:     if e_i and e_j intersect then
3:         report their intersection point
```

A: yes

B: no

Quiz

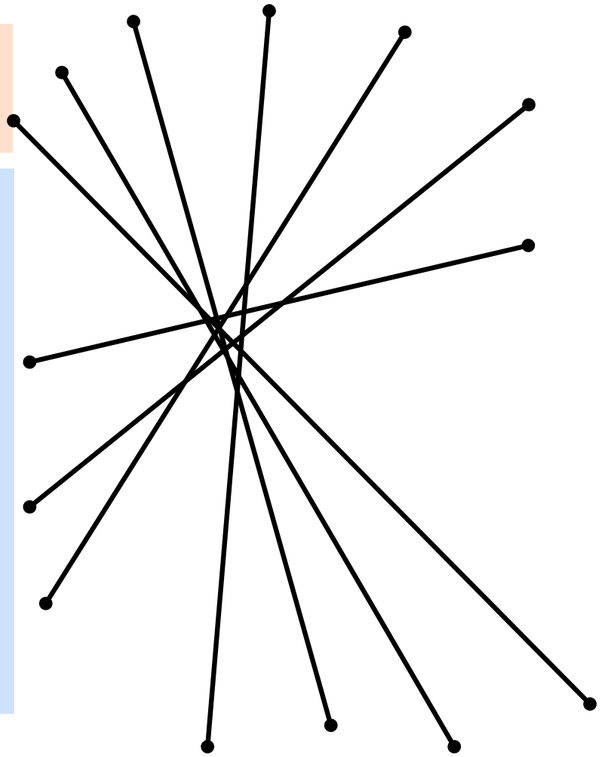
Is this algorithm wost-case optimal?

Algorithm FINDINTERSECTIONS(S)

Input: set S of line segments in the plane

Output: set of intersection points among the segments in S

1: **for** each pair of line segments $e_i, e_j \in S$ **do**
2: **if** e_i and e_j intersect **then**
3: report their intersection point

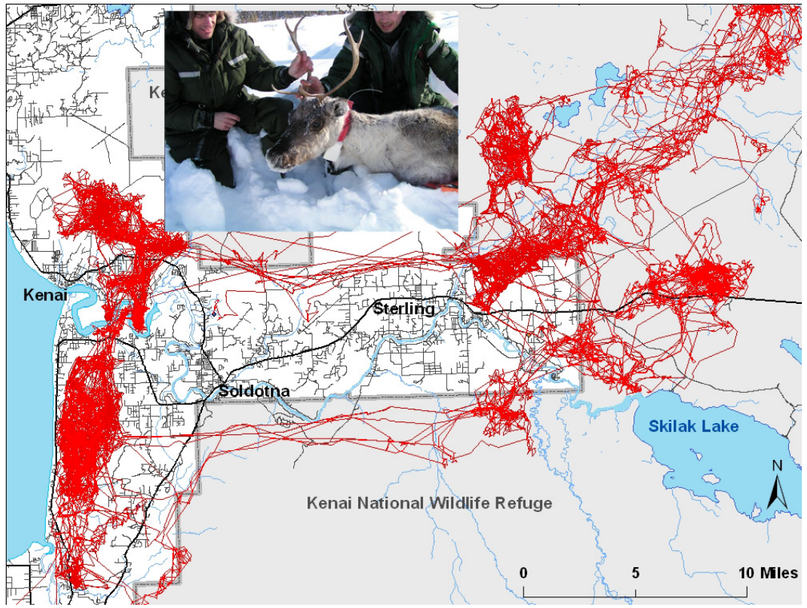


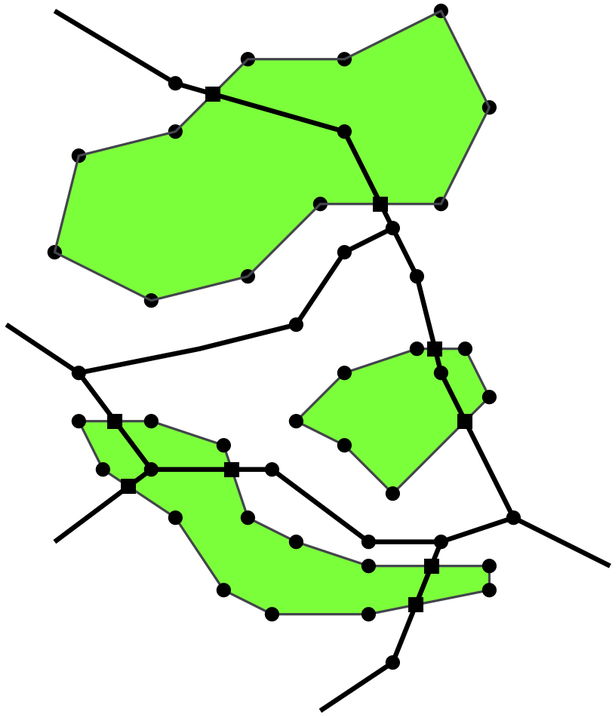
A: yes

B: no

Intersection points in practice

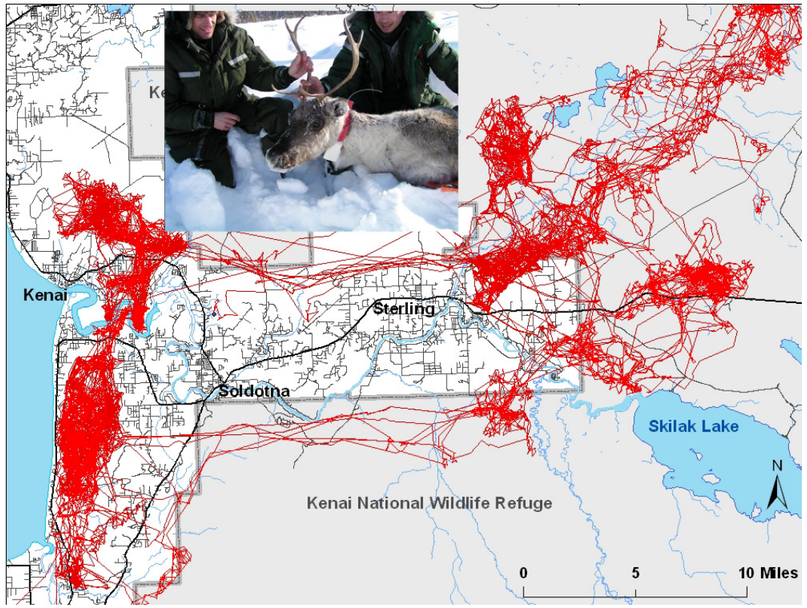
How many intersection points do we typically expect in these applications?

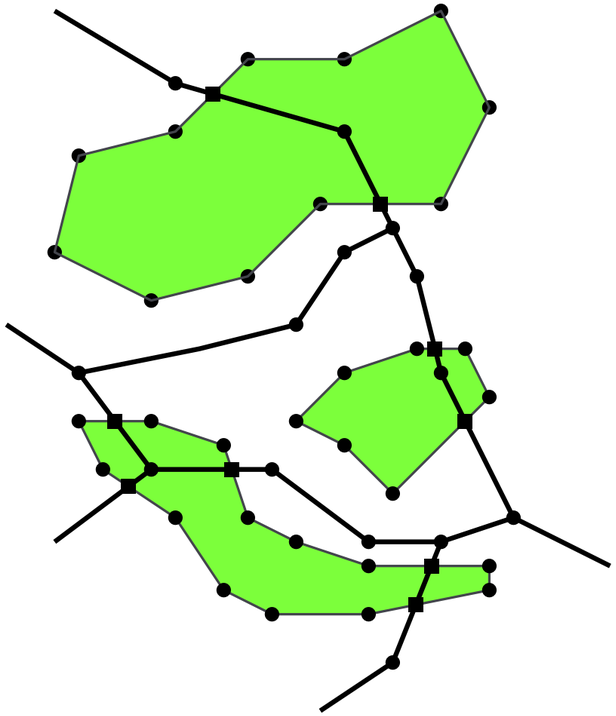




Intersection points in practice

How many intersection points do we typically expect in these applications?

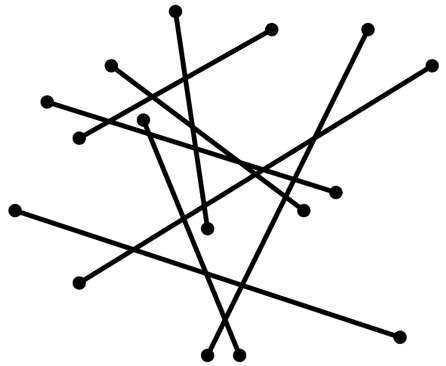




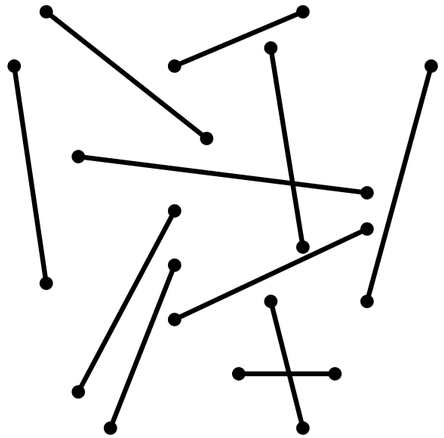
If this number is k, and if $k = O(n)$, it would be nice if the algorithm runs in $O(n \log n)$ time

Output-sensitive algorithm

asymptotic running time is always input-sensitive: depends on n

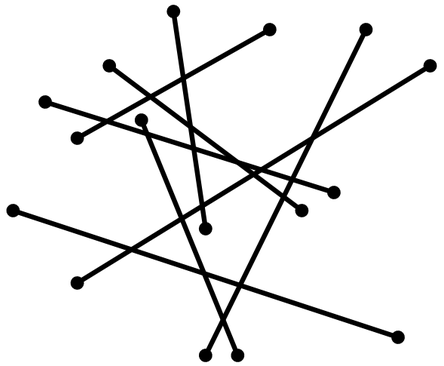


We may also want output-sensitive: algorithm faster if output size k is small

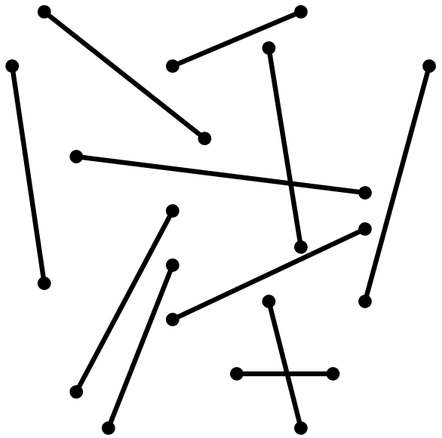


Output-sensitive algorithm

asymptotic running time is always input-sensitive: depends on n



We may also want output-sensitive: algorithm faster if output size k is small



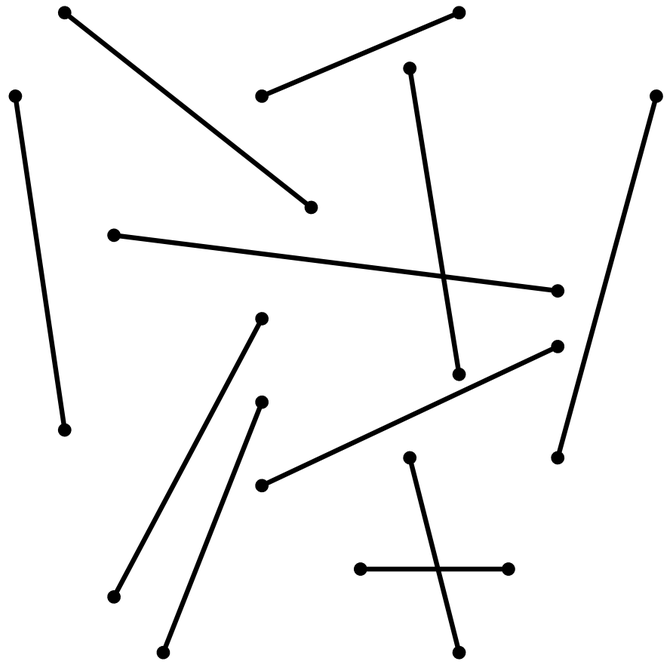
next: designing an output-sensitive algorithm for line segment intersection

Plane Sweep Algorithm for Line Segment Intersection

observations and concepts

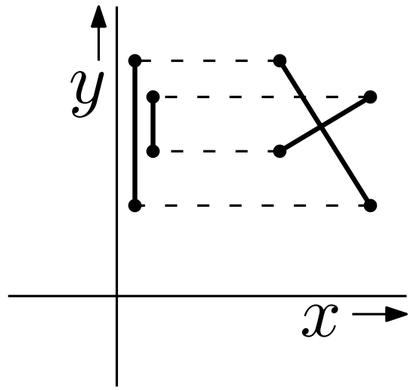
Developing Geometric Algorithms

To develop an algorithm, find useful properties, make various observations, draw many sketches to gain insight.



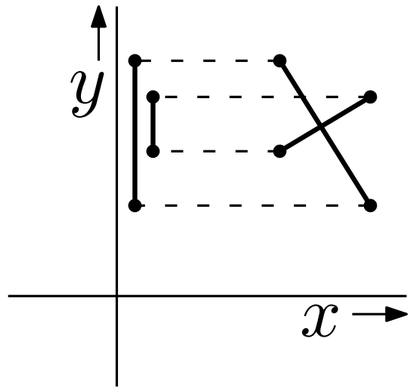
First attempt

Observation: Two line segments can only intersect if their y-spans have an overlap

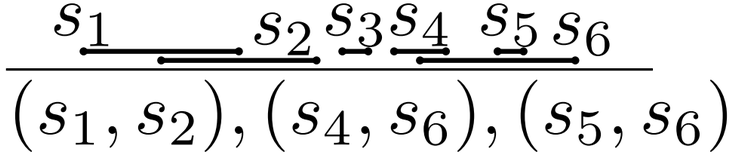


First attempt

Observation: Two line segments can only intersect if their y-spans have an overlap



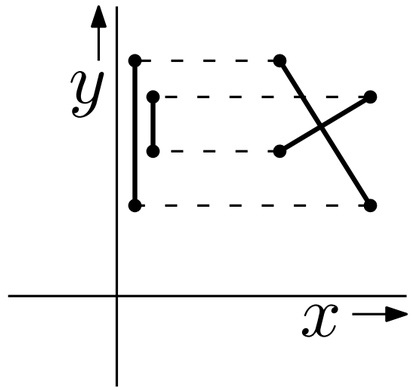
1D problem: Given a set of intervals on the real line, find all partly overlapping pairs



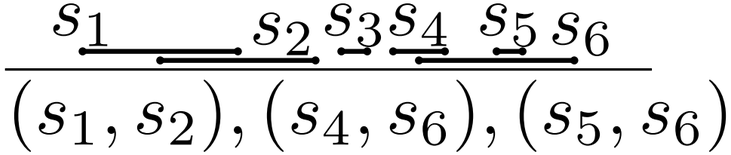
Quiz

Is the following algorithm output-sensitive?

Only test all pairs of line segments with overlapping y-coordinates for intersections



A: yes, but it is not correct



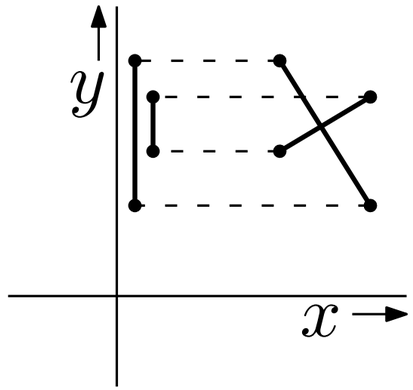
B: no, but testing segments with overlapping x- and overlapping y-coordinates would be.

C: no, and additionally asking for overlapping x-coordinates does not help.

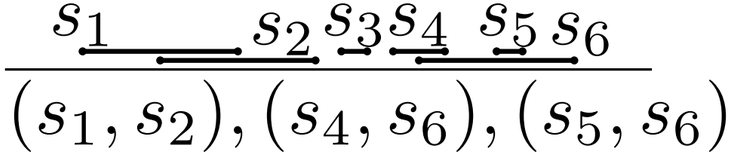
Quiz

Is the following algorithm output-sensitive?

Only test all pairs of line segments with overlapping y-coordinates for intersections



A: yes, but it is not correct



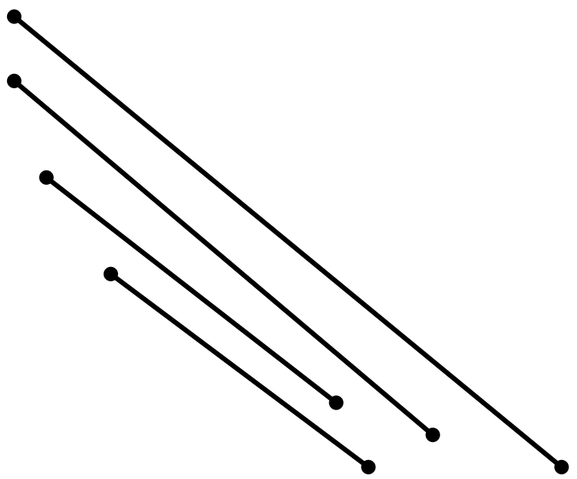
B: no, but testing segments with overlapping x- and overlapping y-coordinates would be.

C: no, and additionally asking for overlapping x-coordinates does not help.

Quiz

Is the following algorithm output-sensitive?

Only test all pairs of line segments with overlapping y-coordinates for intersections



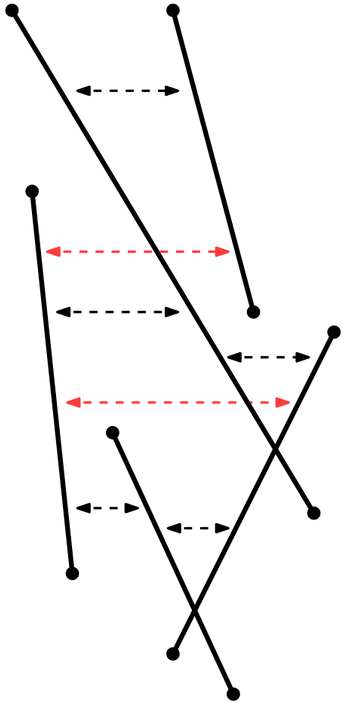
A: yes, but it is not correct

B: no, but testing segments with overlapping x- and overlapping y-coordinates would be.

C: no, and additionally asking for overlapping x-coordinates does not help.

Second attempt

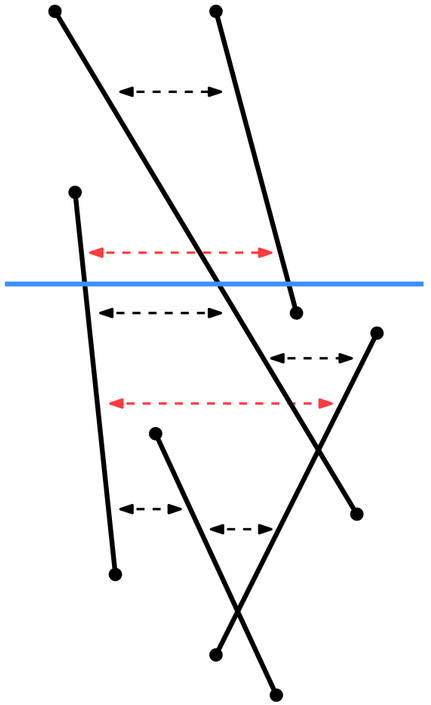
Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are *horizontal neighbors*)



Second attempt

Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are *horizontal neighbors*)

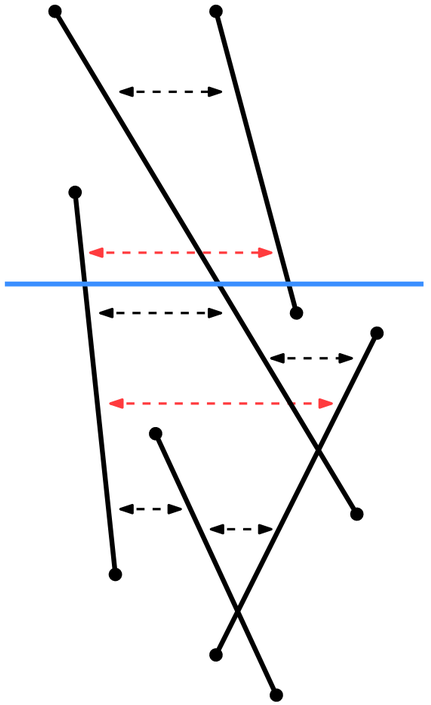
For any y-value: one-dimensional problem



Second attempt

Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are *horizontal neighbors*)

For any y-value: one-dimensional problem

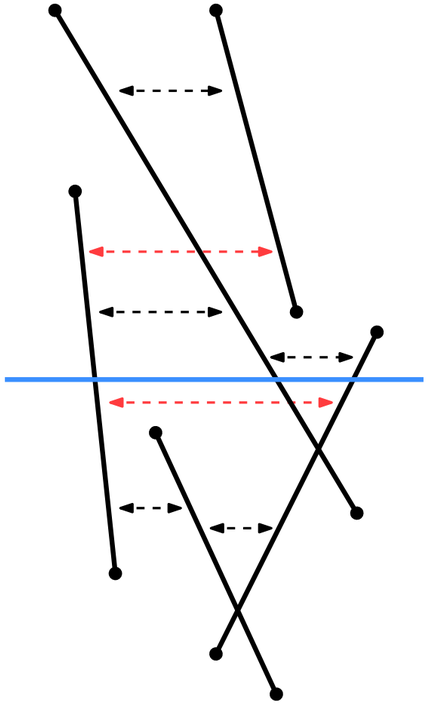


How can we efficiently solve it for all y-values?

Second attempt

Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are *horizontal neighbors*)

For any y-value: one-dimensional problem

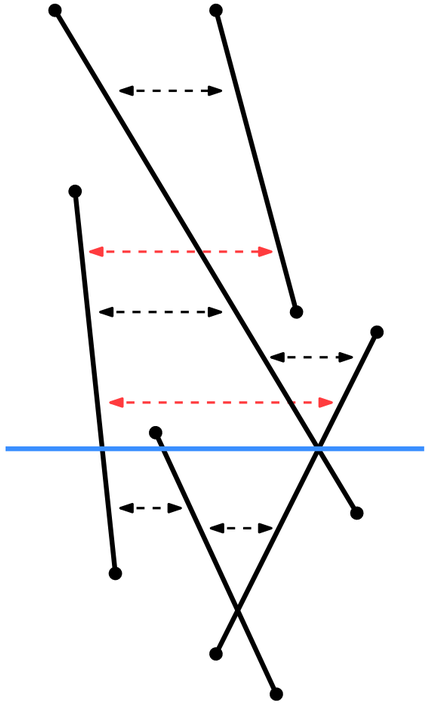


How can we efficiently solve it for all y-values?

Second attempt

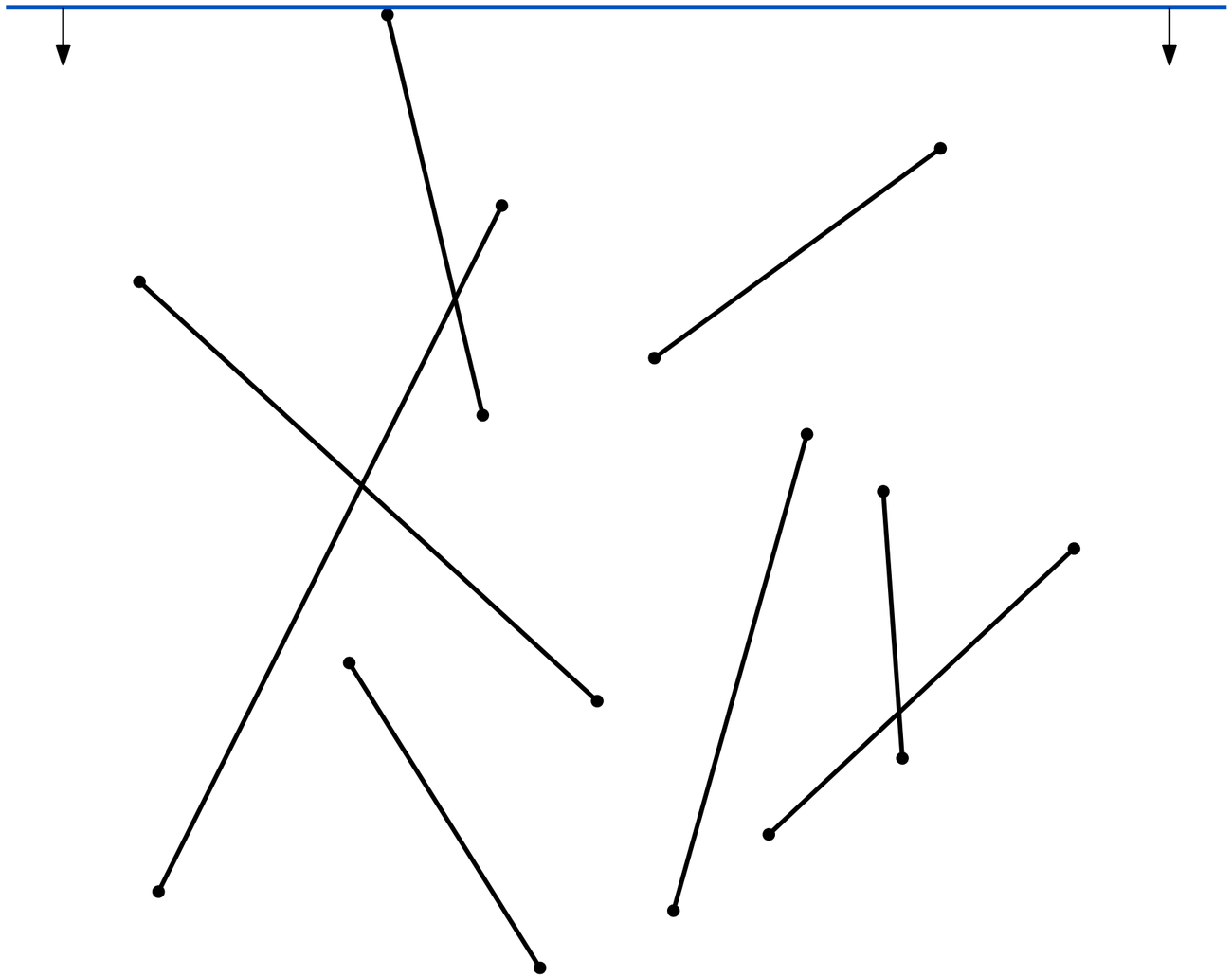
Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are *horizontal neighbors*)

For any y-value: one-dimensional problem

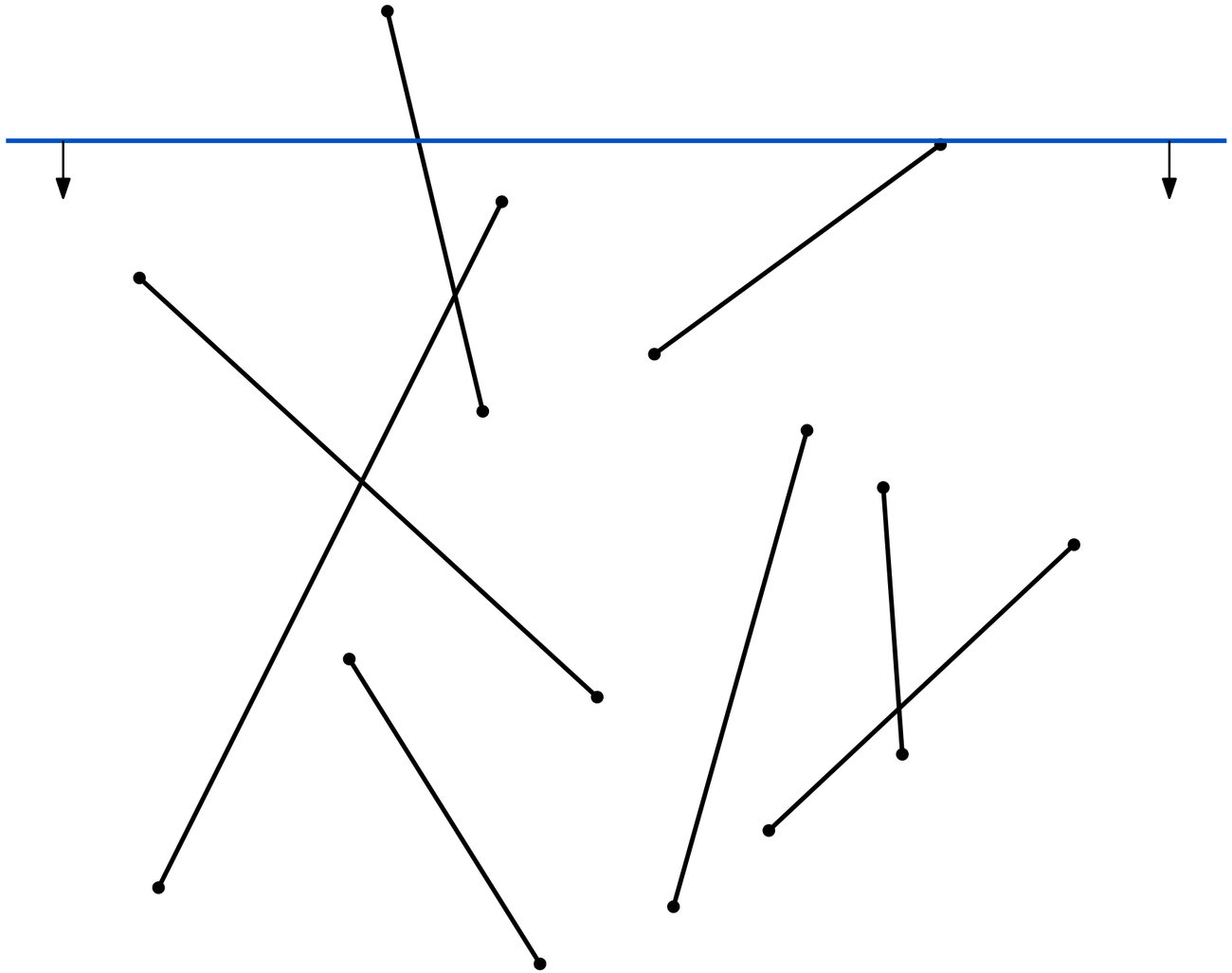


How can we efficiently solve it for all y-values?

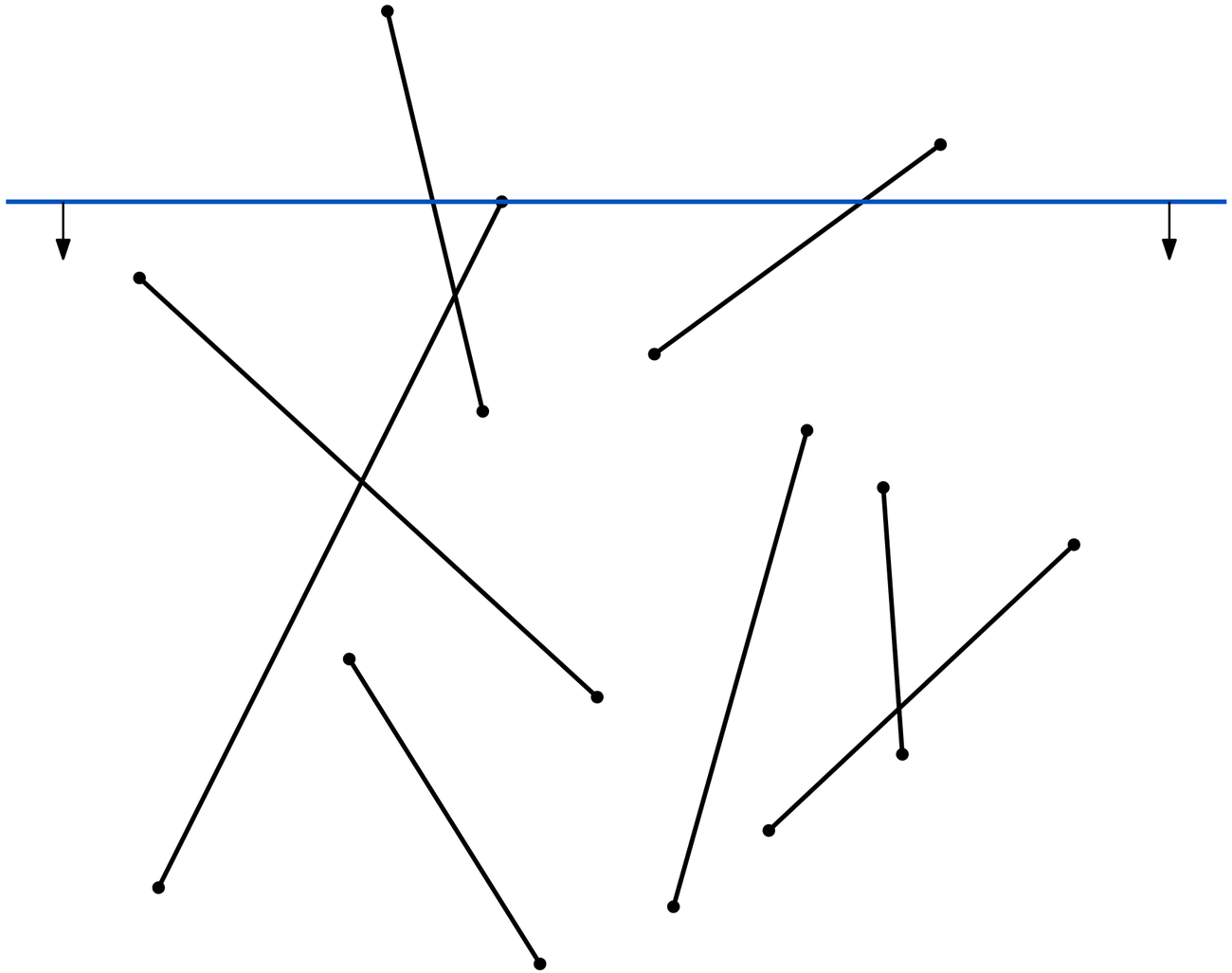
Sweep



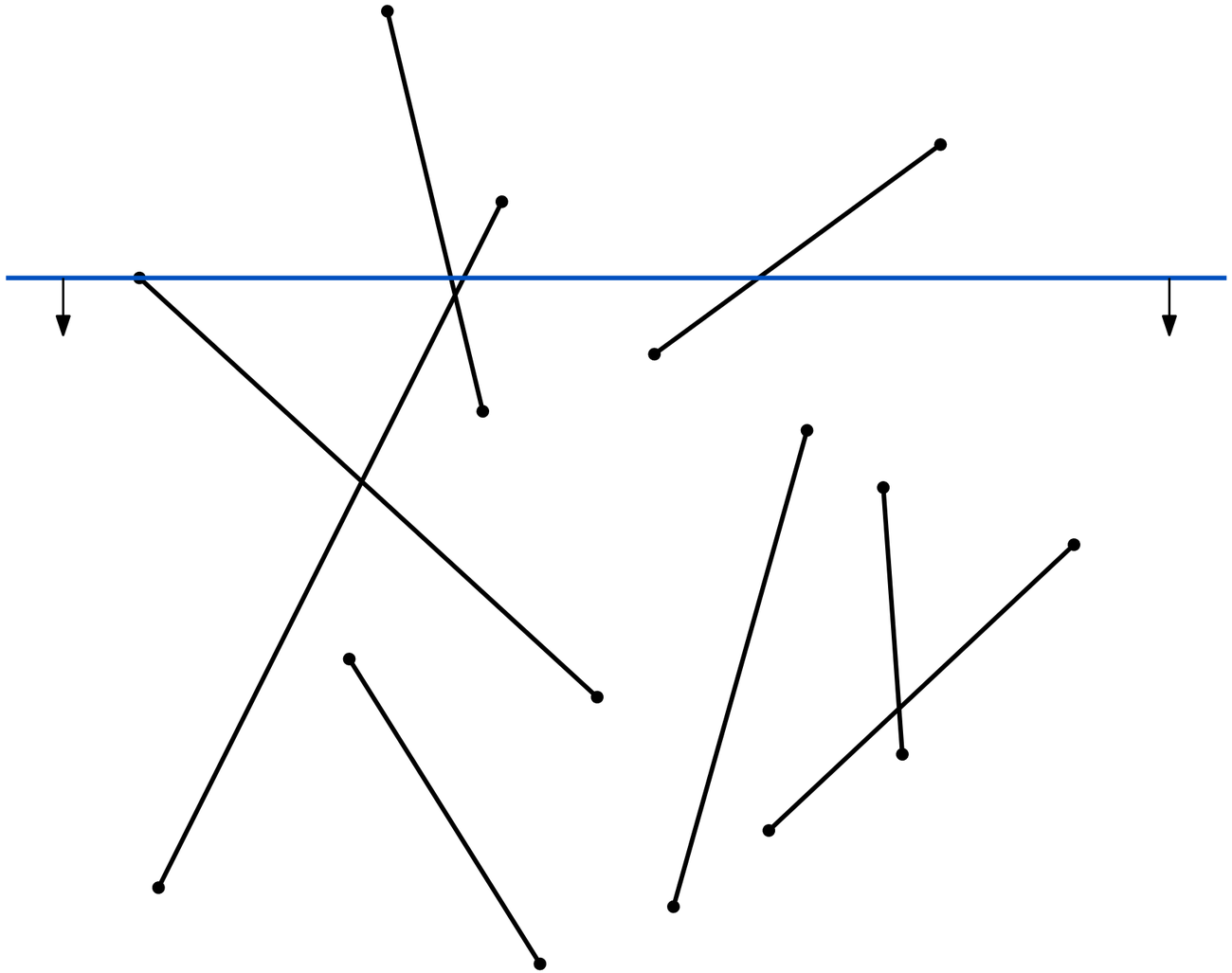
Sweep



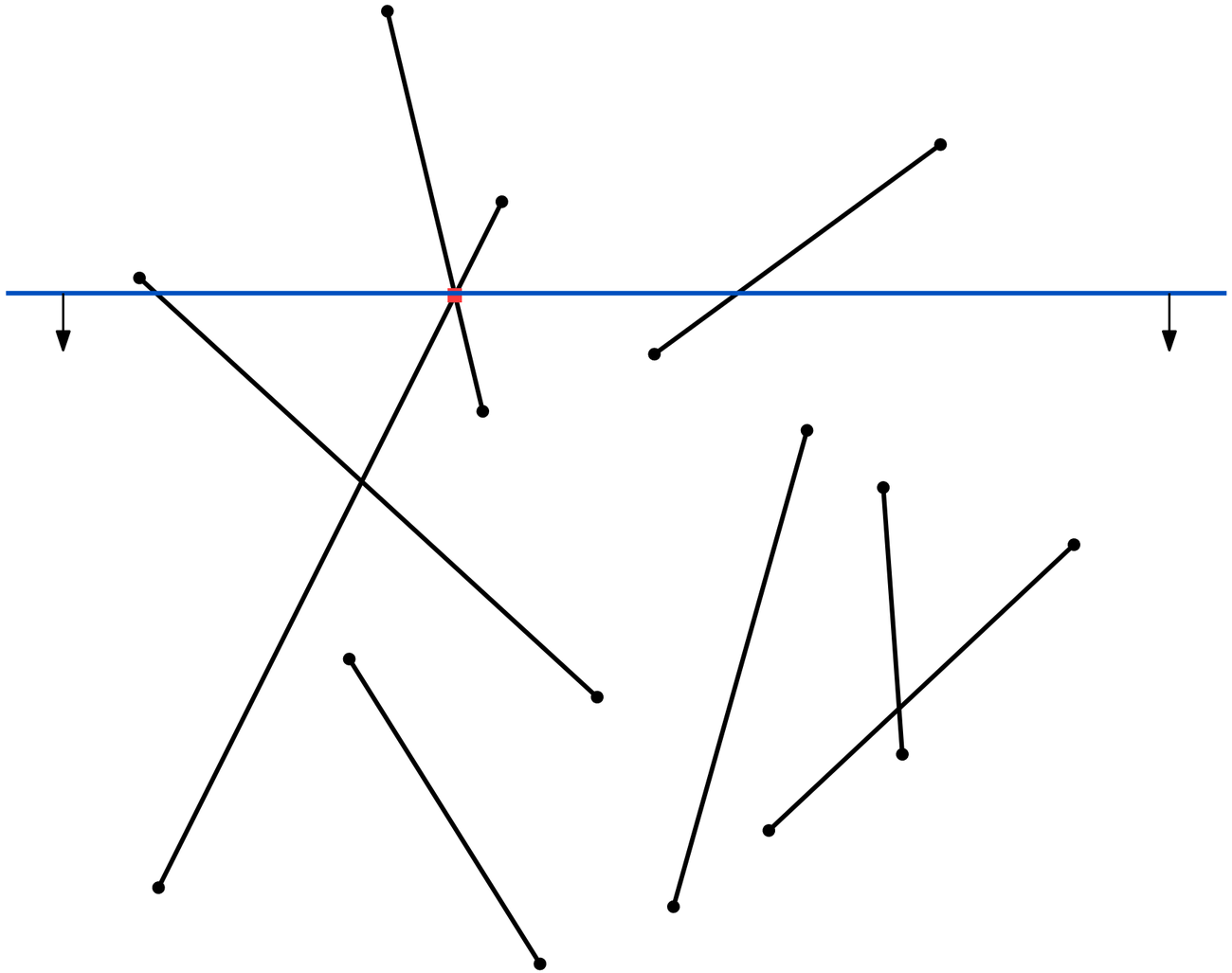
Sweep



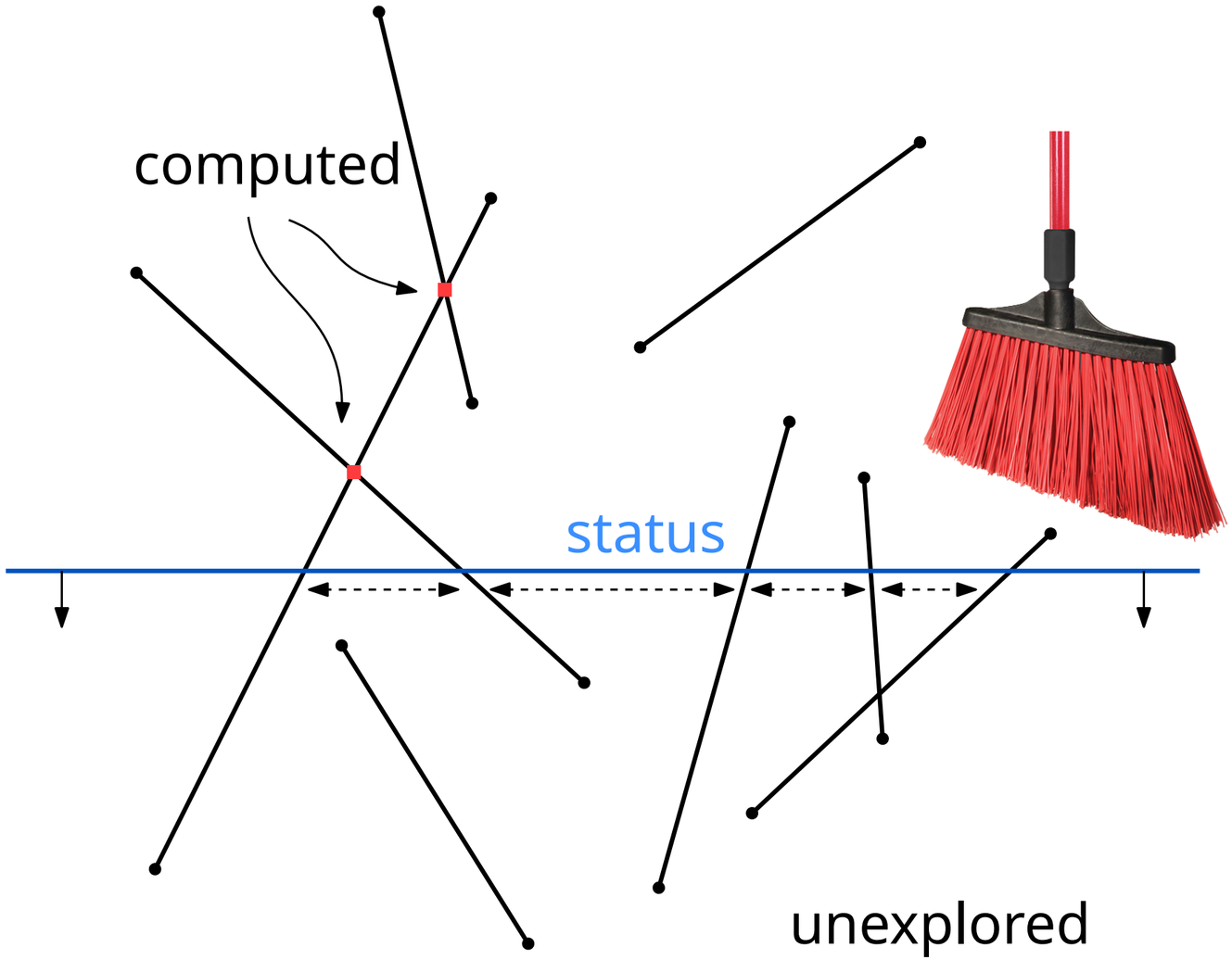
Sweep



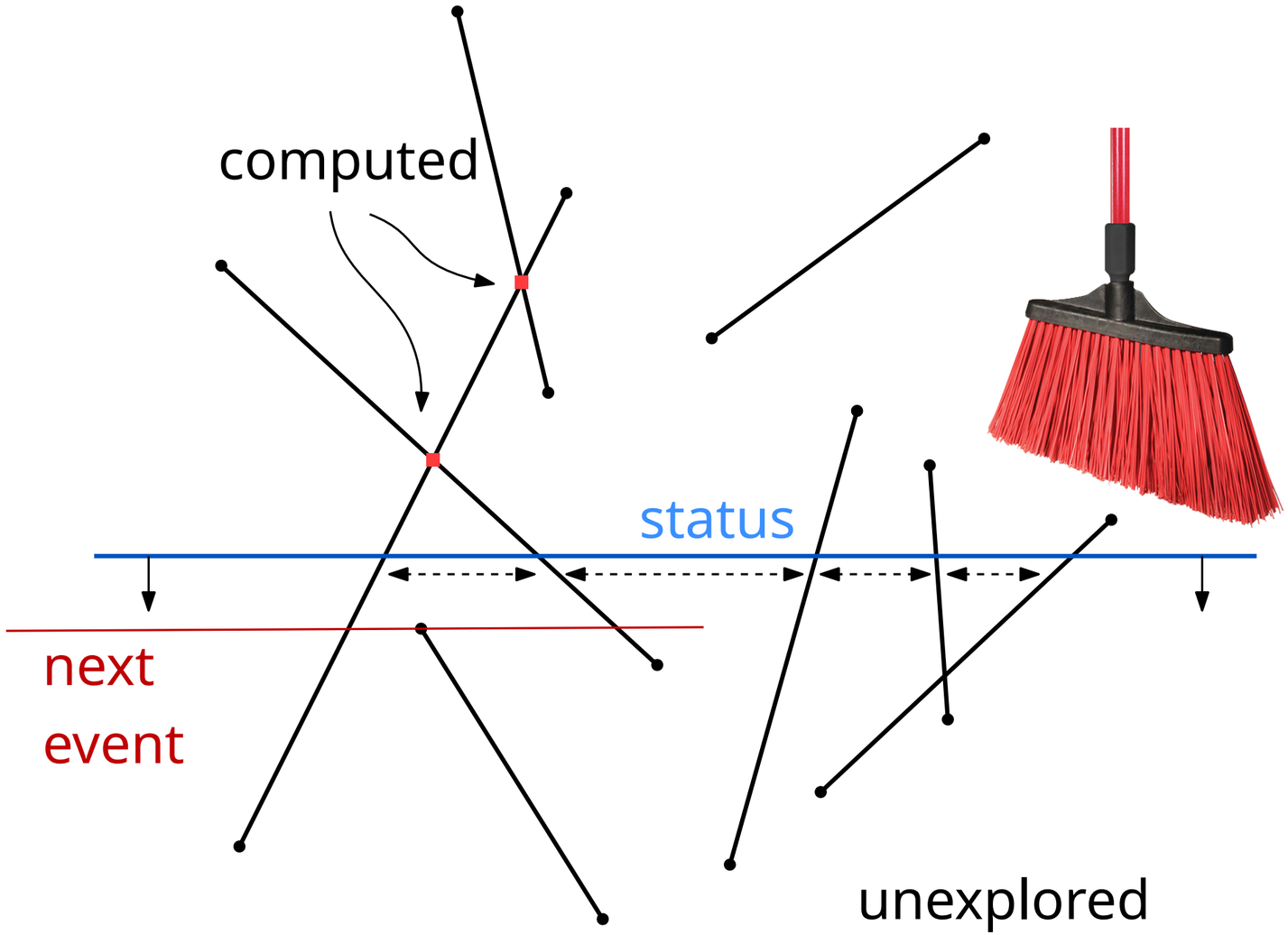
Sweep



Sweep

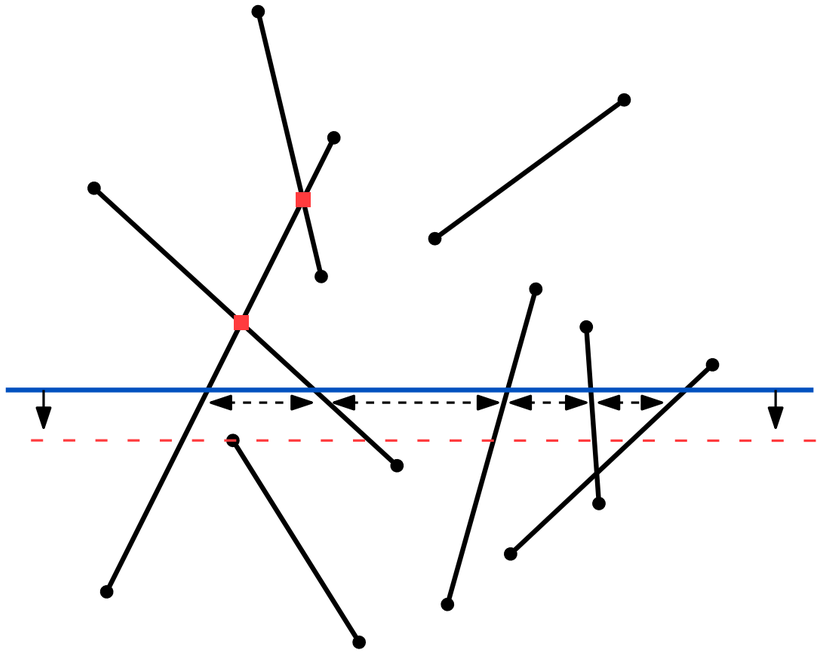


Sweep



Plane sweep

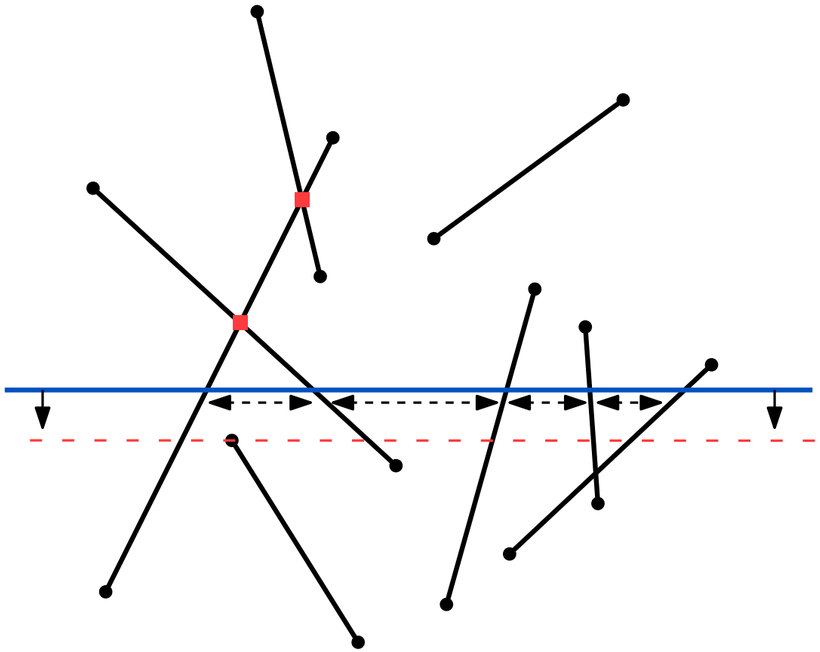
The plane sweep technique: Imagine a horizontal line passing over the plane from top to bottom, solving the problem as it moves



Plane sweep

The plane sweep technique: Imagine a horizontal line passing over the plane from top to bottom, solving the problem as it moves

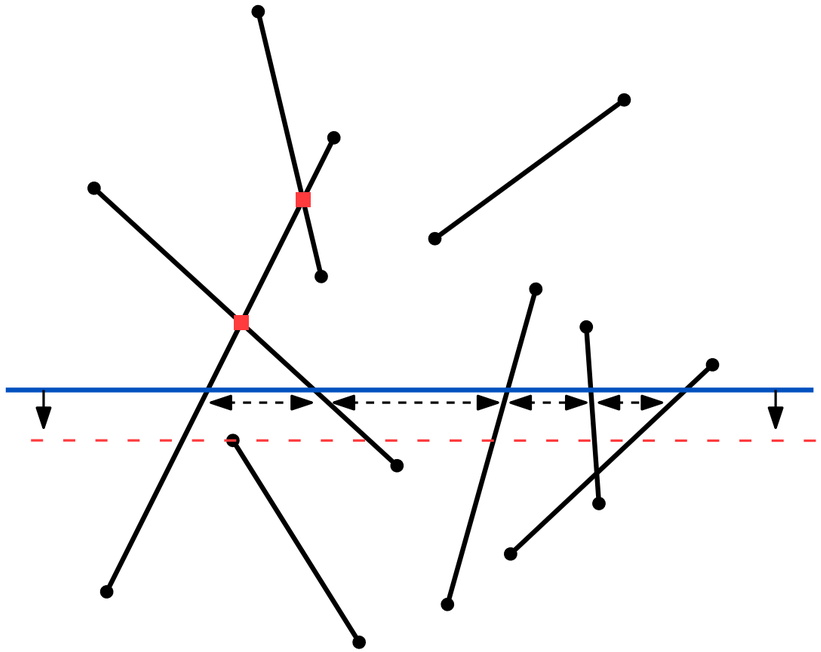
- The algorithm stores the relevant situation at the current position of the sweep line $\Rightarrow$ status



Plane sweep

The plane sweep technique: Imagine a horizontal line passing over the plane from top to bottom, solving the problem as it moves

- The algorithm stores the relevant situation at the current position of the sweep line $\Rightarrow$ status
- The sweep line stops and the algorithm computes at certain positions $\Rightarrow$ events

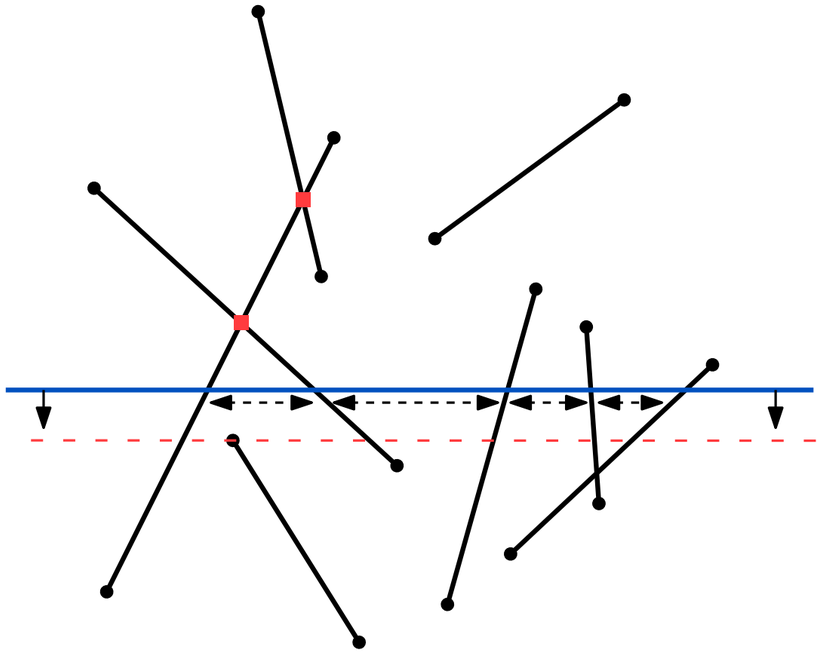


Plane sweep

The plane sweep technique: Imagine a horizontal line passing over the plane from top to bottom, solving the problem as it moves

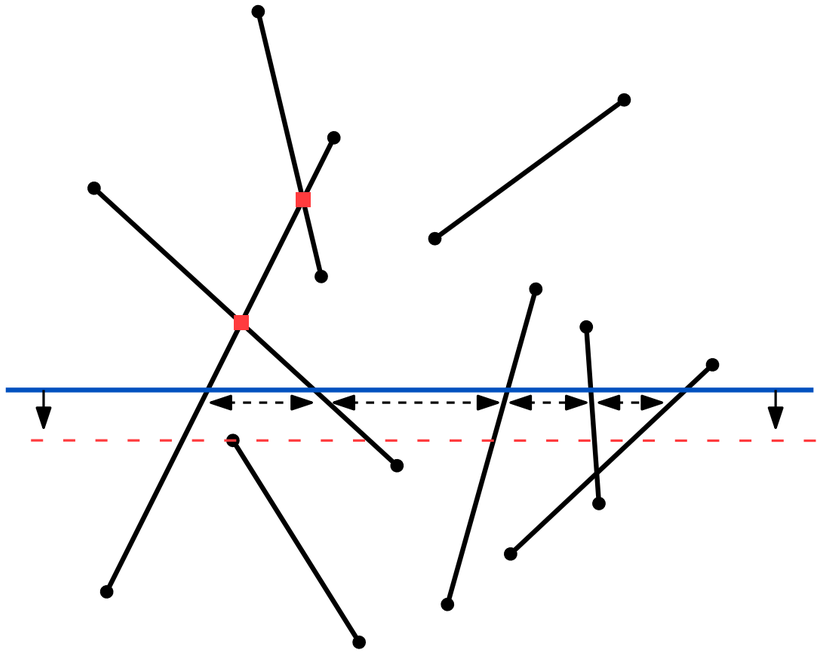
- The algorithm stores the relevant situation at the current position of the sweep line $\Rightarrow$ status
- The sweep line stops and the algorithm computes at certain positions $\Rightarrow$ events

- The algorithm knows everything it needs to know above the sweep line, and found all intersection points



Status and events for line segment intersection

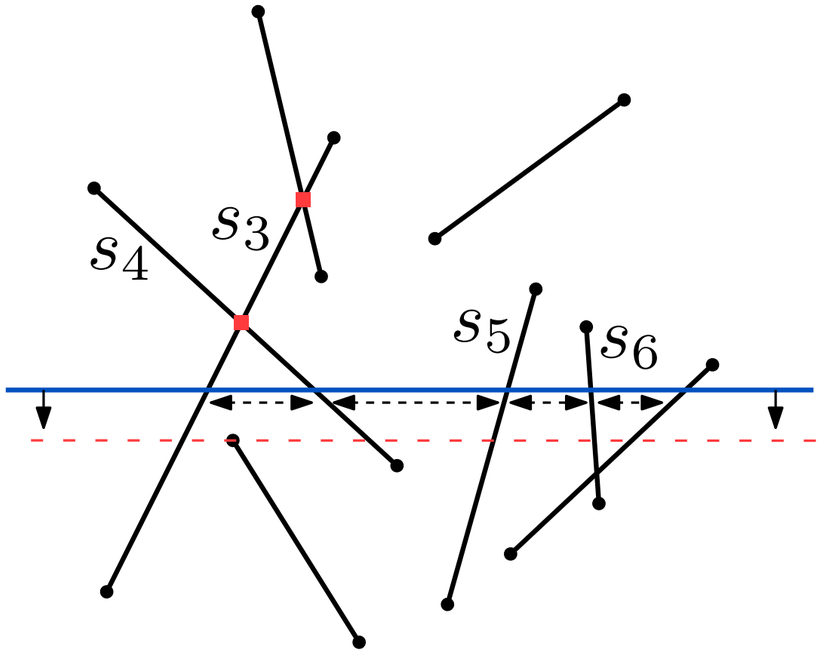
status: set of line segments intersecting the sweep line, ordered from left to right



Status and events for line segment intersection

status: set of line segments intersecting the sweep line, ordered from left to right

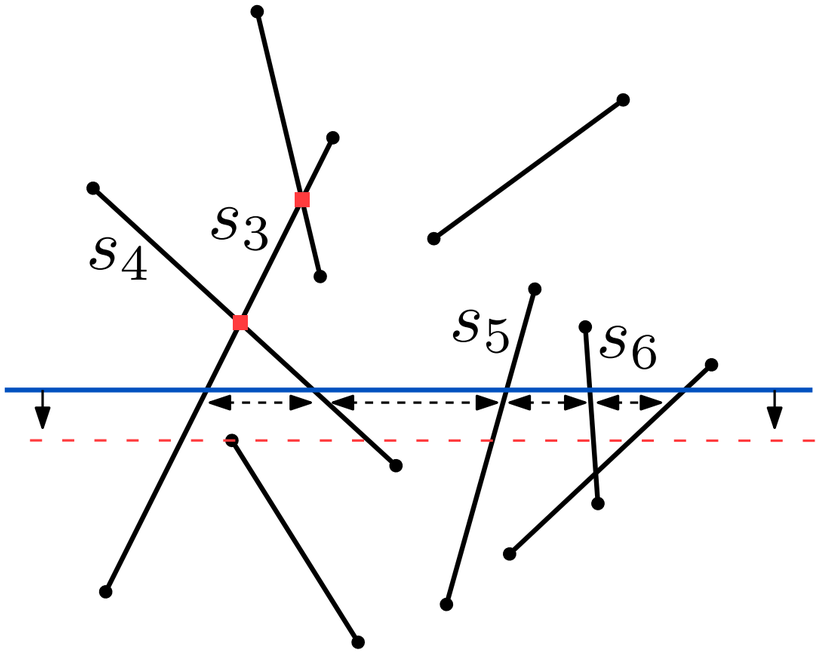
s_3, s_4, s_5, s_6



Status and events for line segment intersection

status: set of line segments intersecting the sweep line, ordered from left to right

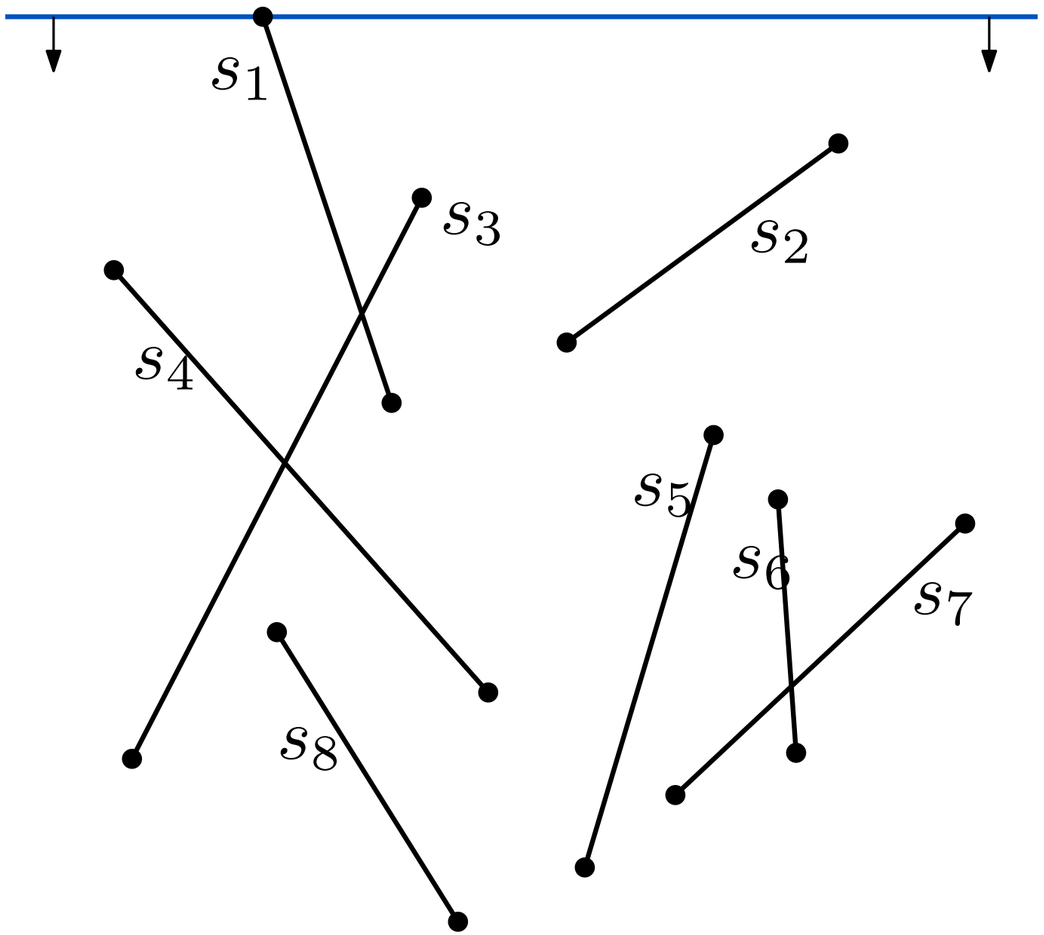
s_3, s_4, s_5, s_6



events occur when the *status changes*, and when *output is generated*

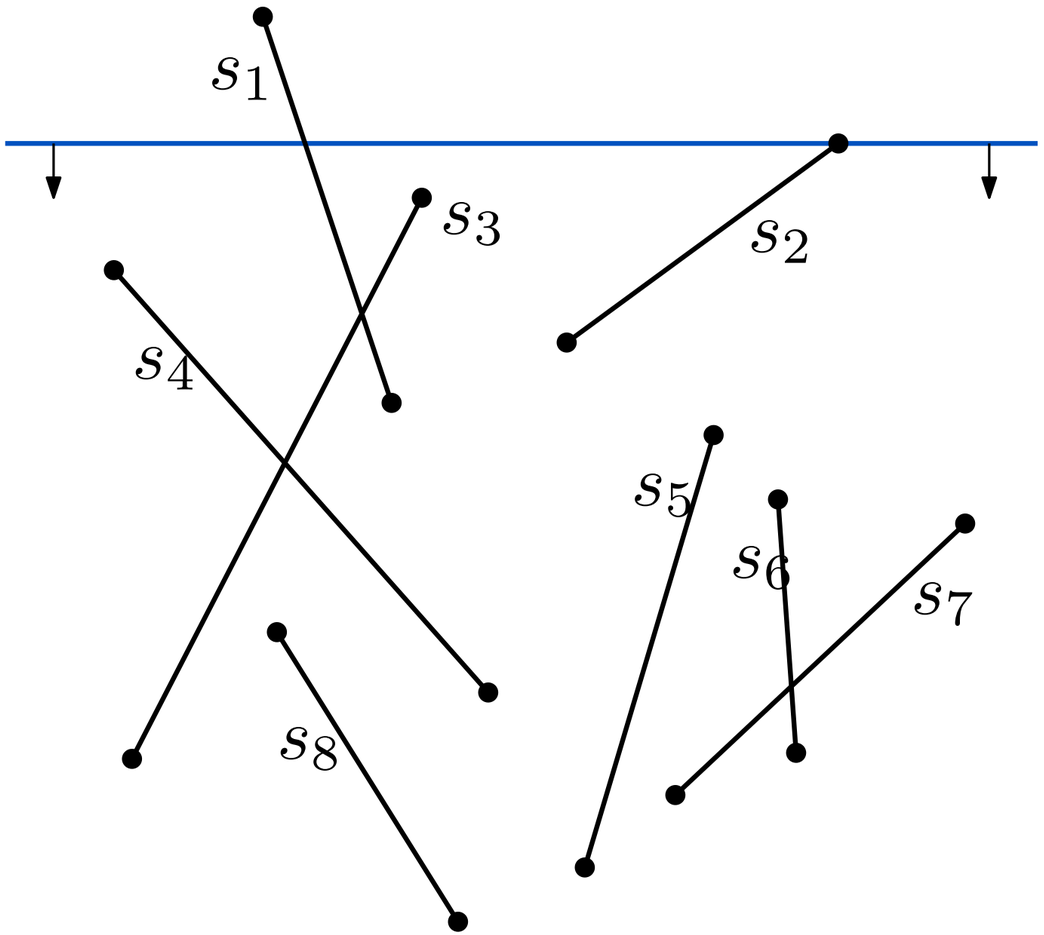
event $\approx$ interesting y-coordinate

Line sweep example



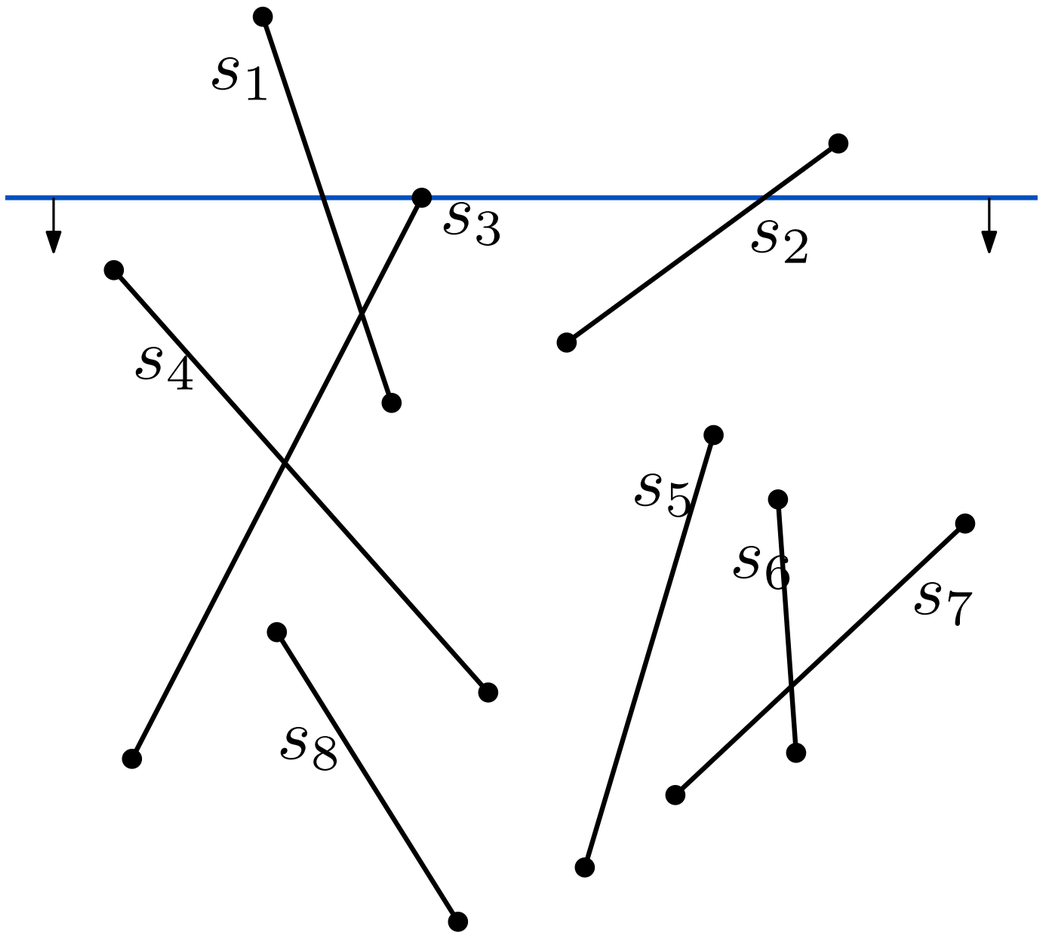
add s_1

Line sweep example



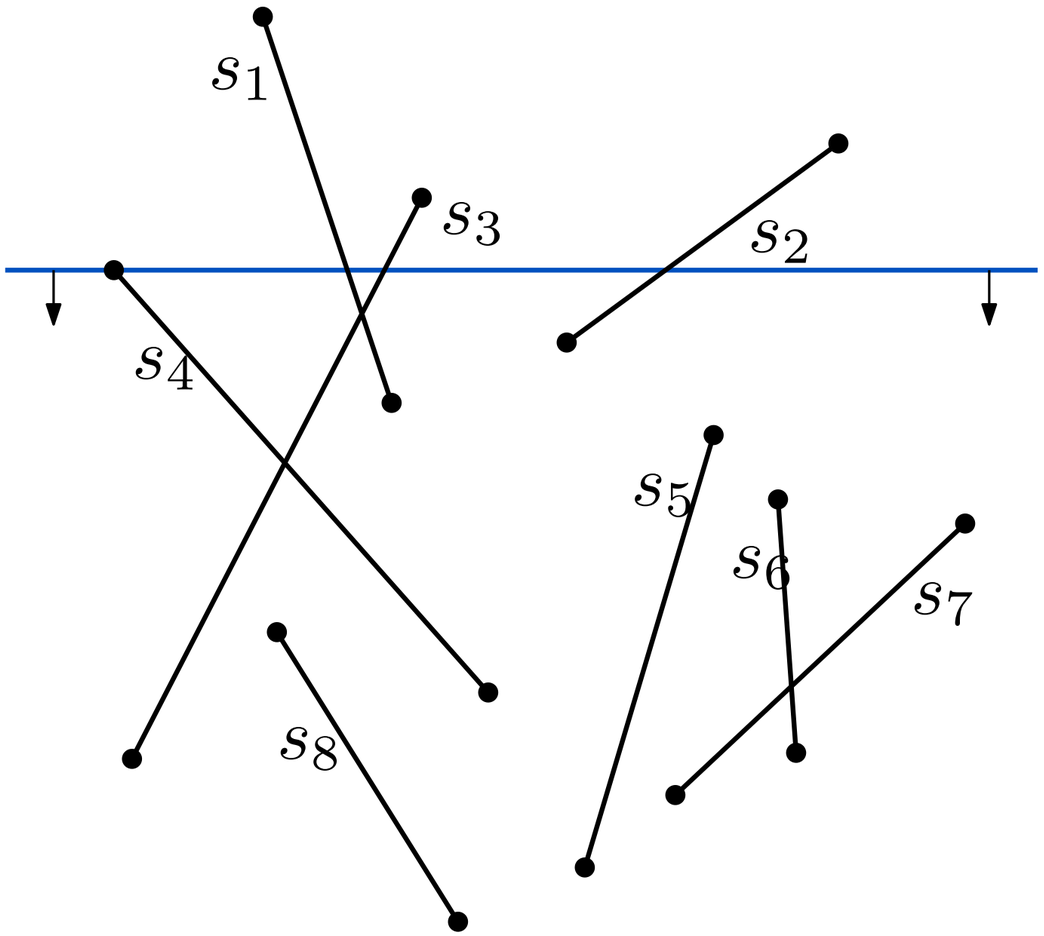
add s_2 after s_1

Line sweep example



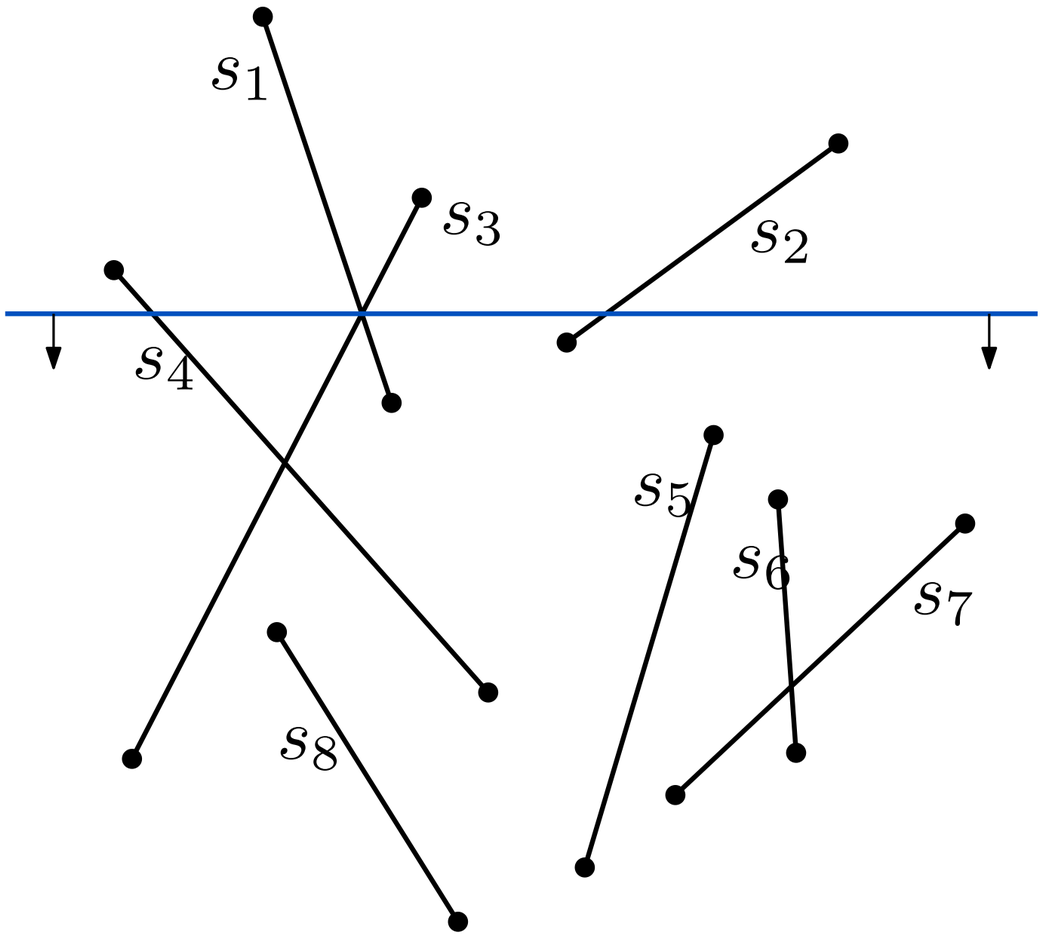
add s_3 between s_1 and s_2

Line sweep example



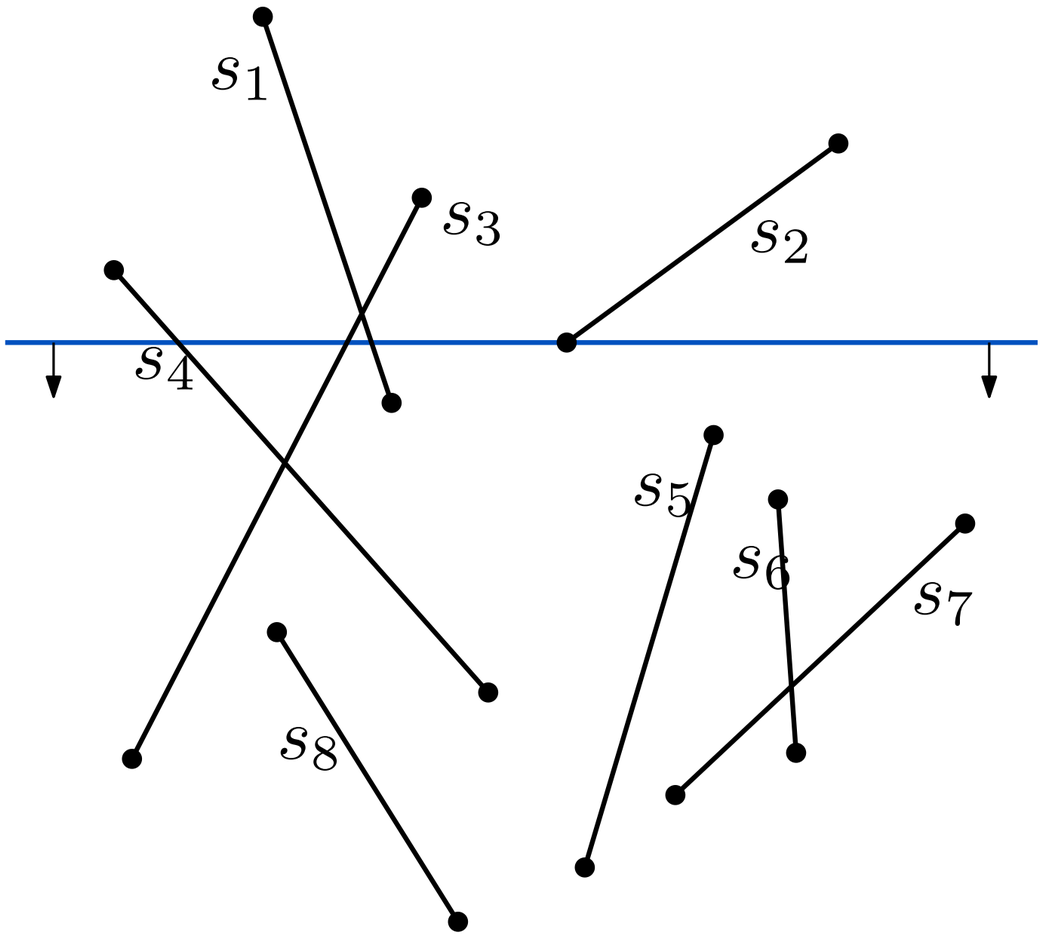
add s_4 before s_1

Line sweep example



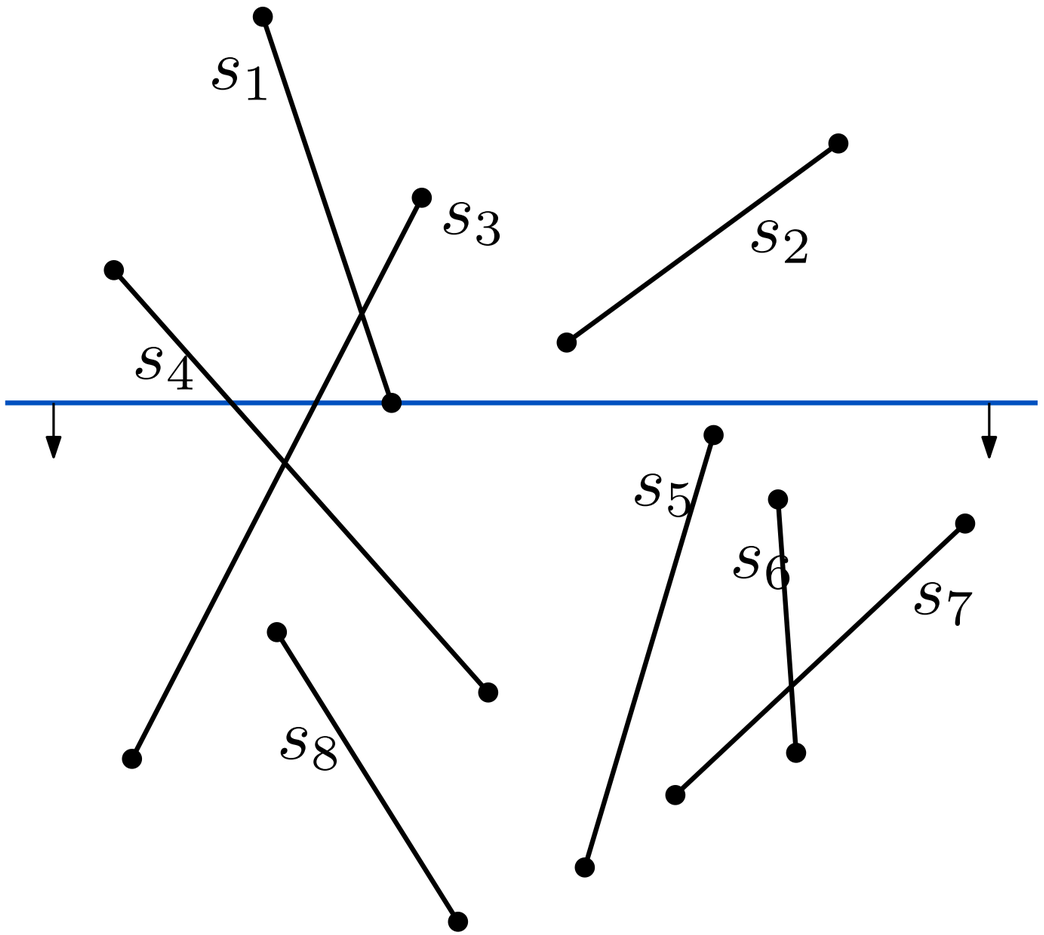
report intersection (s_1, s_3);
swap s_1 and s_3

Line sweep example



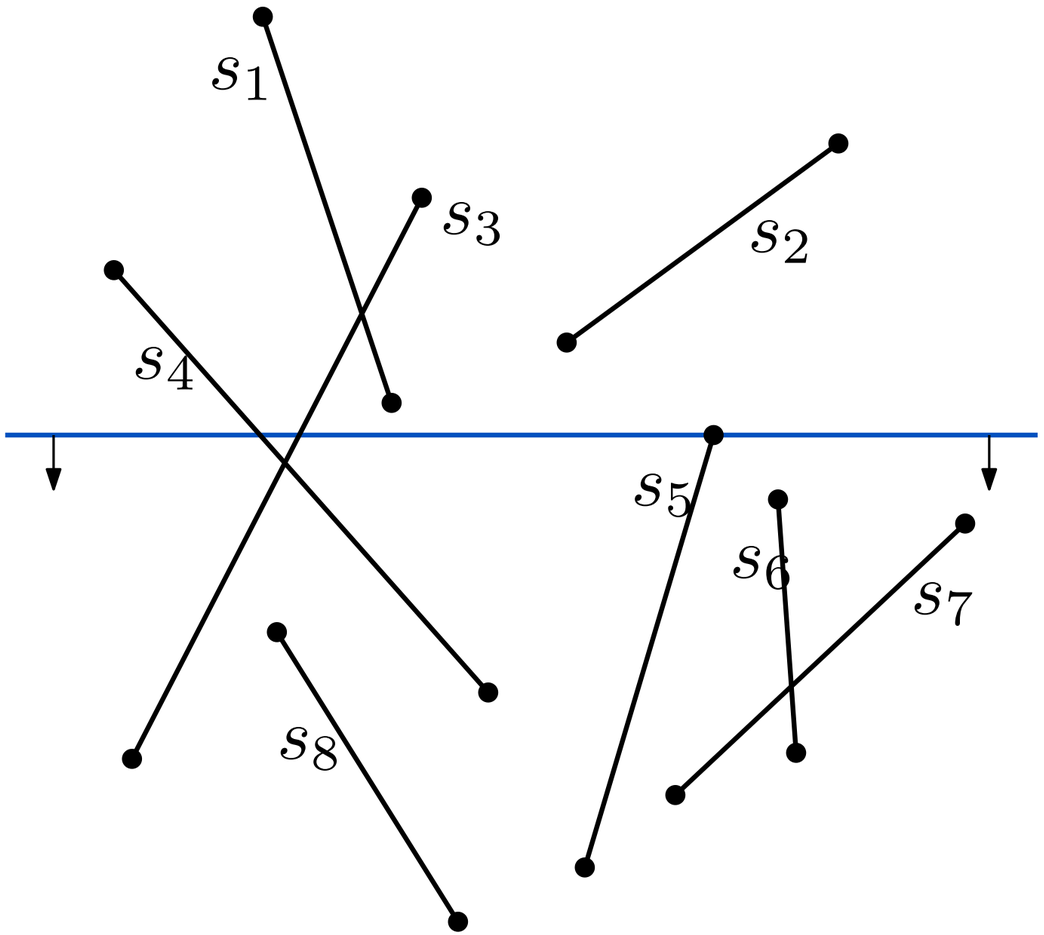
remove s_2

Line sweep example



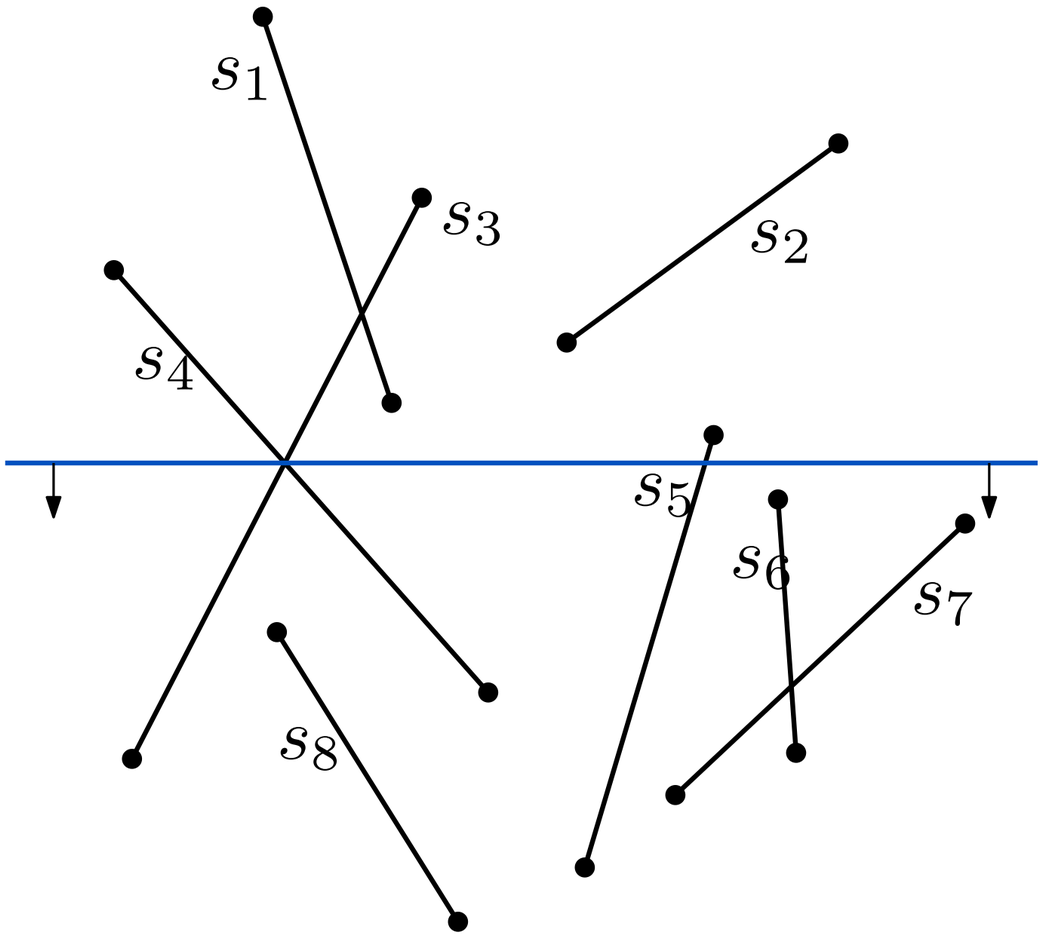
remove s_1

Line sweep example



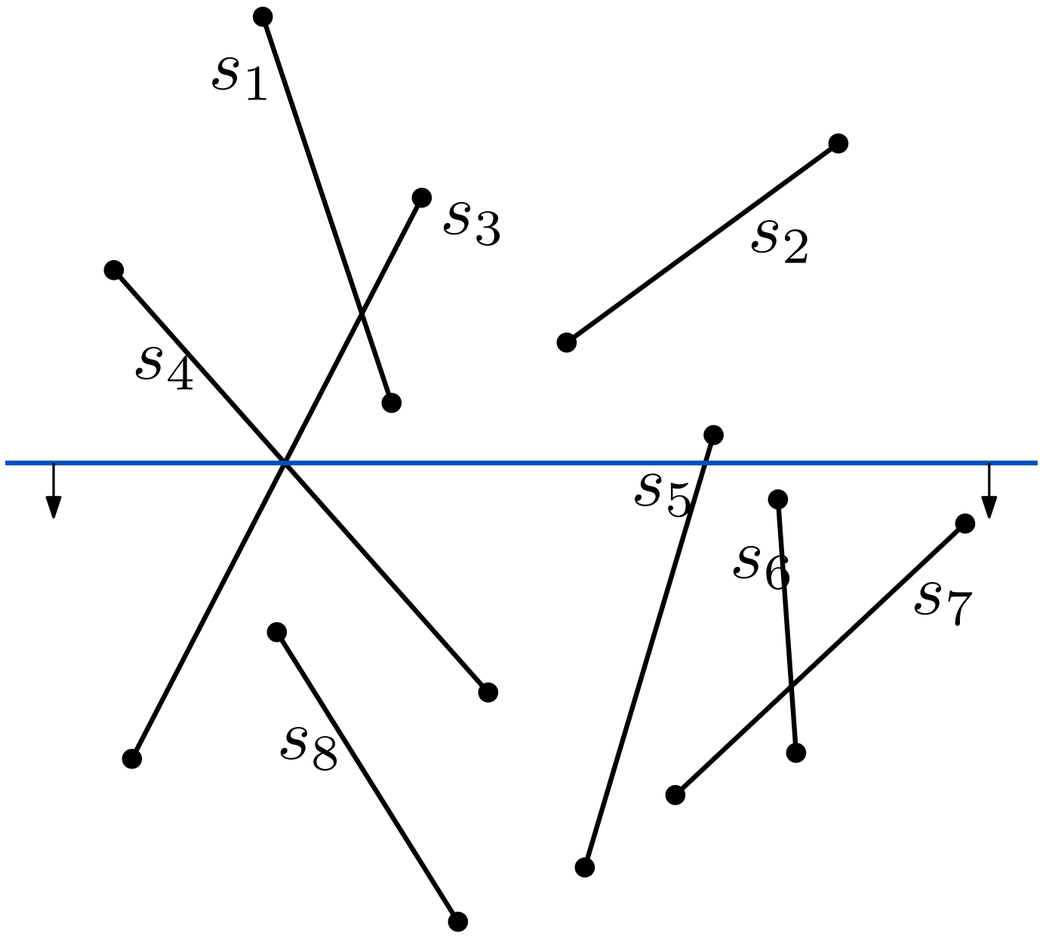
add s_5

Line sweep example



report intersection (s_3, s_4);
swap s_3 and s_4

Line sweep example

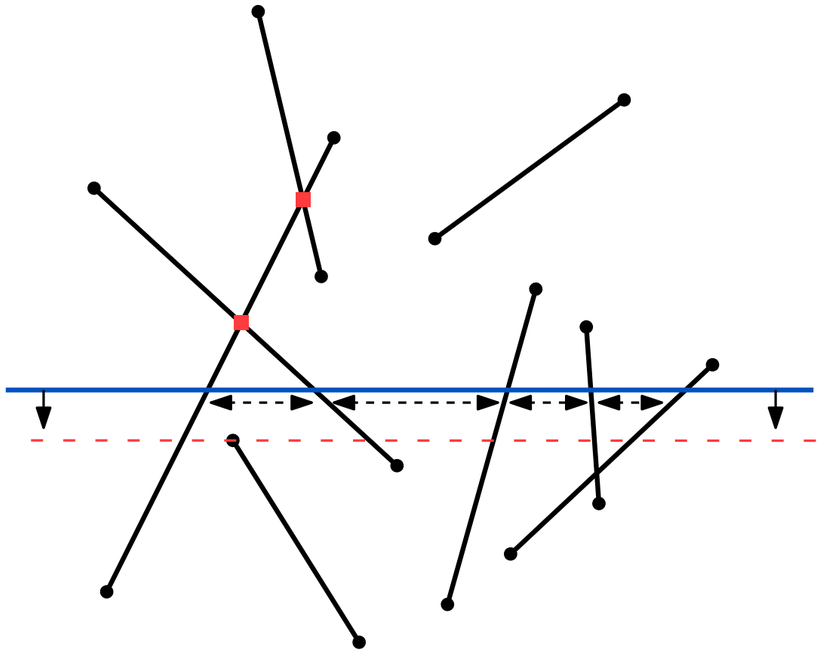


report intersection (s_3, s_4);
swap s_3 and s_4

…and so on …

The events

When do the events happen?

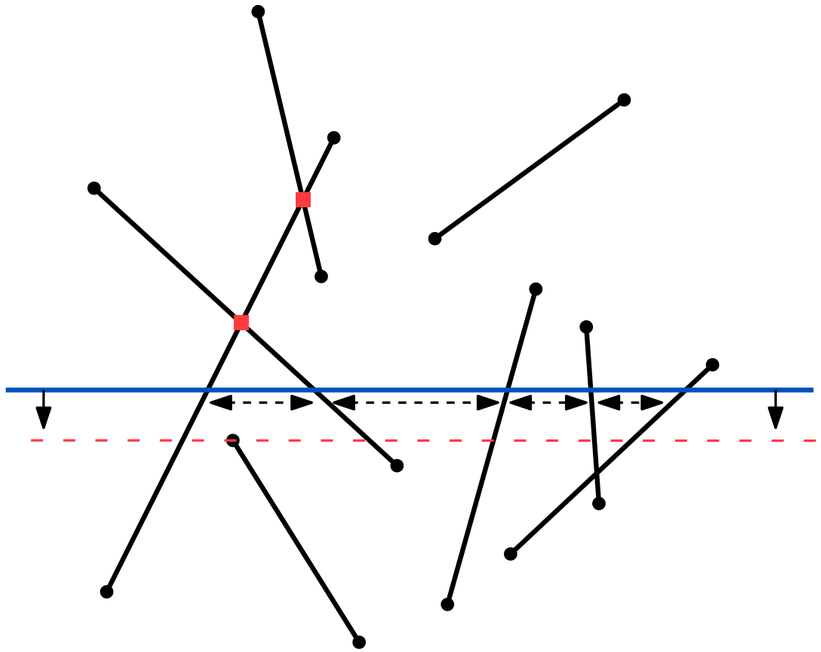


The events

When do the events happen?

When the sweep line is

- at an upper endpoint of a line segment
- at a lower endpoint of a line segment
- at an intersection point of line segments

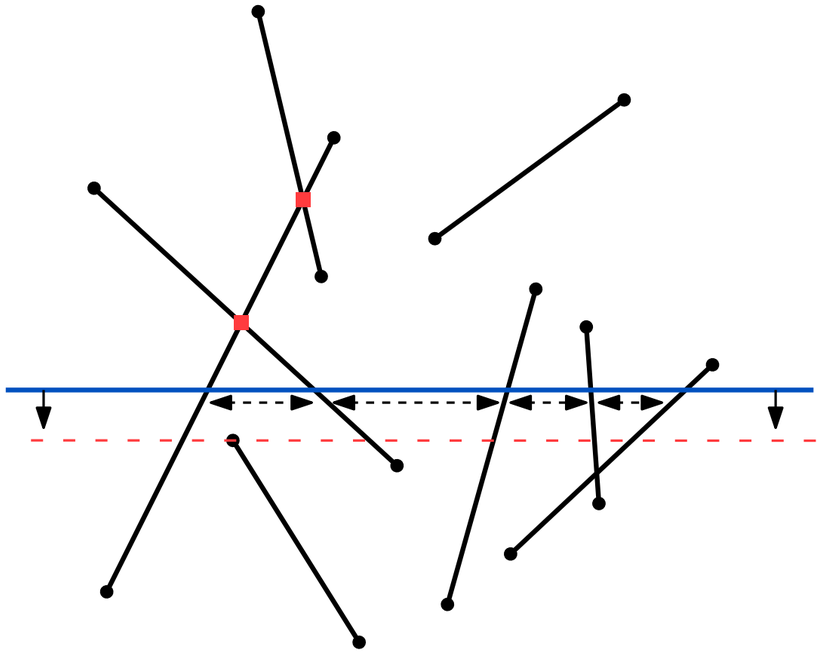


The events

When do the events happen?

When the sweep line is

- at an upper endpoint of a line segment
- at a lower endpoint of a line segment
- at an intersection point of line segments



At each type, the status changes; at the third type output is found too

At no other y-coordinates the status changes

Assume no degenerate cases

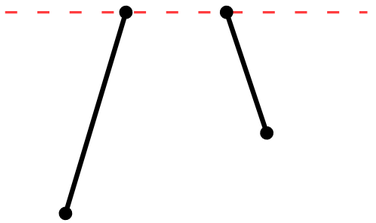
We will at first exclude degenerate cases:

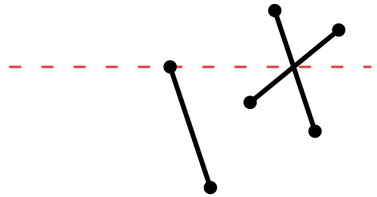
- No two events (endpoints/intersection points) have the same y-coordinate

Assume no degenerate cases

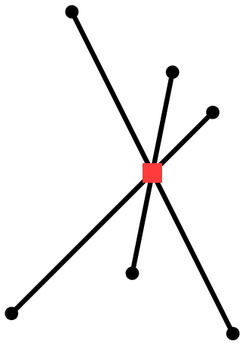
We will at first exclude degenerate cases:

- No two events (endpoints/intersection points) have the same y-coordinate





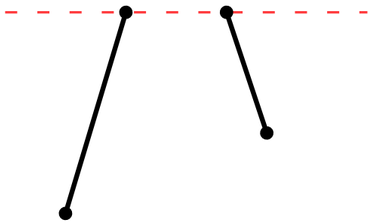
- No more than two line segments intersect in a point

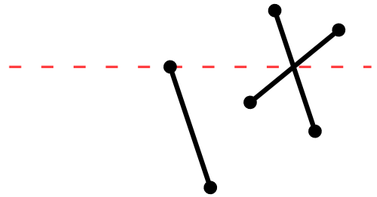


Assume no degenerate cases

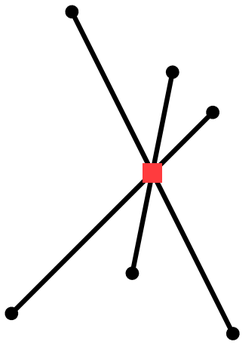
We will at first exclude degenerate cases:

- No two events (endpoints/intersection points) have the same y-coordinate





- No more than two line segments intersect in a point
- …

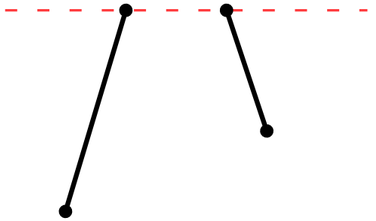


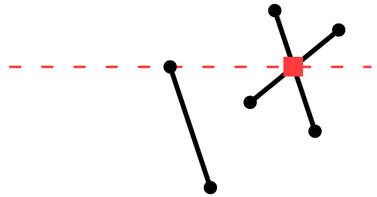
Question: Are there more degenerate cases?

Assume no degenerate cases

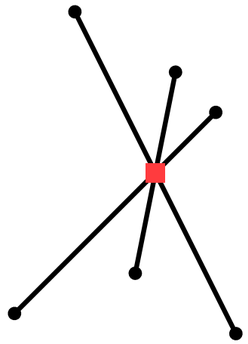
We will at first exclude degenerate cases:

- No two events (endpoints/intersection points) have the same y-coordinate

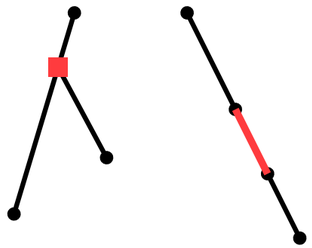




- No more than two line segments intersect in a point



- No intersections include endpoints



Quiz

Assume we want to compute the intersection of two convex polygons using a plane sweep algorithm. What is the complexity of the status?

A: $O(n)$

B: $O(1)$

C: $O(n^2)$

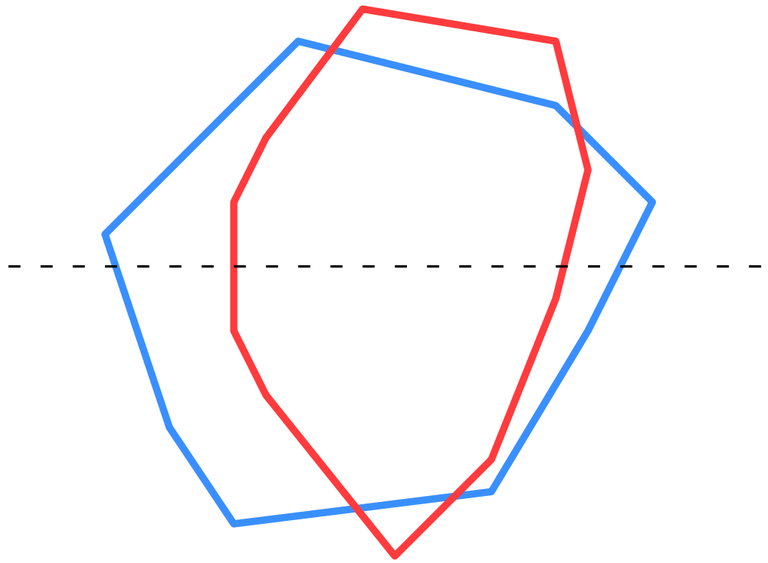
Quiz

Assume we want to compute the intersection of two convex polygons using a plane sweep algorithm. What is the complexity of the status?

A: $O(n)$

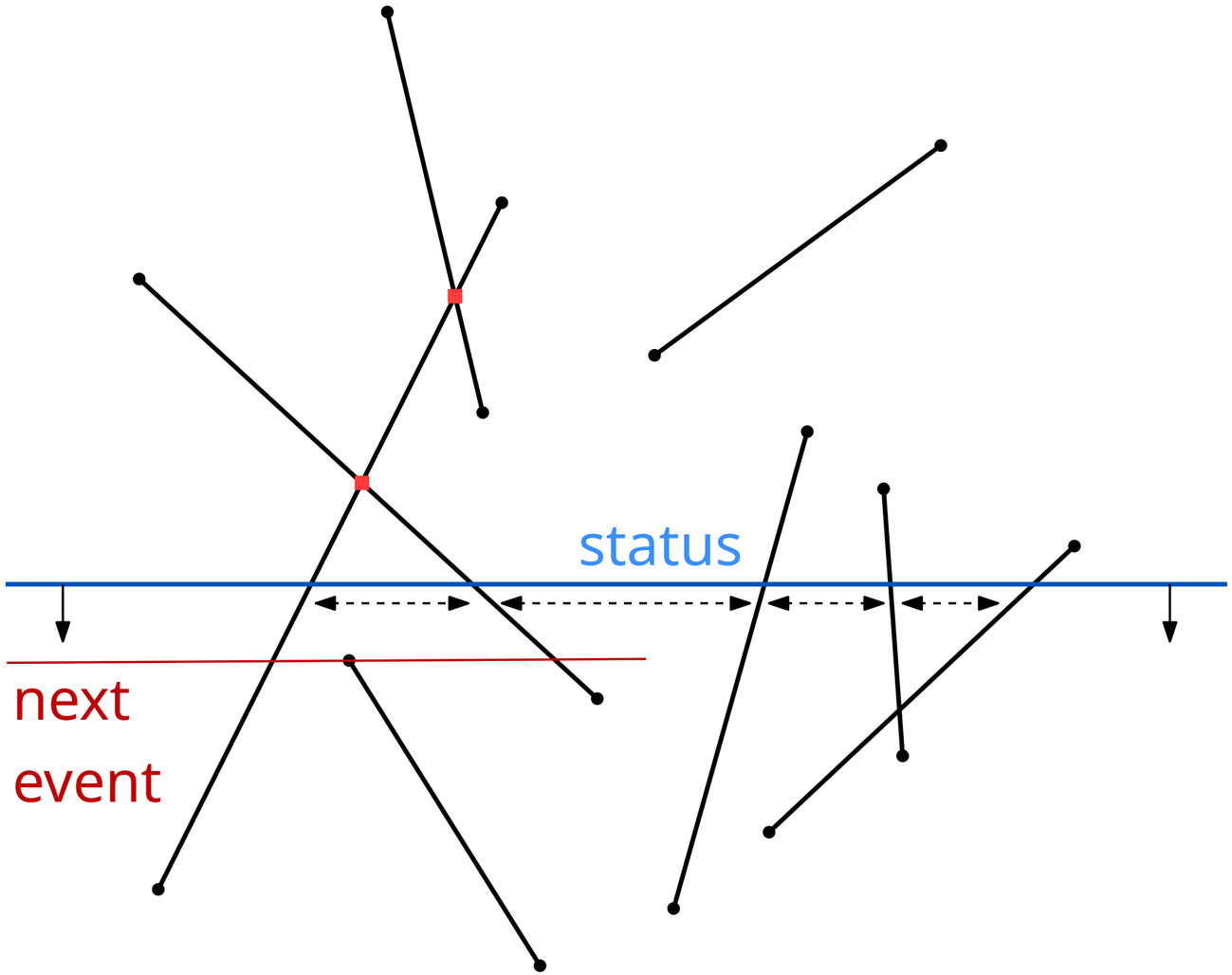
B: $O(1)$

C: $O(n^2)$



The status stores the currently intersected boundaries: ≤ 4

Summary



Ingredients for

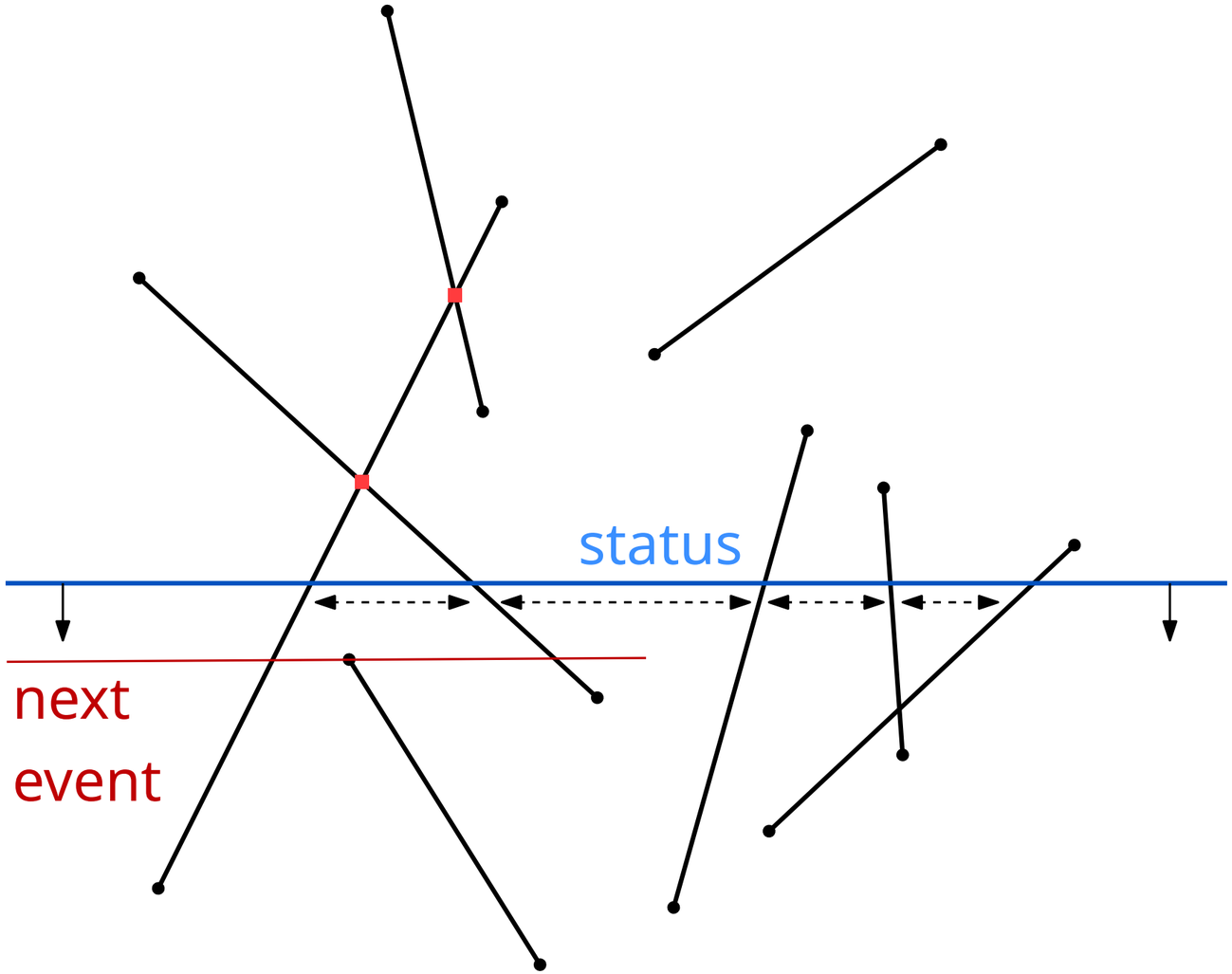
Plane Sweep technique:

- status
- events

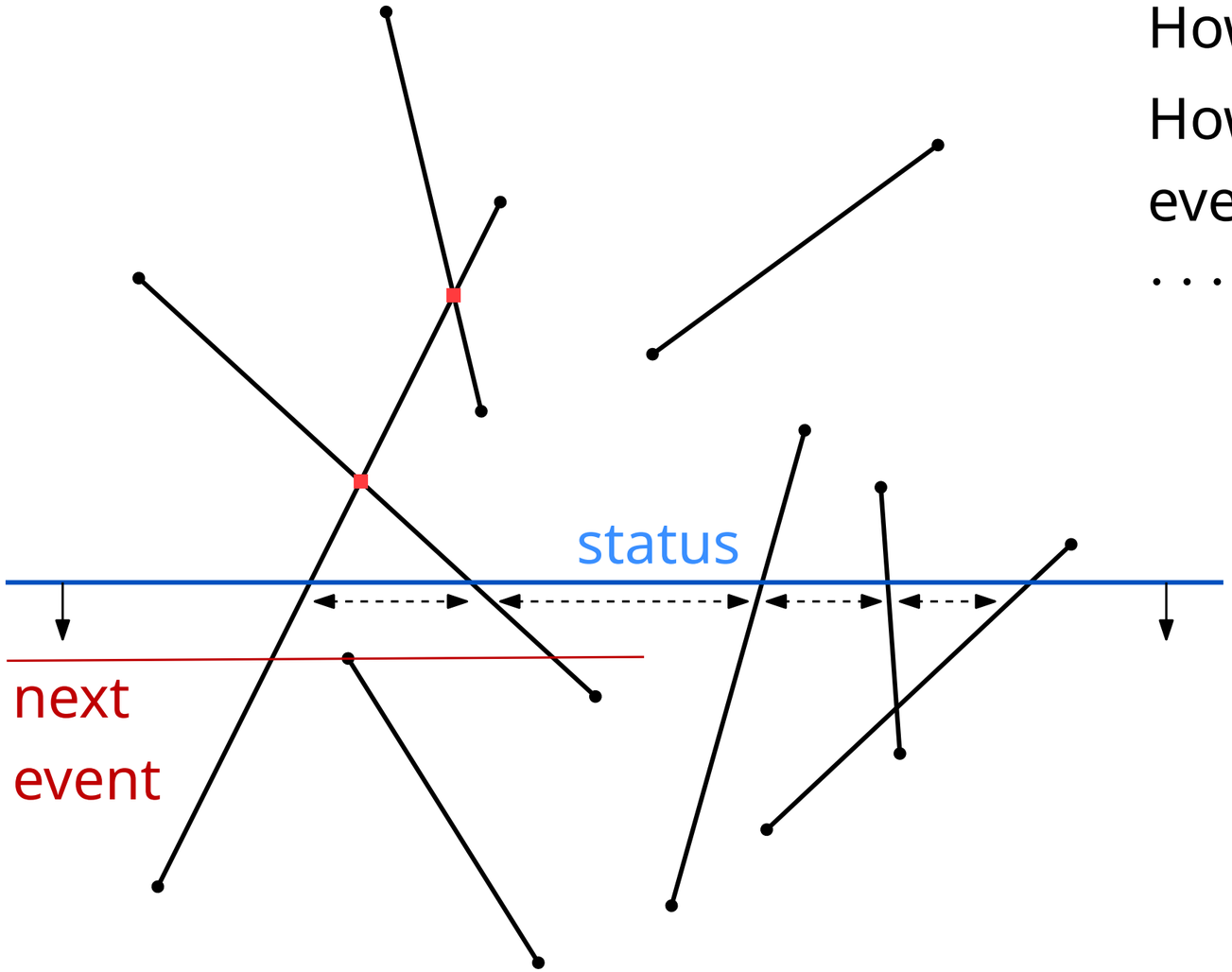
Plane Sweep Algorithm for Line Segment Intersection

data structures

Status and Events



Status and Events

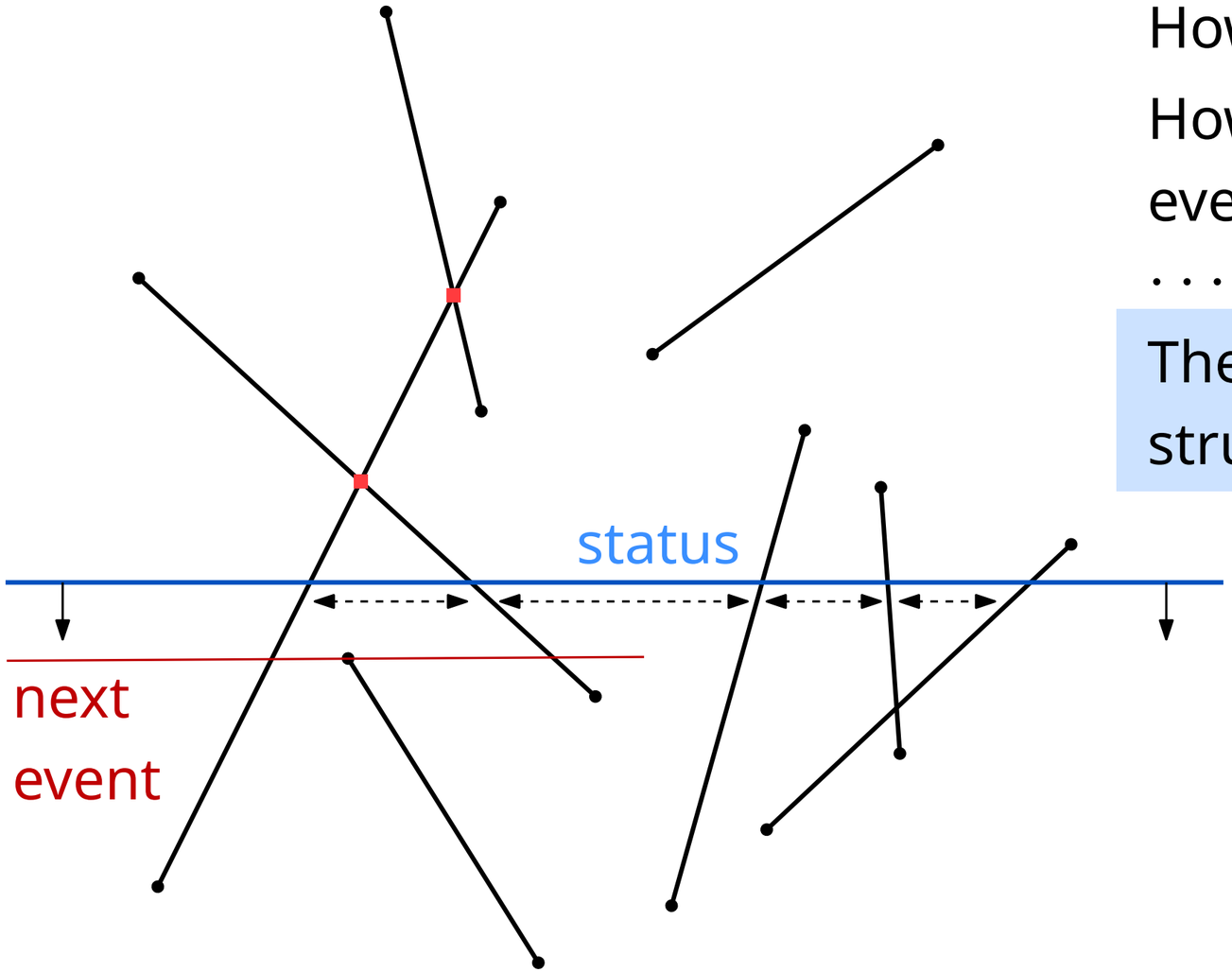


How do we find the next event?

How do we update the status at the event?

. . .

Status and Events



How do we find the next event?

How do we update the status at the event?

. . .

These operations require suitable data structures!

Event list and status structure

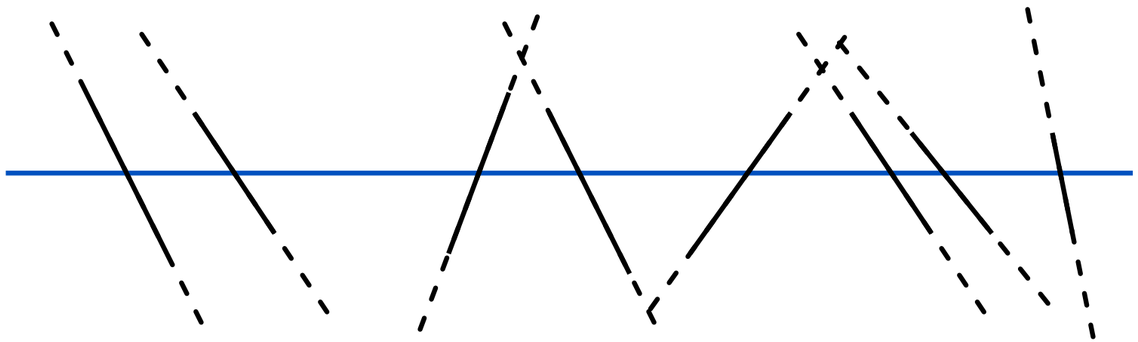
The event list is an abstract data structure that stores all events in the order in which they occur

The status structure is an abstract data structure that maintains the current status

Event list and status structure

The event list is an abstract data structure that stores all events in the order in which they occur

The status structure is an abstract data structure that maintains the current status

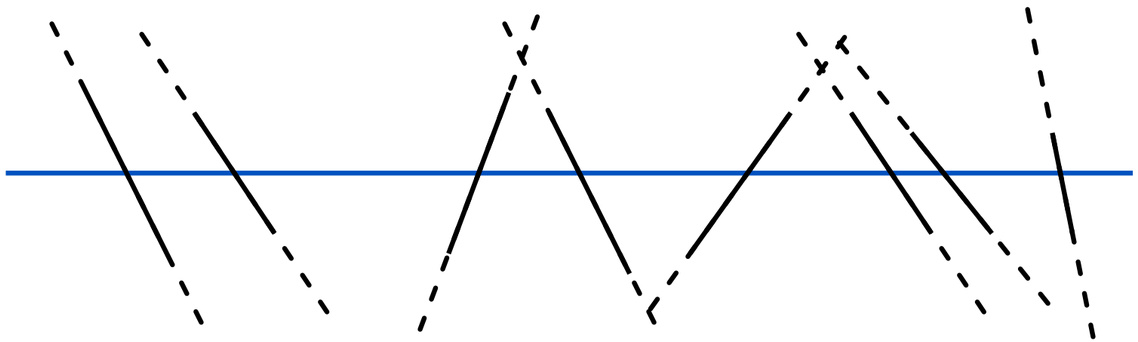


Here: The status is the subset of currently intersected line segments in the order of intersection by the sweep line

Event list and status structure

The event list is an abstract data structure that stores all events in the order in which they occur

The status structure is an abstract data structure that maintains the current status

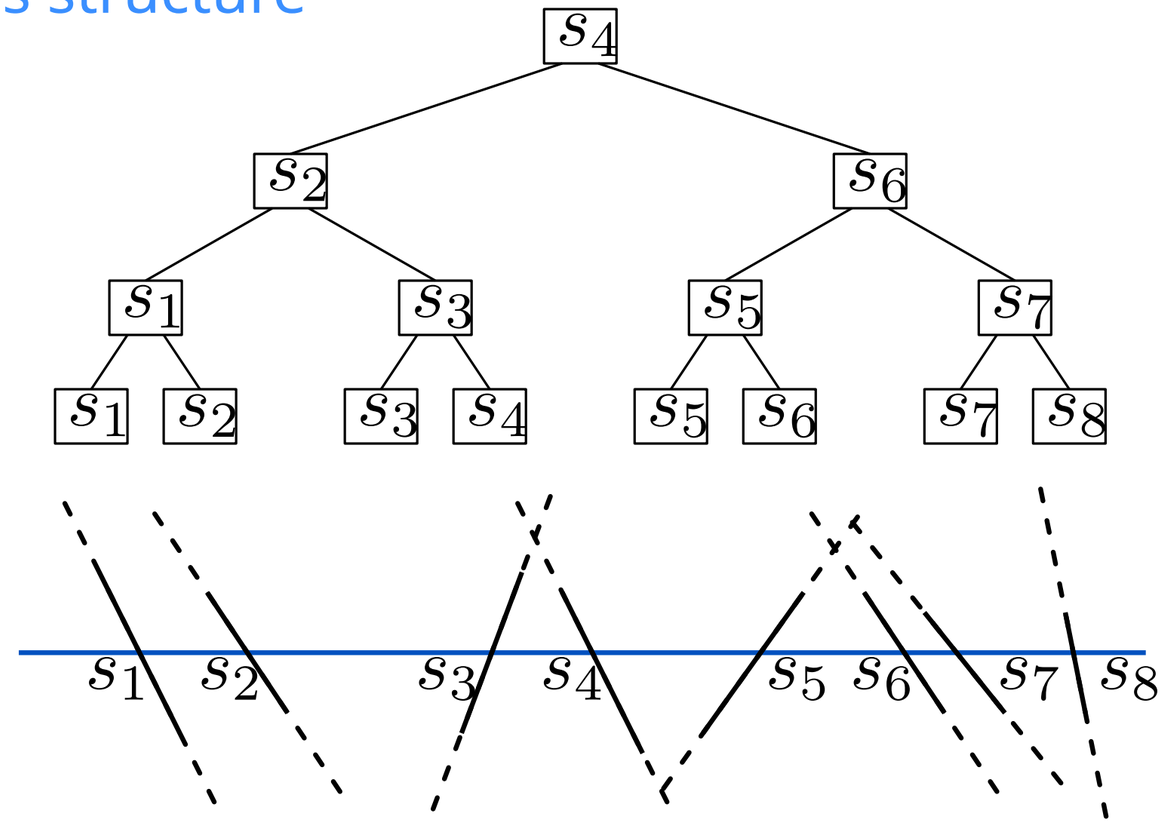


Here: The status is the subset of currently intersected line segments in the order of intersection by the sweep line

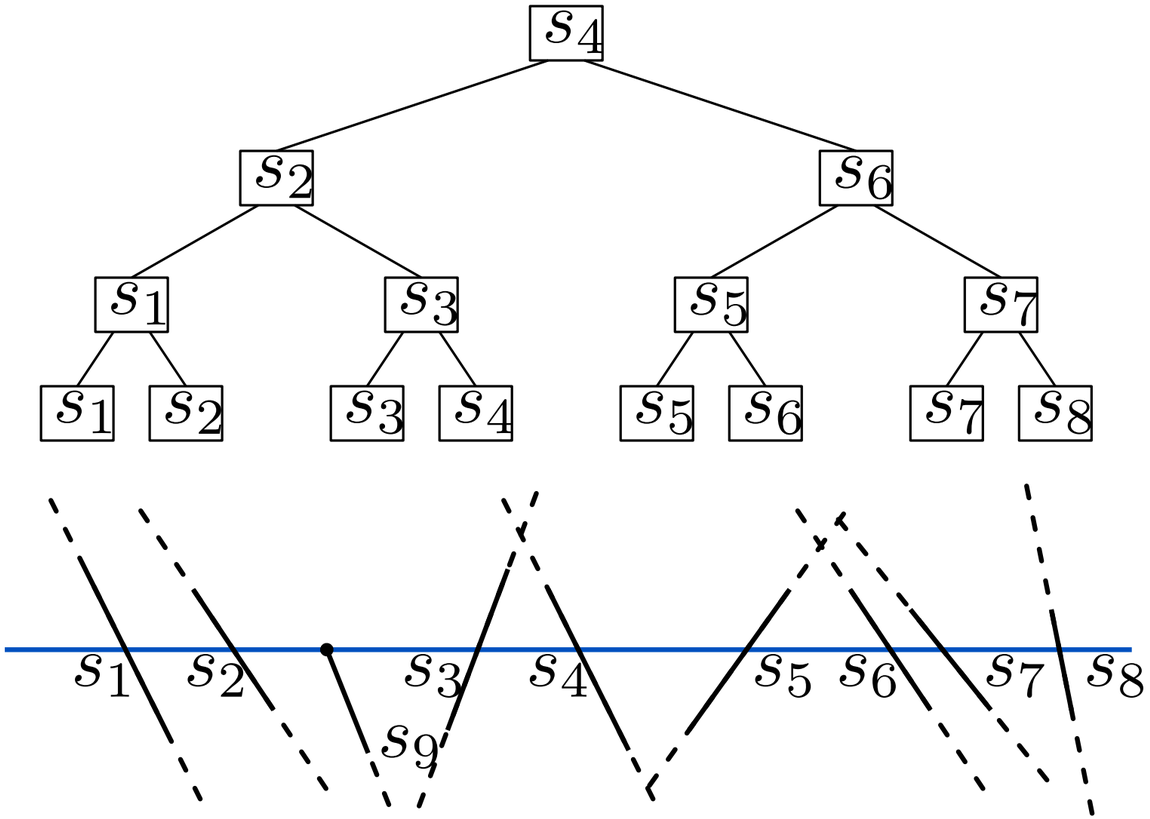
Question: Data structure for status?

Status structure

We use a balanced binary search tree with the line segments in the leaves as the status structure

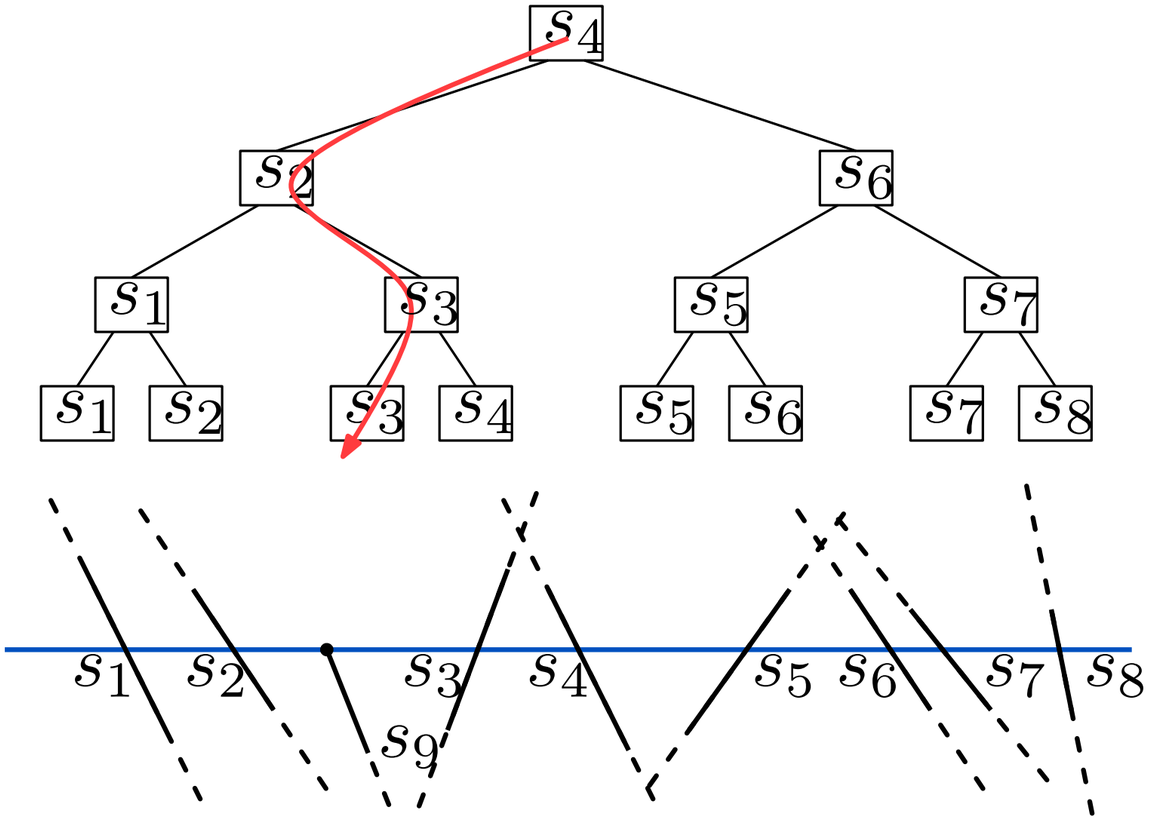


Status structure



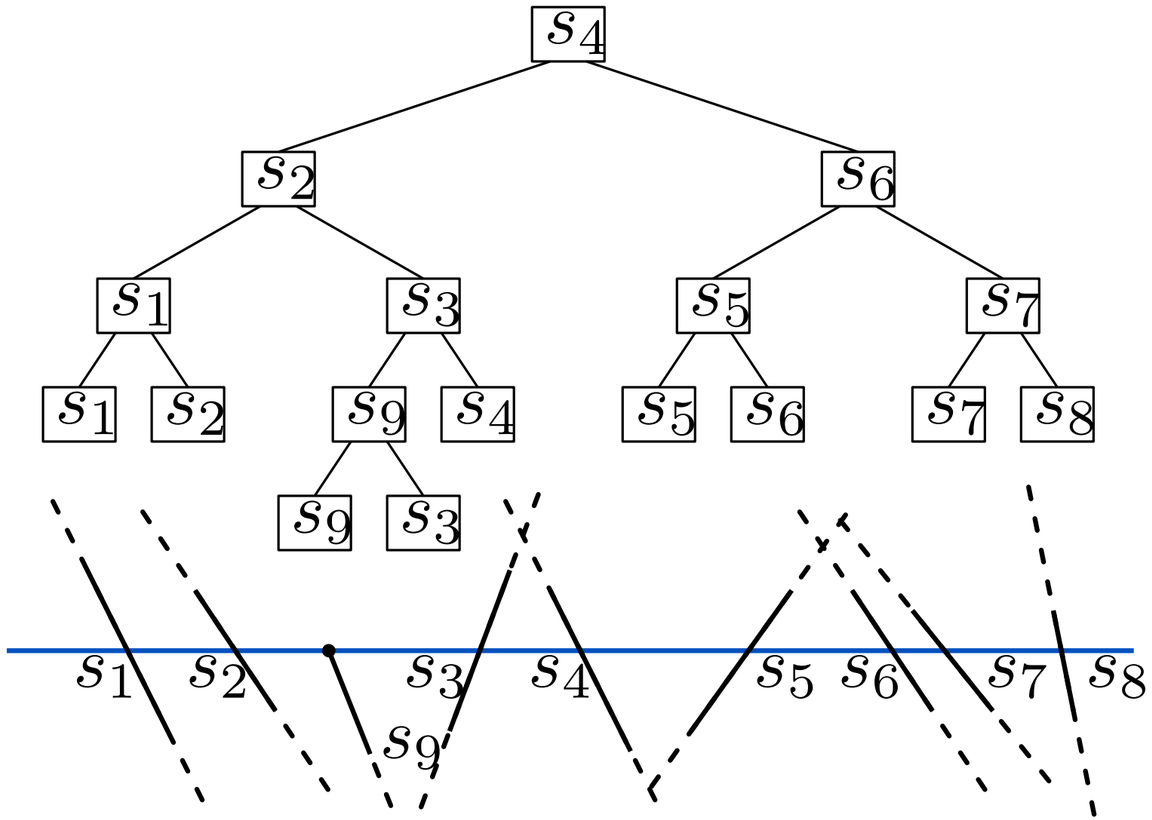
Upper endpoint: search, and insert

Status structure



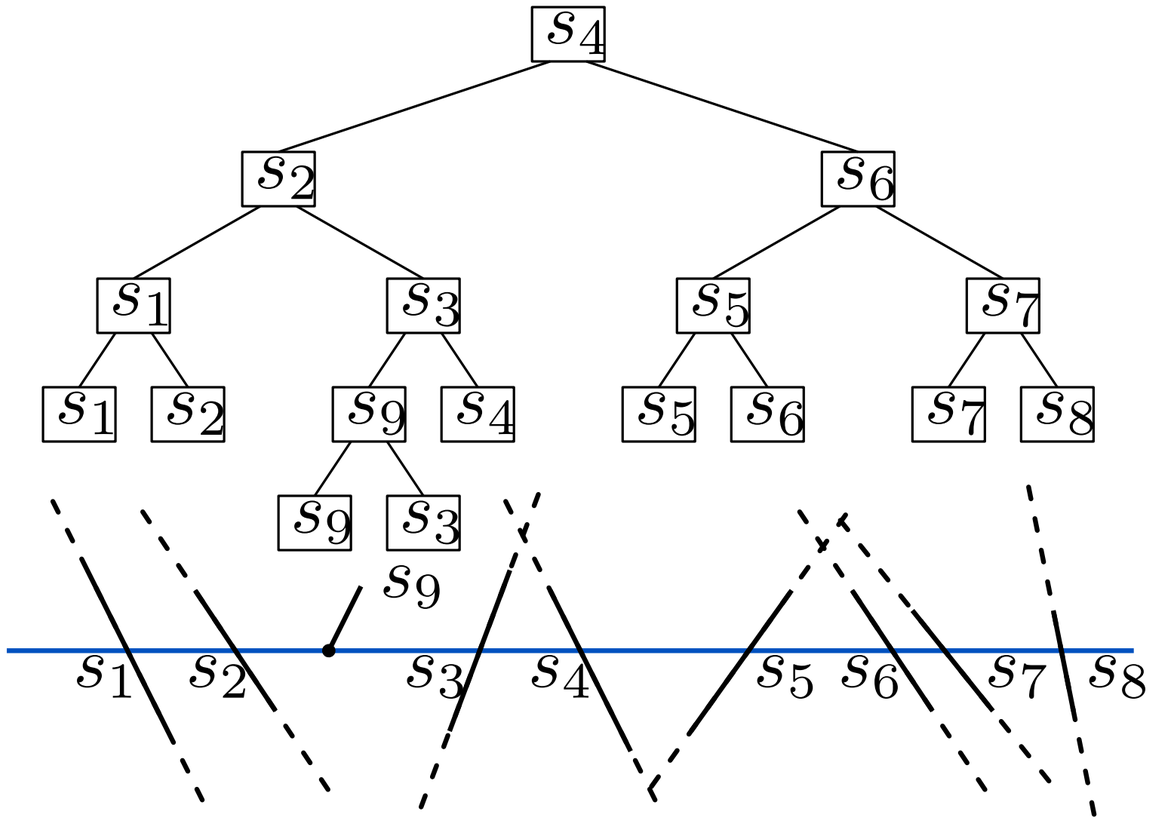
Upper endpoint: search, and insert

Status structure



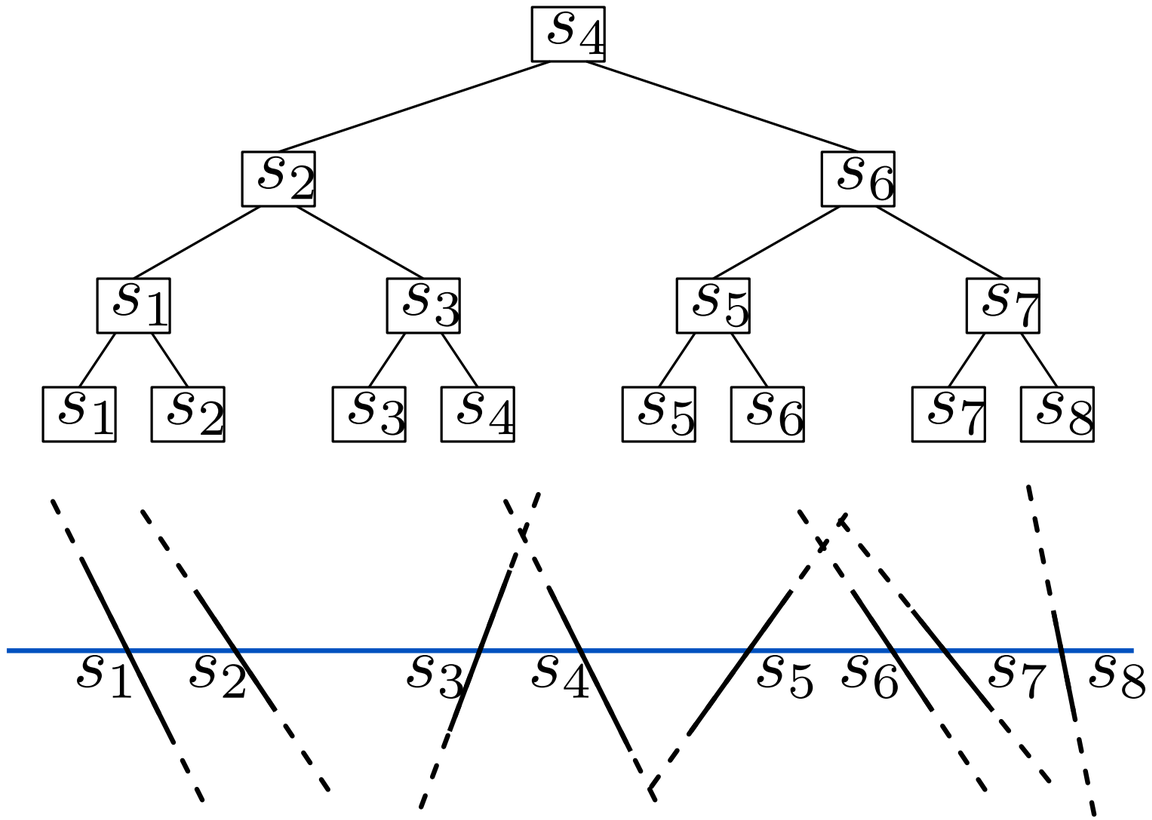
Upper endpoint: search, and insert

Status structure



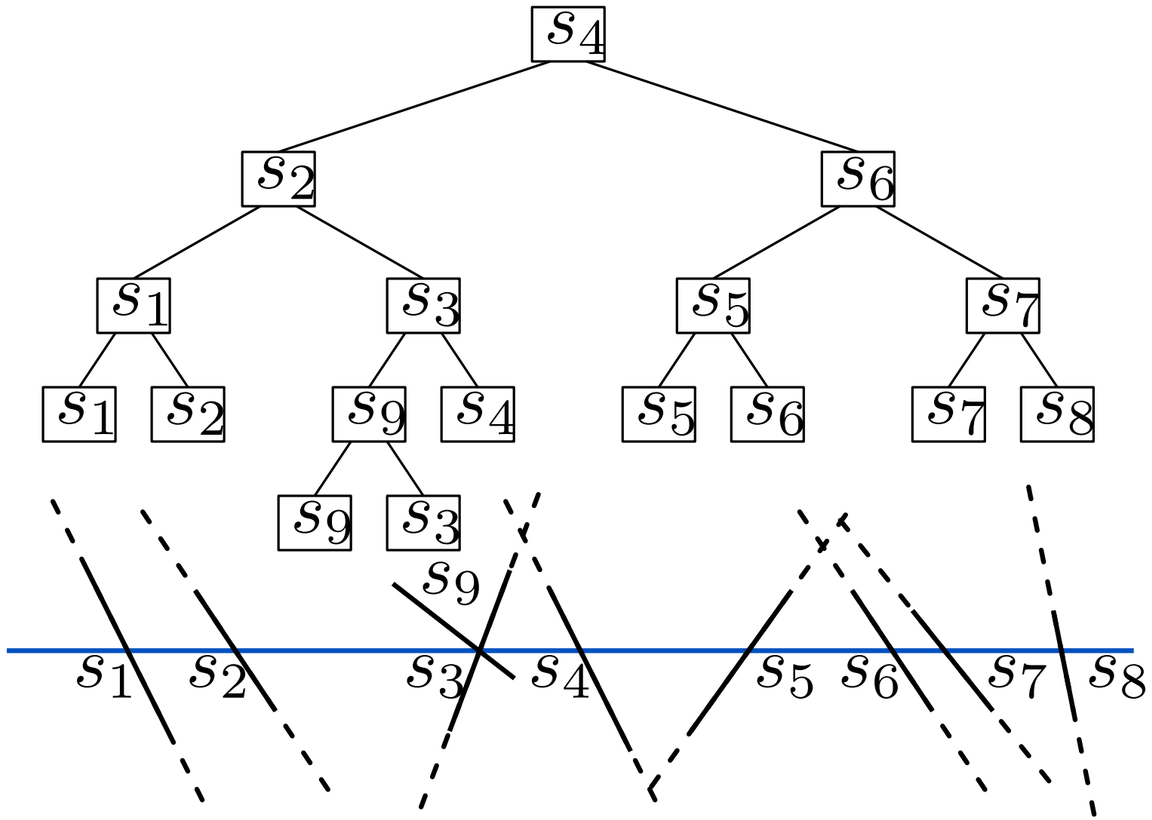
Lower endpoint: delete

Status structure



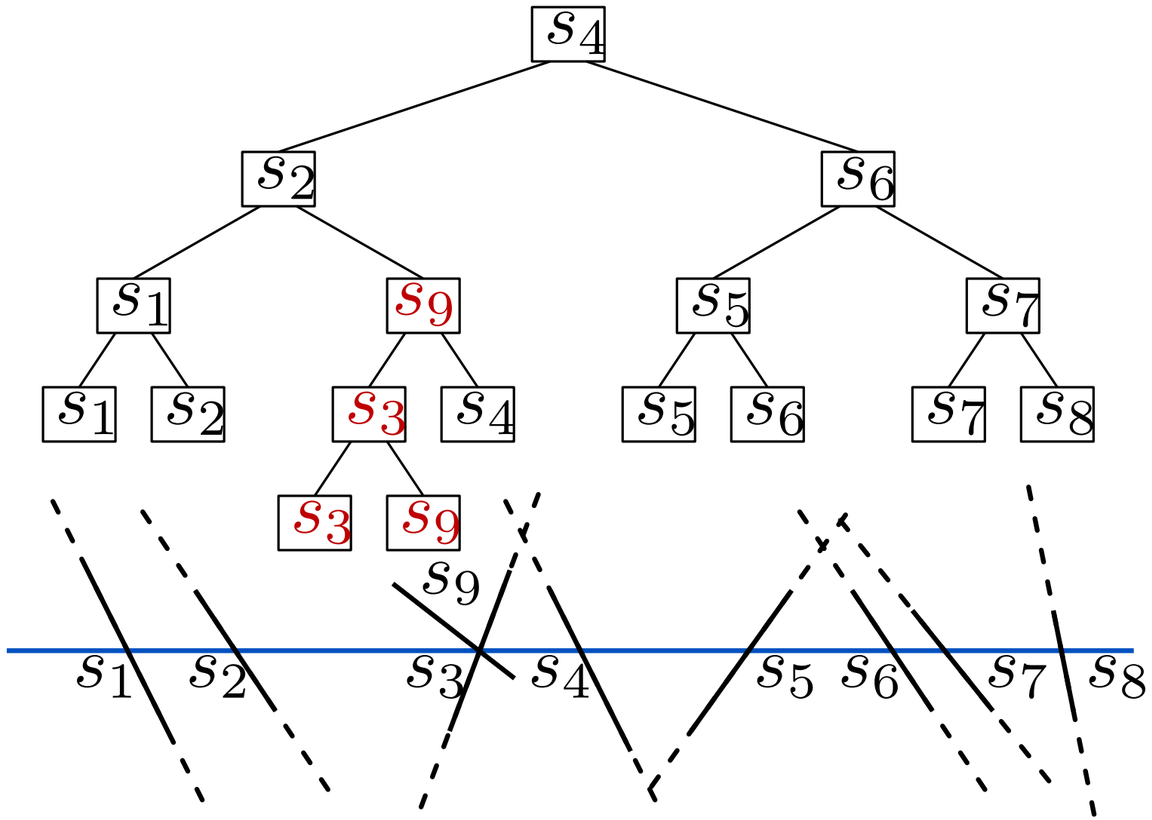
Lower endpoint: delete

Status structure



Intersection point: swap leaves and update information in search path

Status structure



Intersection point: swap leaves and update information in search path

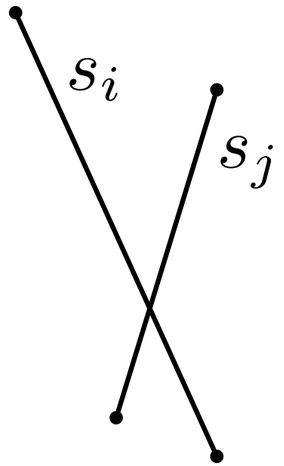
Finding events

Before the sweep algorithm starts, we know all upper endpoint events and all lower endpoint events

Finding events

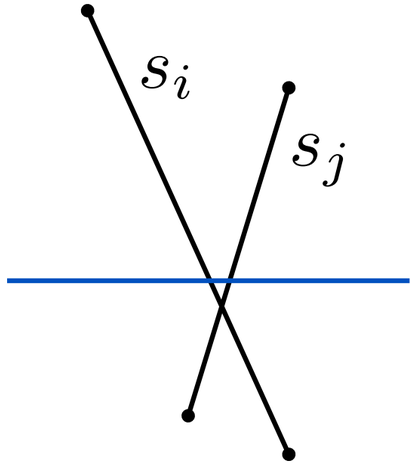
Before the sweep algorithm starts, we know all upper endpoint events and all lower endpoint events

But: How do we know intersection point events? (those we were trying to find…)



Finding events

Before the sweep algorithm starts, we know all upper endpoint events and all lower endpoint events

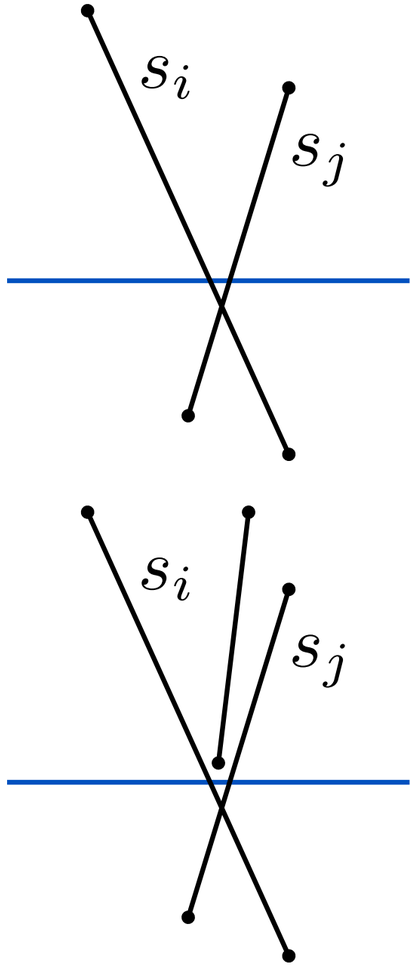


But: How do we know intersection point events? (those we were trying to find…)

Recall: Two line segments can only intersect if they are horizontal neighbors

Finding events

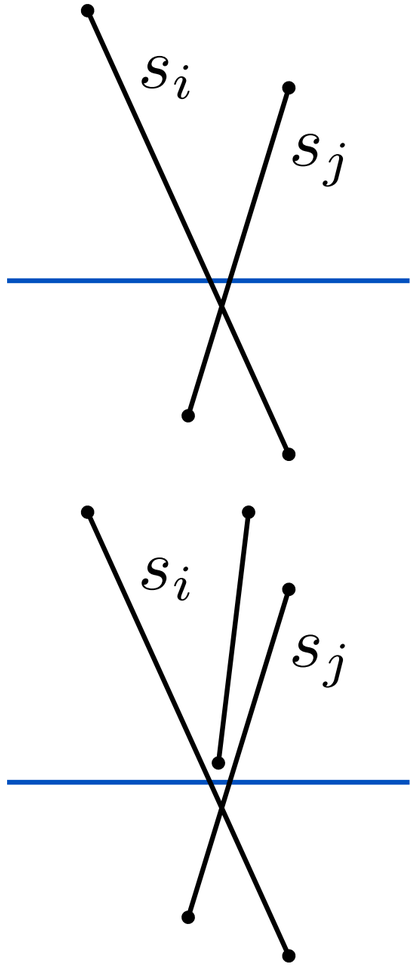
Lemma: Two line segments s_i and s_j can only intersect after (= below) they have become horizontal neighbors



Finding events

Lemma: Two line segments s_i and s_j can only intersect after (= below) they have become horizontal neighbors

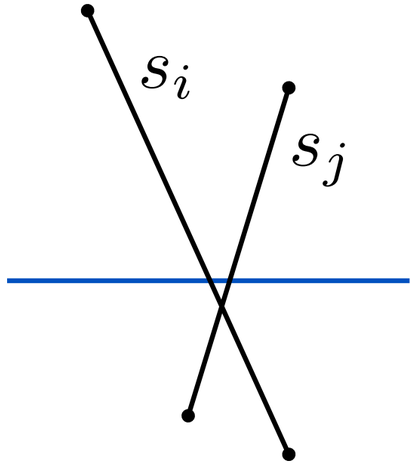
Proof idea: Just imagine that the sweep line is ever so slightly above the intersection point of s_i and s_j, but below any other event □



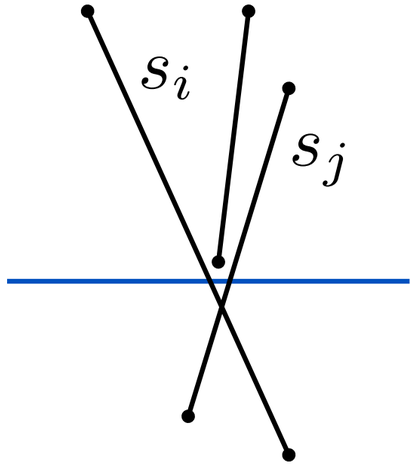
Finding events

Lemma: Two line segments s_i and s_j can only intersect after (= below) they have become horizontal neighbors

Proof idea: Just imagine that the sweep line is ever so slightly above the intersection point of s_i and s_j, but below any other event □



Also: some earlier (= higher) event made s_i and s_j horizontally adjacent



Quiz

Which data structure should we use as event queue if we want to avoid inserting the same event several times?

A: array

B: min-heap

C: balanced binary search tree

Quiz

Which data structure should we use as event queue if we want to avoid inserting the same event several times?

~~A: array~~

we need to insert intersection point events, when we find them $\rightarrow$ dynamic

B: min-heap

C: balanced binary search tree

Quiz

Which data structure should we use as event queue if we want to avoid inserting the same event several times?

A: ~~array~~ ← we need to insert intersection point events, when we find them → dynamic

B: ~~min-heap~~ ← we want to check if event already inserted → search

C: balanced binary search tree

Quiz

Which data structure should we use as event queue if we want to avoid inserting the same event several times?

A: array

we need to insert intersection point events, when we find them → dynamic

B: min-heap

we want to check if event already inserted → search

C: balanced binary search tree

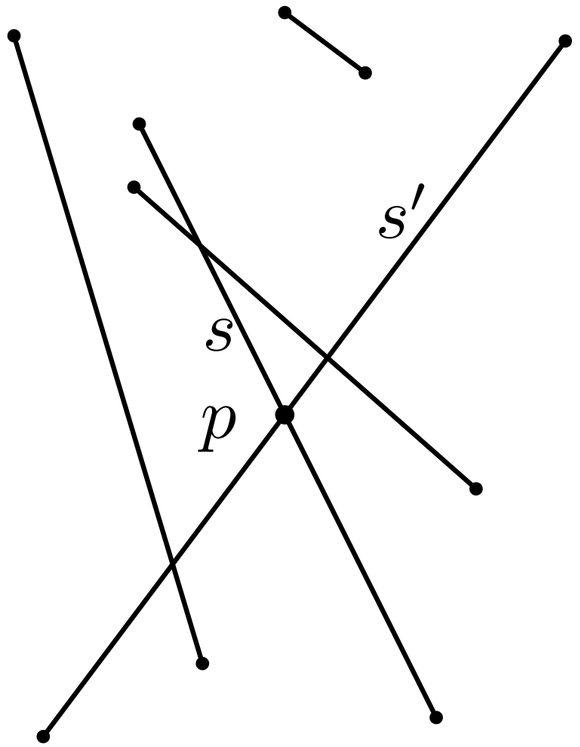
Quiz

Which data structure should we use as event queue if we want to avoid inserting the same event several times?

A: array

B: min-heap

C: balanced binary search tree



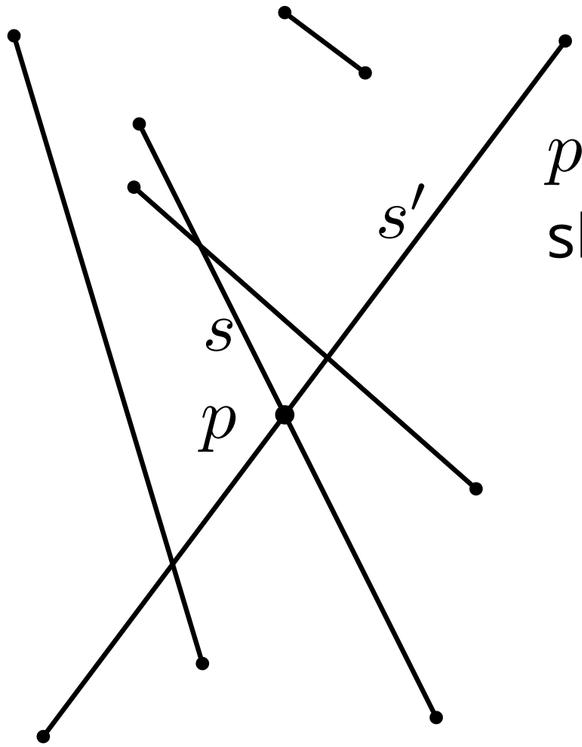
Quiz

Which data structure should we use as event queue if we want to avoid inserting the same event several times?

A: array

B: min-heap

C: balanced binary search tree



p is detected twice, but should be inserted once

Summary

The event list is an abstract data structure that stores all events in the order in which they occur

Here: binary search tree (ordered by decreasing y coordinate)

Summary

The event list is an abstract data structure that stores all events in the order in which they occur

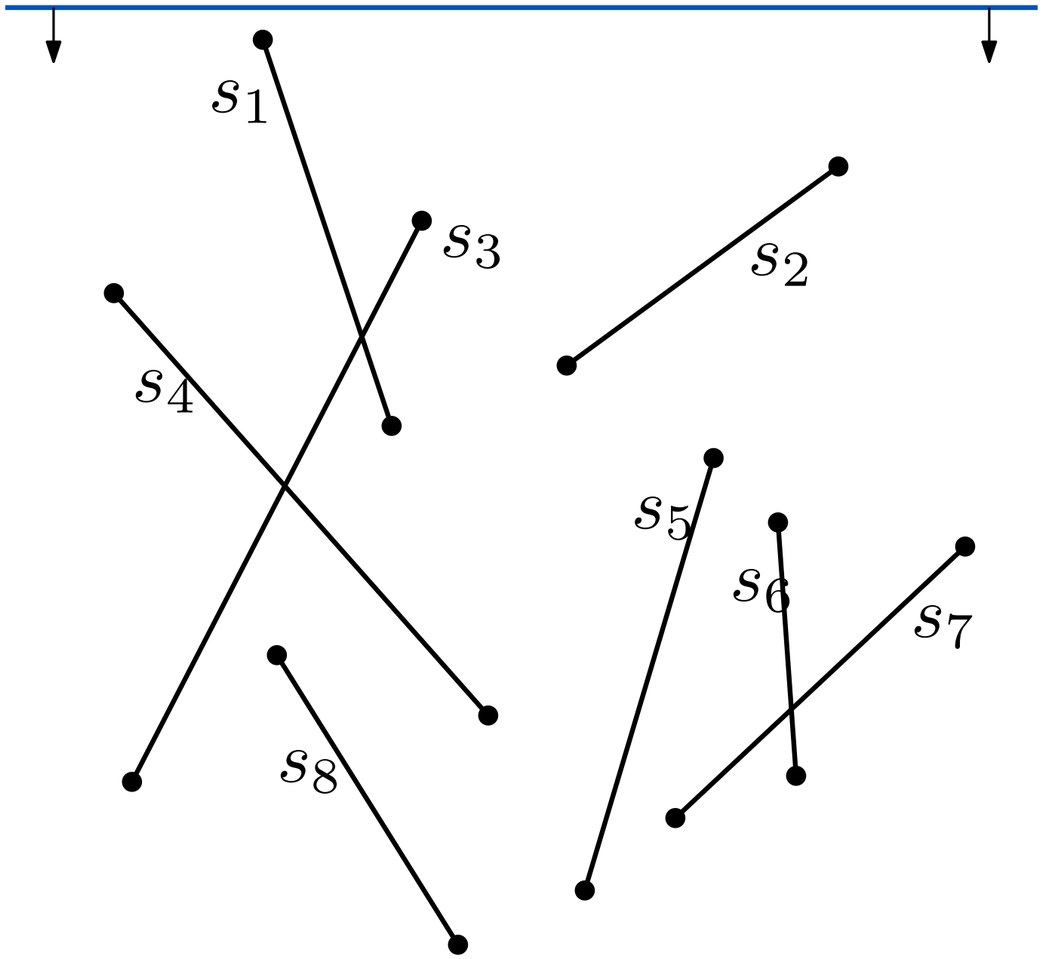
Here: binary search tree (ordered by decreasing y coordinate)

The status structure is an abstract data structure that maintains the current status

Here: binary search tree (ordered by x at current status)

Plane Sweep Algorithm

Structure of sweep algorithm



Structure of sweep algorithm

Algorithm FINDINTERSECTIONS(S)

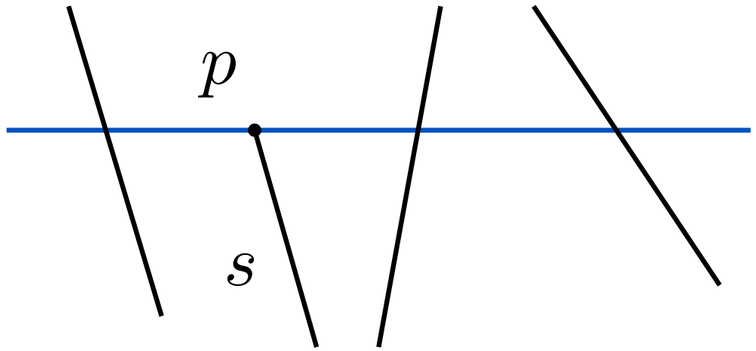
Input: set S of line segments in the plane
Output: intersection points of the segments in S, with a pair of segments containing it for each

1: initialize an empty event queue Q
2: insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it
3: initialize an empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

Event handling

If the event is an upper endpoint event, and s is the line segment that starts at p:

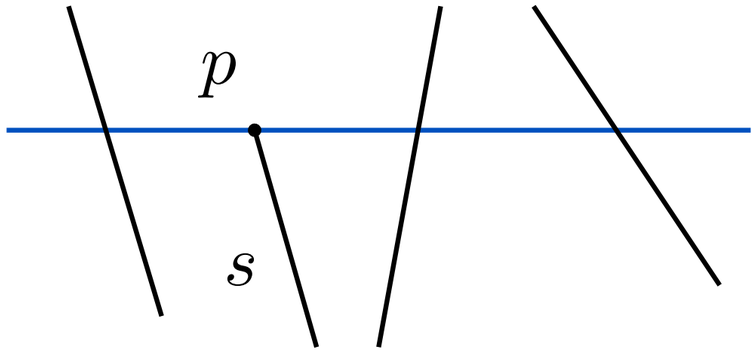
1. search with p in T, and
   insert s



Event handling

If the event is an upper endpoint event, and s is the line segment that starts at p:

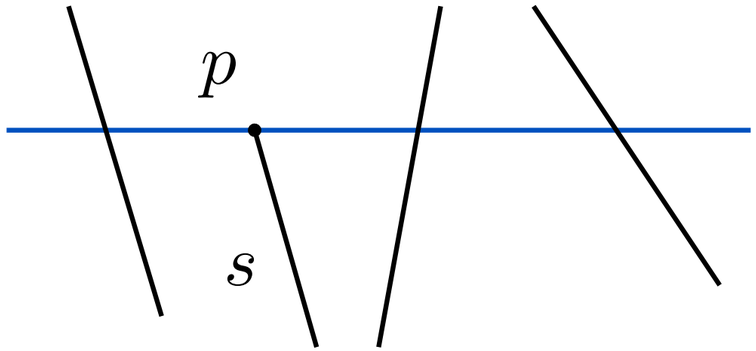
1. search with p in T, and insert s
2. if s intersects its left neighbor in T, then determine the intersection point and insert it in Q



Event handling

If the event is an upper endpoint event, and s is the line segment that starts at p:

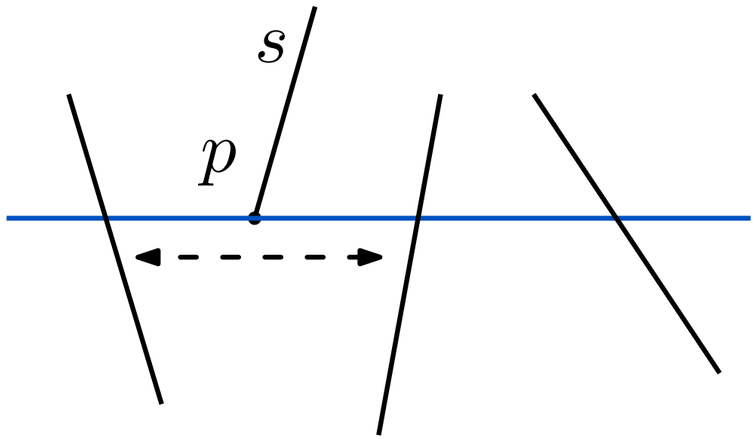
1. search with p in T, and insert s
2. if s intersects its left neighbor in T, then determine the intersection point and insert it in Q
3. if s intersects its right neighbor in T, then determine the intersection point and insert it in Q



Event handling

If the event is an lower endpoint event, and s is the line segment that ends at p:

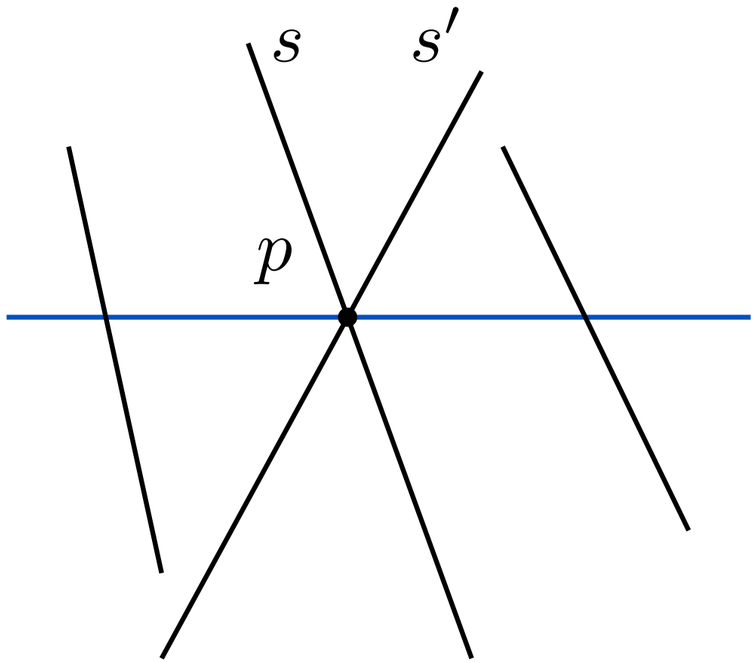
1. search with p in T, and delete s
2. let s_ℓ and s_r be the left and right neighbors of s in T (before deletion). If they intersect below the sweep line, then insert their intersection point as an event in Q



Event handling

If the event is an intersection point event, and s and s' intersect at p:

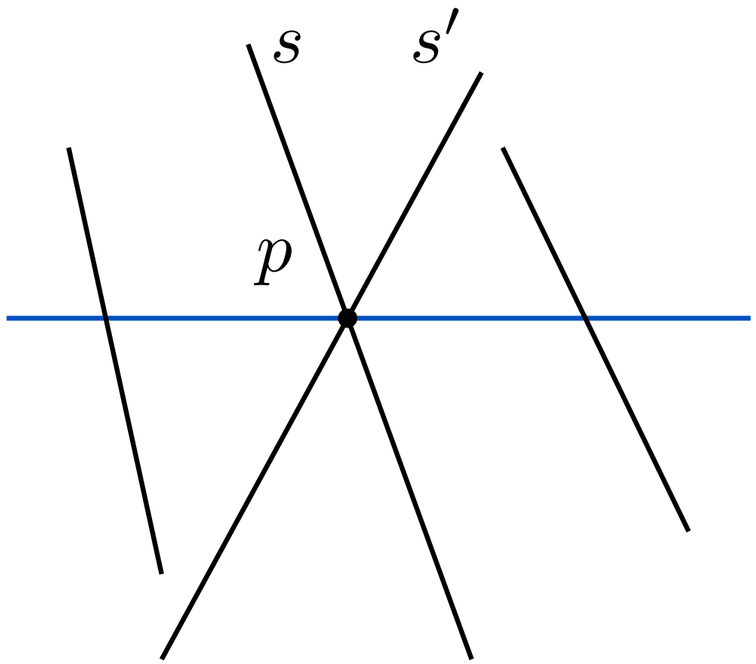
Question: steps?



Event handling

If the event is an intersection point event, and s and s' intersect at p:

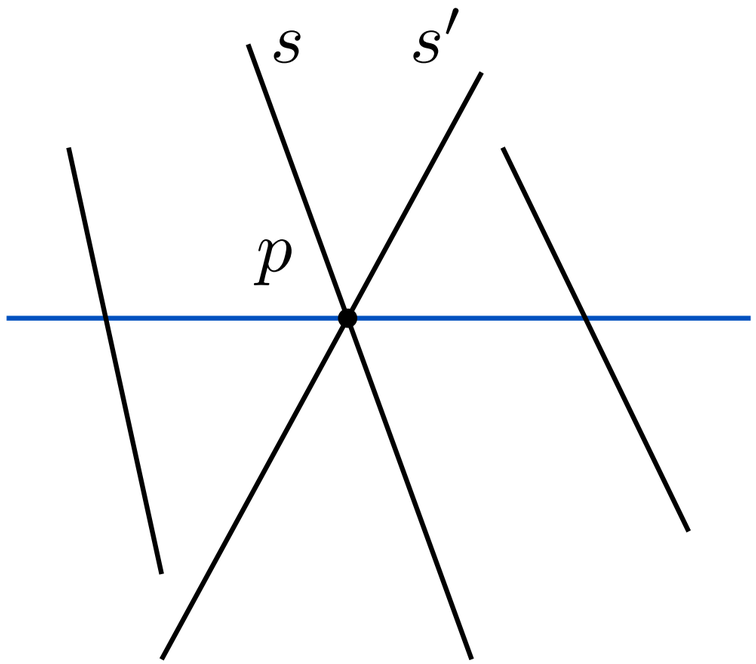
1. exchange s and s' in T



Event handling

If the event is an intersection point event, and s and s' intersect at p:

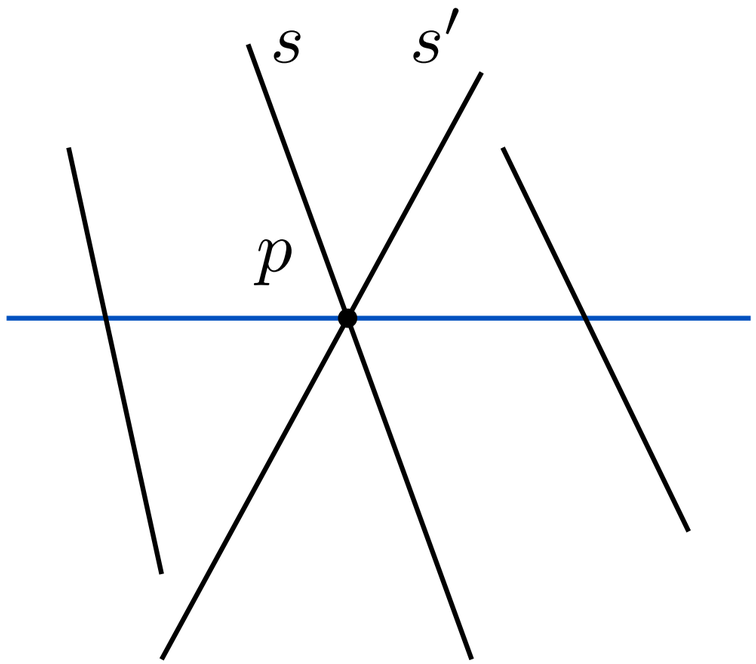
1. exchange s and s' in T
2. if s' and its new left neighbor in T intersect below the sweep line, insert this intersection point in Q



Event handling

If the event is an intersection point event, and s and s' intersect at p:

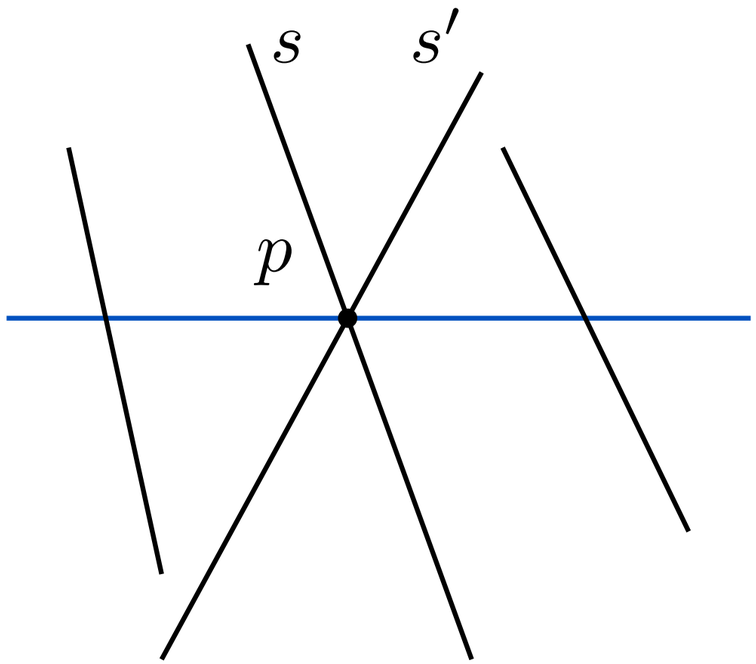
1. exchange s and s' in T
2. if s' and its new left neighbor in T intersect below the sweep line, insert this intersection point in Q
3. if s and its new right neighbor in T intersect below the sweep line, insert this intersection point in Q



Event handling

If the event is an intersection point event, and s and s' intersect at p:

1. exchange s and s' in T
2. if s' and its new left neighbor in T intersect below the sweep line, insert this intersection point in Q
3. if s and its new right neighbor in T intersect below the sweep line, insert this intersection point in Q
4. report the intersection point



Quiz

What is the overall running time of the algorithm for n line segments with k intersections?

A: $O(n \log n + k)$

B: $O((n + k) \log n)$

C: $O(nk)$

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

Quiz

What is the overall running time of the algorithm for n line segments with k intersections?

A: $O(n \log n + k)$

B: $O((n + k) \log n)$

C: $O(nk)$

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

Efficiency

How much time to handle an event?

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

Efficiency

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

How much time to handle an event?

At most one search in T and/or one insertion, deletion, or swap

At most twice finding a neighbor in T

At most one deletion from and two insertions in Q

Efficiency

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

How much time to handle an event?

At most one search in T and/or one insertion, deletion, or swap

At most twice finding a neighbor in T

At most one deletion from and two insertions in Q

Since T and Q are balanced binary search trees, handling an event takes only $O(\log n)$ time

Efficiency

How many events?

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

Efficiency

How many events?

- $2n$ for the upper and lower endpoints
- k for the intersection points, if there are k of them

Algorithm FindIntersections(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HandleEventPoint(p)

Efficiency

How many events?

- $2n$ for the upper and lower endpoints
- k for the intersection points, if there are k of them

In total: $O(n + k)$ events

Algorithm FINDINTERSECTIONS(S)

1: initialize empty event queue Q
2: insert segment endpoints into Q;
3: initialize empty status structure T
4: **while** Q is not empty **do**
5: determine next event point p in Q and delete it
6: HANDLEEVENTPOINT(p)

Efficiency

Initialization takes $O(n \log n)$ time (to put all endpoint events into Q)

Each of the $O(n + k)$ events takes $O(\log n)$ time

Efficiency

Initialization takes $O(n \log n)$ time (to put all endpoint events into Q)

Each of the $O(n + k)$ events takes $O(\log n)$ time

The algorithm takes $O(n \log n + k \log n)$ time

Efficiency

Initialization takes $O(n \log n)$ time (to put all endpoint events into Q)
Each of the $O(n + k)$ events takes $O(\log n)$ time

The algorithm takes $O(n \log n + k \log n)$ time

If $k = O(n)$, then this is $O(n \log n)$

Efficiency

Initialization takes $O(n \log n)$ time (to put all endpoint events into Q)

Each of the $O(n + k)$ events takes $O(\log n)$ time

The algorithm takes $O(n \log n + k \log n)$ time

If $k = O(n)$, then this is $O(n \log n)$

Notes:

if k is really large, the brute force $O(n^2)$ time algorithm is more efficient

Faster algorithm exists: $O(n \log n + k)$

Efficiency

Question: How much storage does the algorithm take?

Efficiency

Question: Given that the event list is a binary tree that may store $O(k) = O(n^2)$ events, is the efficiency in jeopardy?

Efficiency

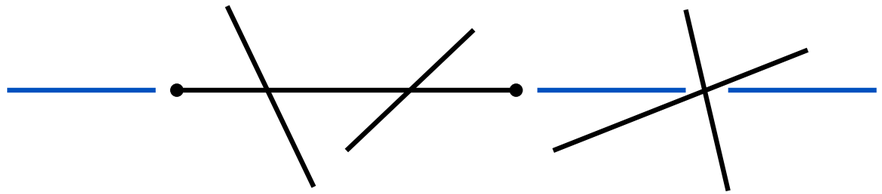
Solution:

Only store intersection points of currently adjacent segments.

Degenerate cases

How do we deal with degenerate cases?

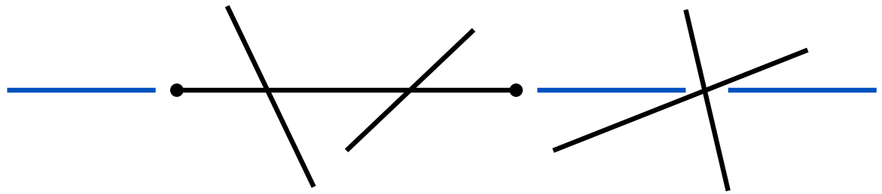
For two different events with the same y-coordinate, we treat them from left to right $\Rightarrow$ the “upper” endpoint of a horizontal line segment is its left endpoint



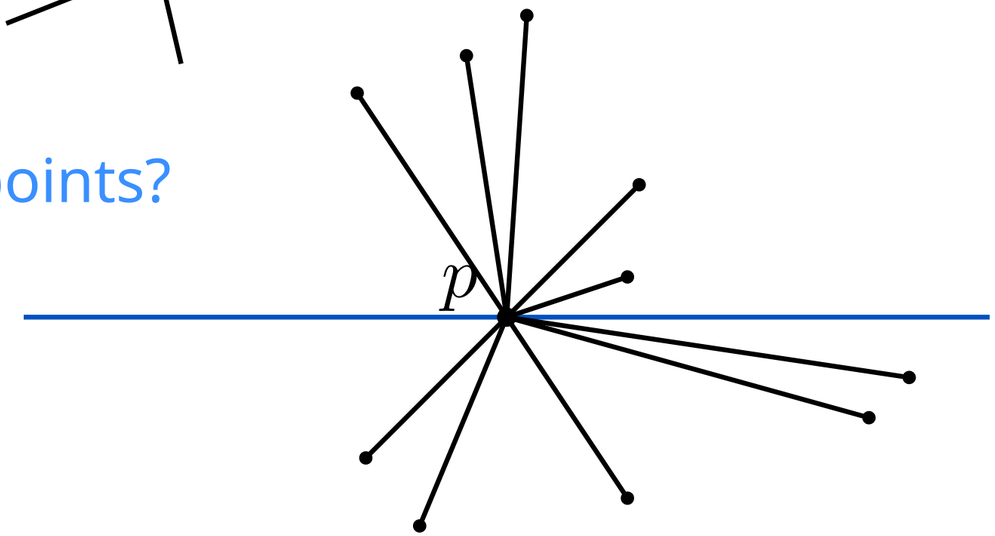
Degenerate cases

How do we deal with degenerate cases?

For two different events with the same y-coordinate, we treat them from left to right $\Rightarrow$ the “upper” endpoint of a horizontal line segment is its left endpoint



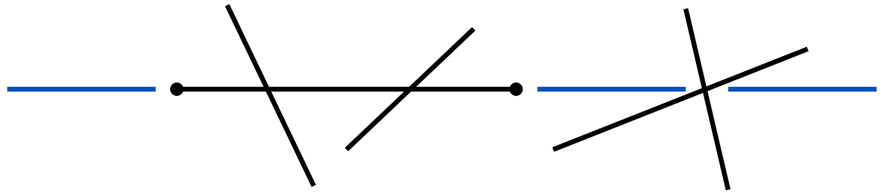
How about multiply coinciding event points?



Degenerate cases

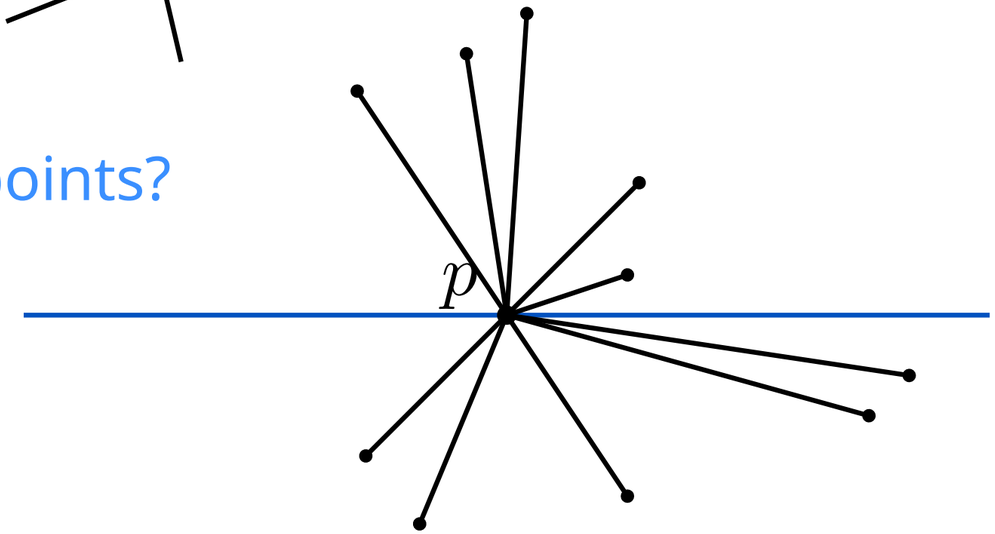
How do we deal with degenerate cases?

For two different events with the same y-coordinate, we treat them from left to right $\Rightarrow$ the “upper” endpoint of a horizontal line segment is its left endpoint



How about multiply coinciding event points?

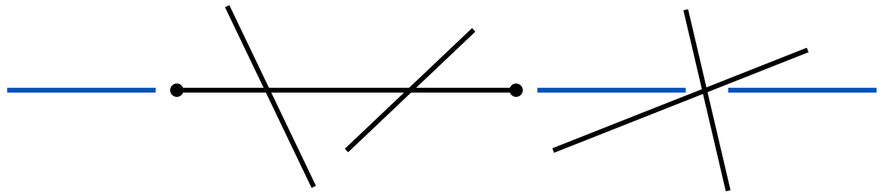
with care (no short answer)



Degenerate cases

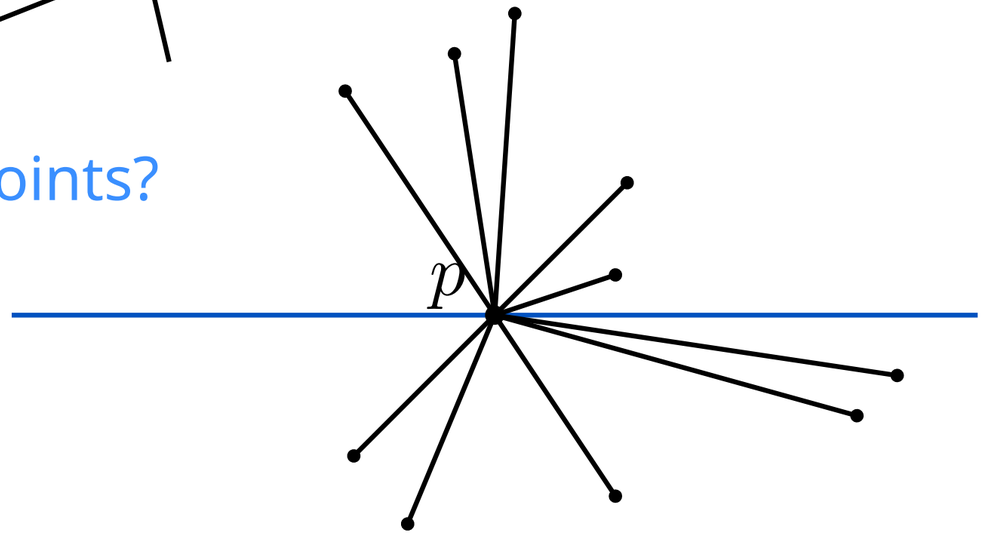
How do we deal with degenerate cases?

For two different events with the same y-coordinate, we treat them from left to right $\Rightarrow$ the “upper” endpoint of a horizontal line segment is its left endpoint

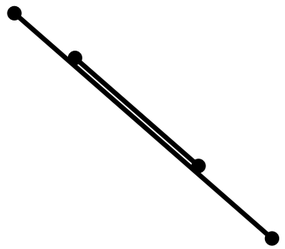


How about multiply coinciding event points?

with care (no short answer)



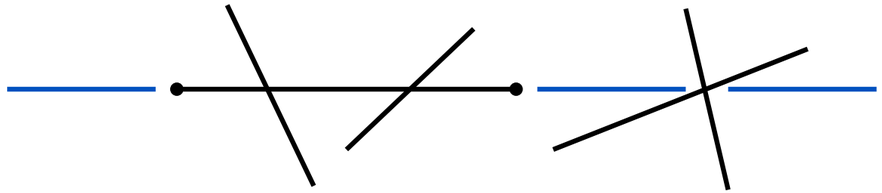
How do we handle overlap?



Degenerate cases

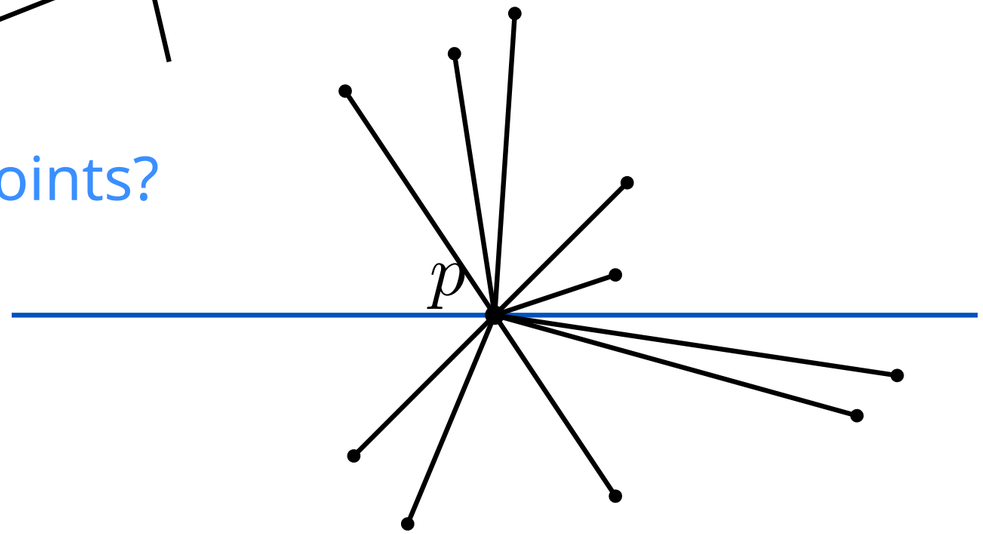
How do we deal with degenerate cases?

For two different events with the same y-coordinate, we treat them from left to right $\Rightarrow$ the “upper” endpoint of a horizontal line segment is its left endpoint



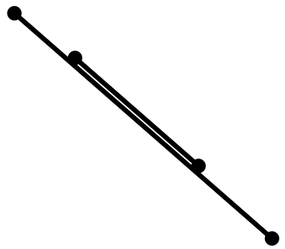
How about multiply coinciding event points?

with care (no short answer)



How do we handle overlap?

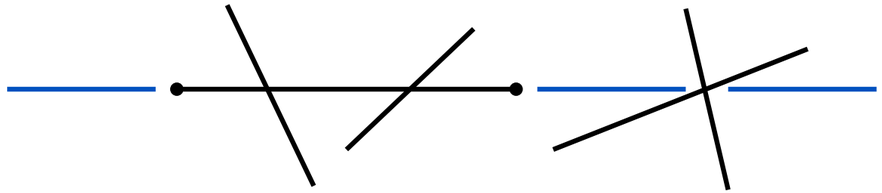
sort by ID



Degenerate cases

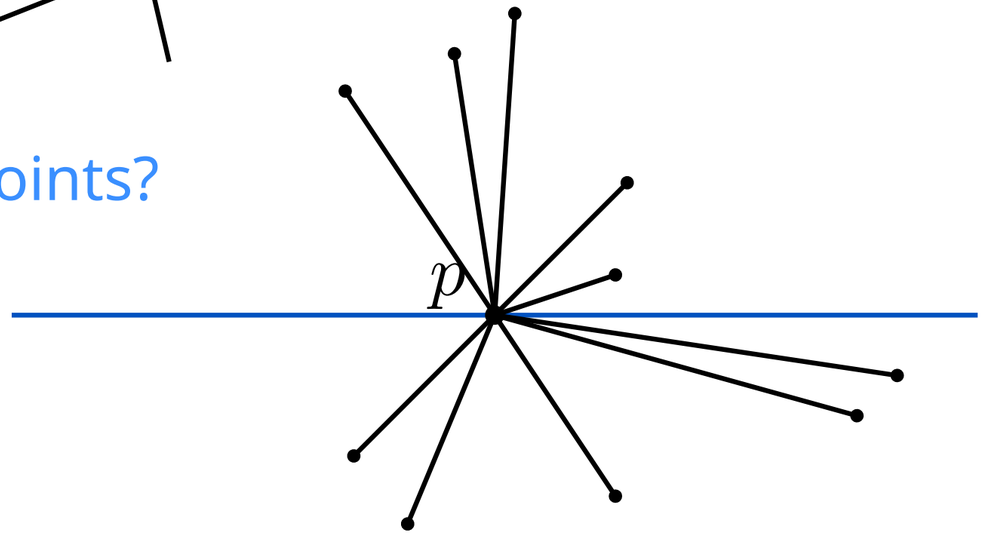
How do we deal with degenerate cases?

For two different events with the same y-coordinate, we treat them from left to right $\Rightarrow$ the “upper” endpoint of a horizontal line segment is its left endpoint



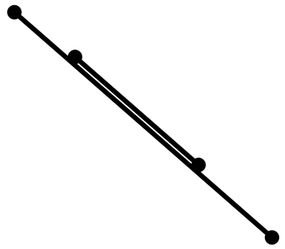
How about multiply coinciding event points?

with care (no short answer)



How do we handle overlap?

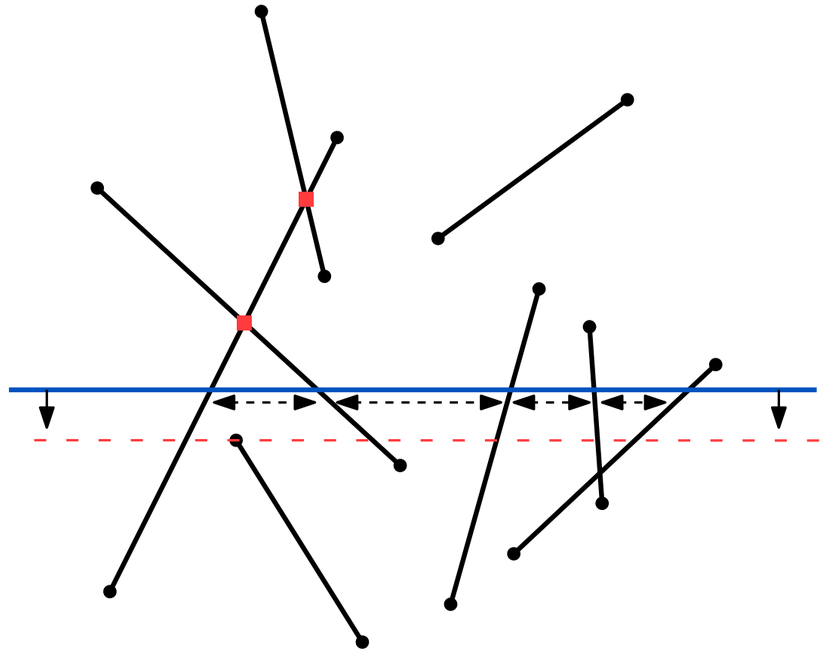
sort by ID



more difficult than it might seem

Plane Sweep – Conclusion

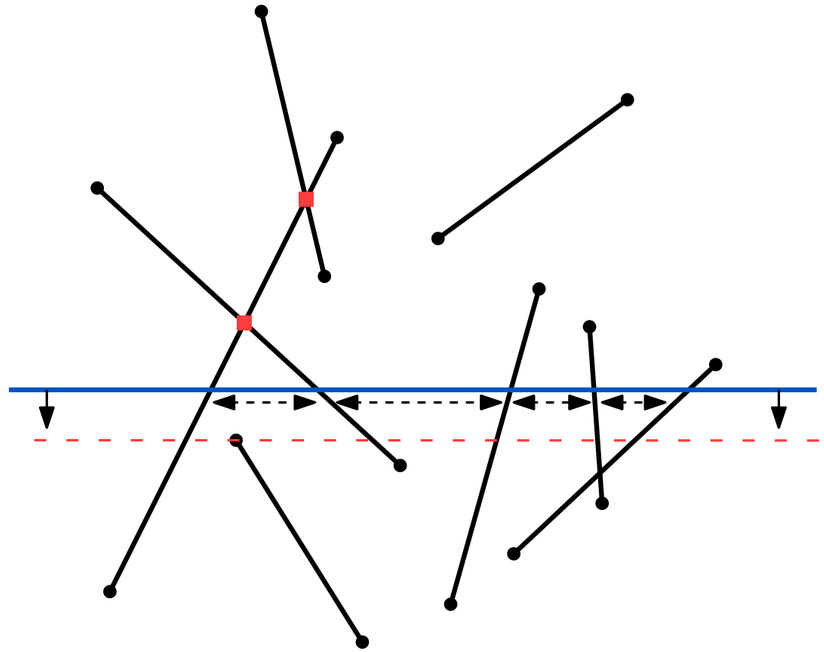
For every sweep algorithm:



Plane Sweep – Conclusion

For every sweep algorithm:

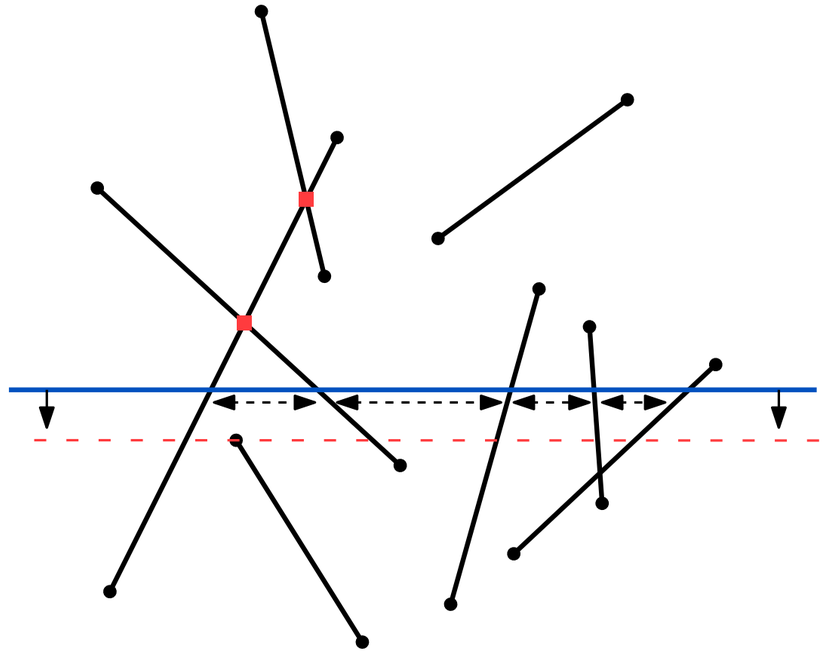
- define the status, determine events



Plane Sweep – Conclusion

For every sweep algorithm:

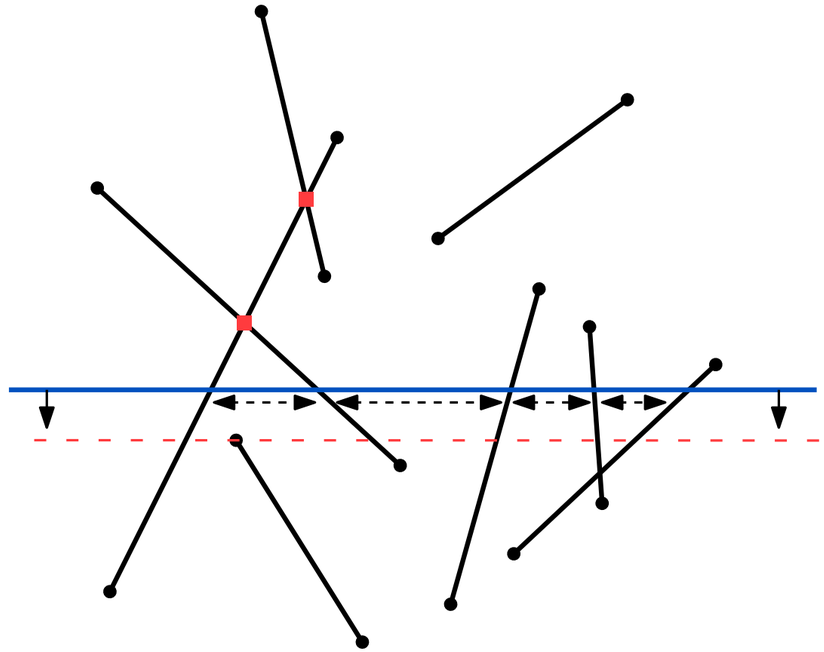
- define the status, determine events
- choose data structures for the status and event queue



Plane Sweep – Conclusion

For every sweep algorithm:

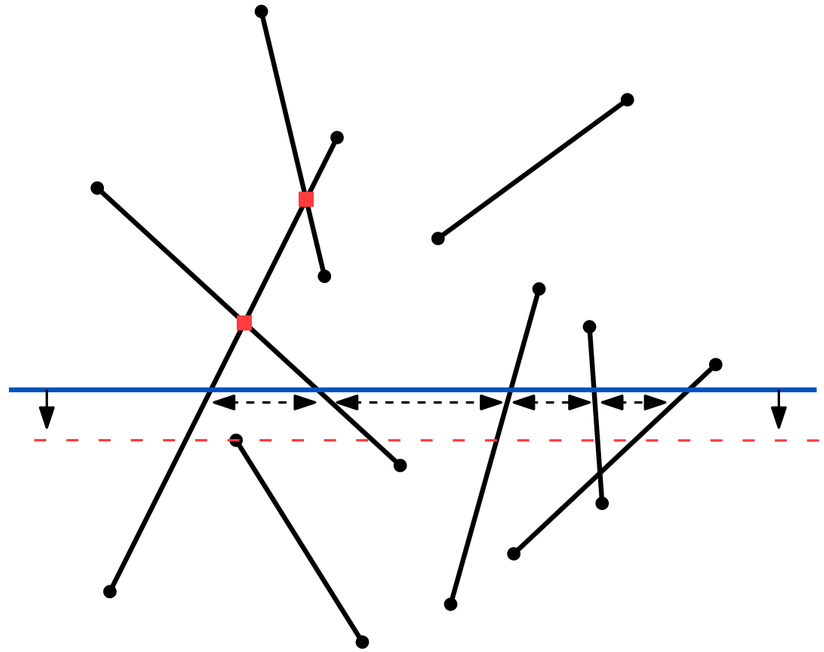
- define the status, determine events
- choose data structures for the status and event queue
- figure out how events must be handled (with sketches!)



Plane Sweep – Conclusion

For every sweep algorithm:

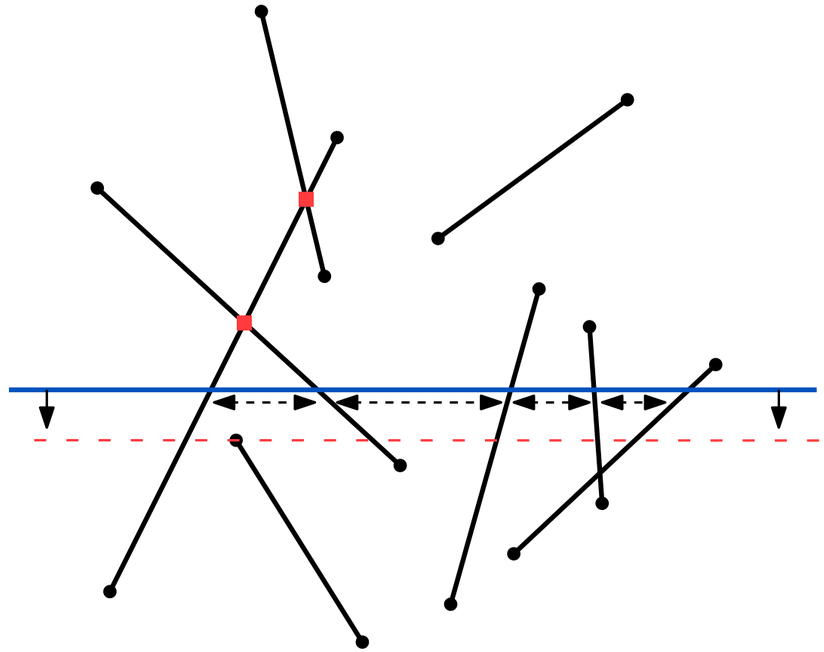
- define the status, determine events
- choose data structures for the status and event queue
- figure out how events must be handled (with sketches!)
- to analyze, determine the number of events and how much time they take



Plane Sweep – Conclusion

For every sweep algorithm:

- define the status, determine events
- choose data structures for the status and event queue
- figure out how events must be handled (with sketches!)
- to analyze, determine the number of events and how much time they take

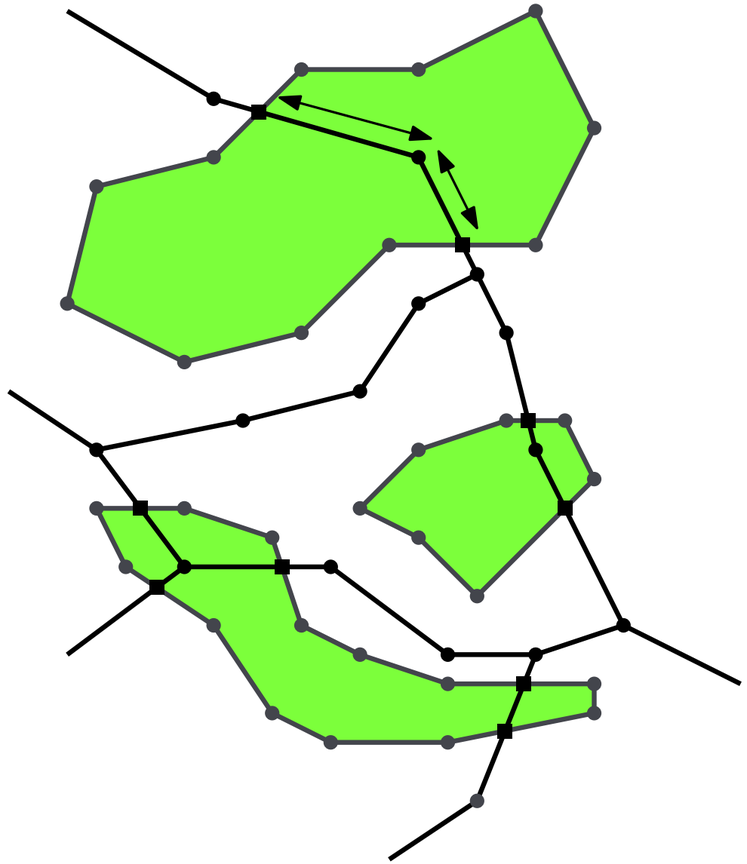


Then deal with degeneracies

Planar Subdivisions and Map Overlay

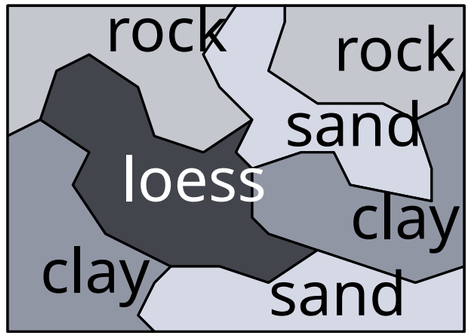
Map overlay

To solve map overlay questions, we need (at the least) intersection points from two sets of line segments.

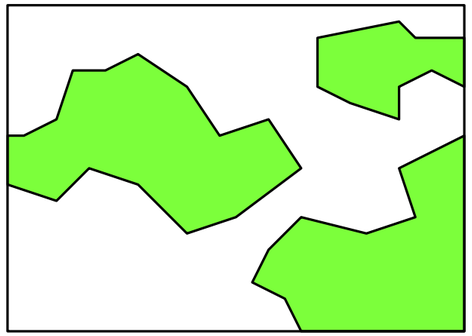


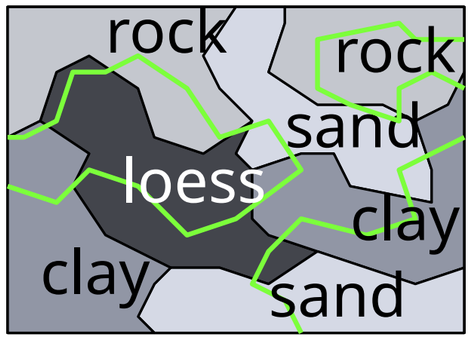
Map overlay

To solve map overlay questions, we need (at the least) intersection points from two sets of line segments.



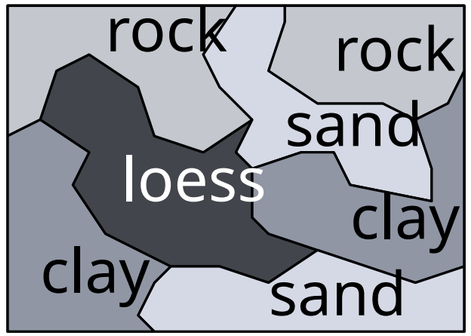
To solve map overlay questions, we also need to be able to represent subdivisions



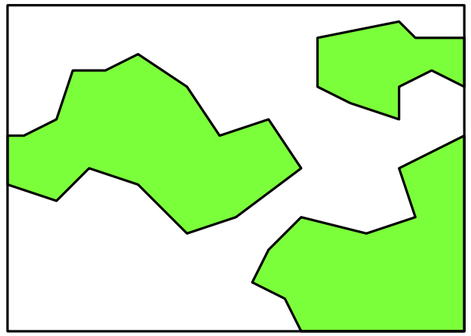


Map overlay

To solve map overlay questions, we need (at the least) intersection points from two sets of line segments.

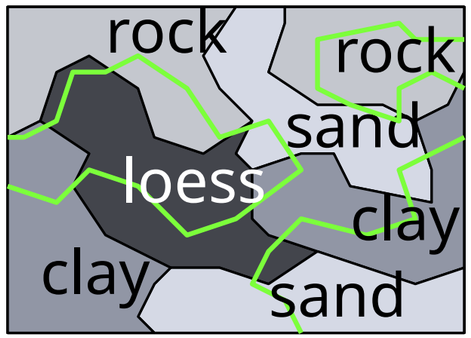


To solve map overlay questions, we also need to be able to represent subdivisions



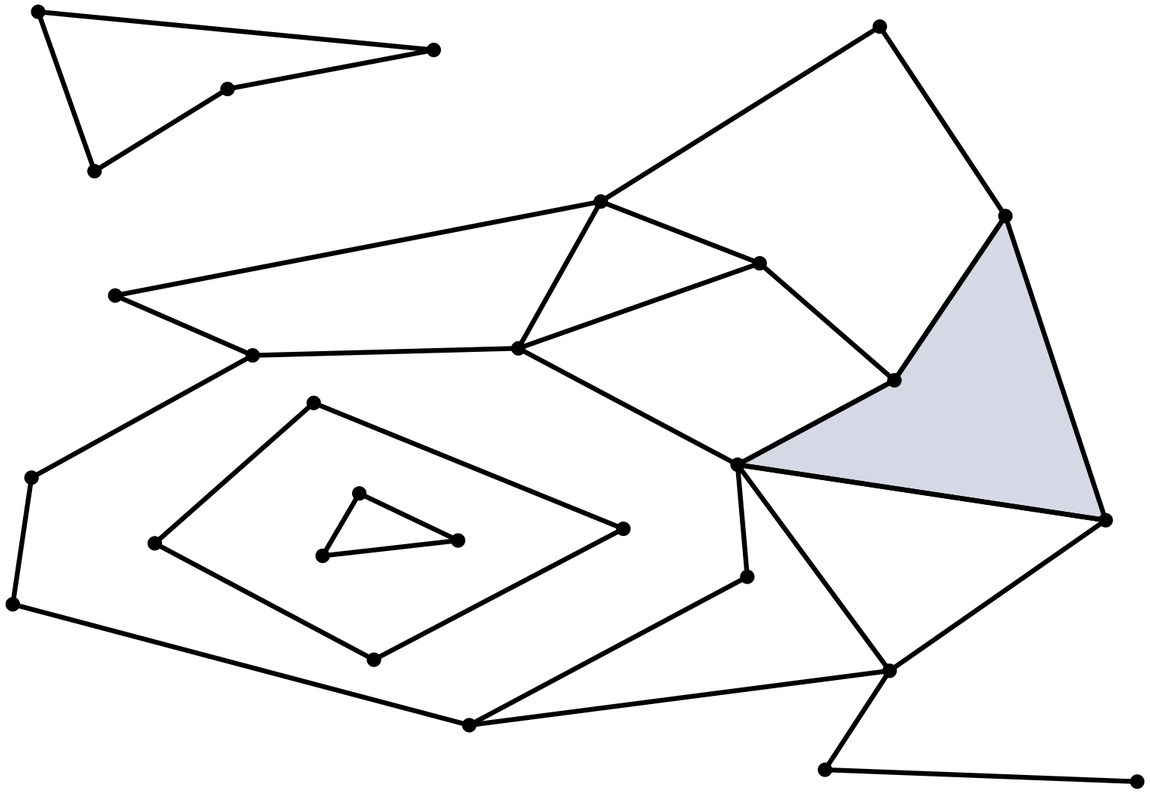
now

a data structure for planar subdivisions:
Doubly-connected edge list (DCEL)



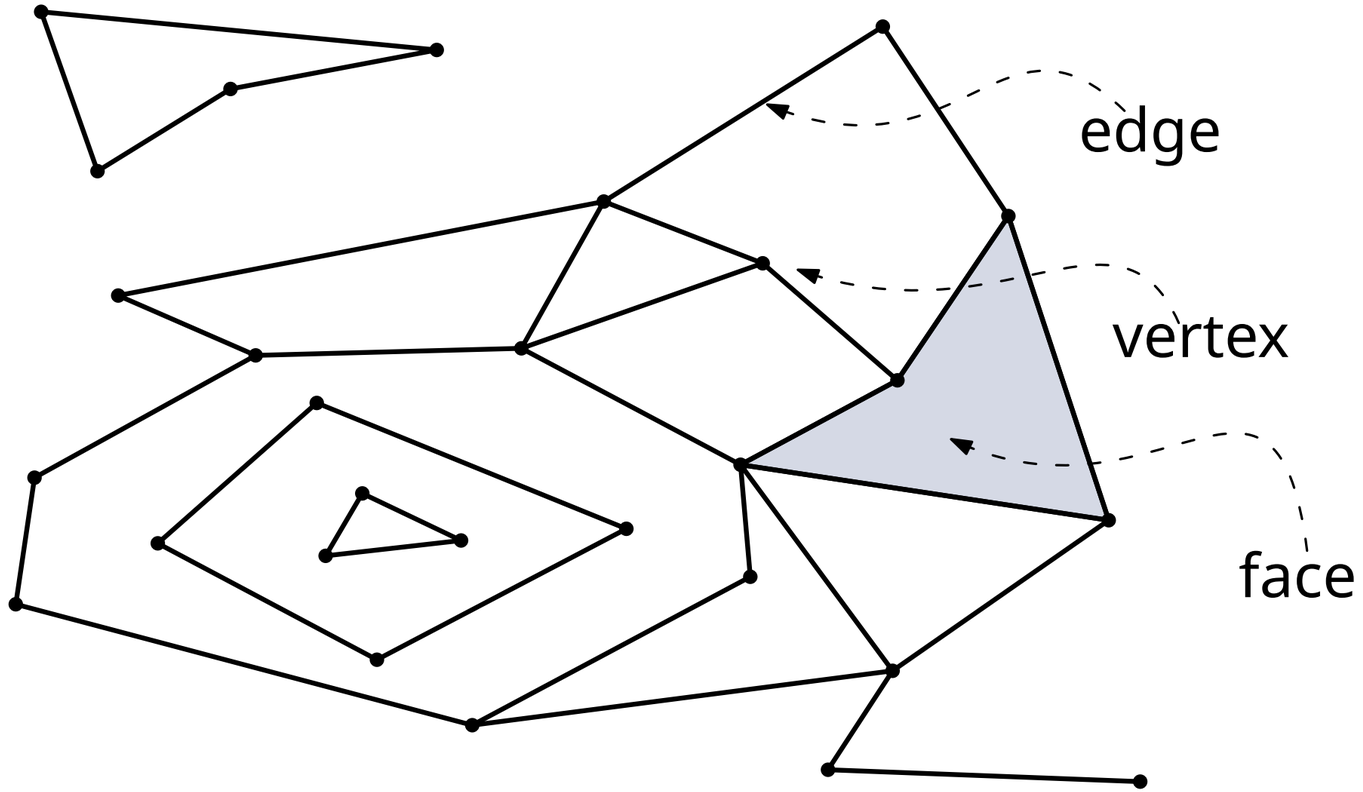
Subdivisions

A planar subdivision is a structure induced by a set of line segments in the plane that can only intersect at common endpoints.



Subdivisions

A planar subdivision is a structure induced by a set of line segments in the plane that can only intersect at common endpoints.



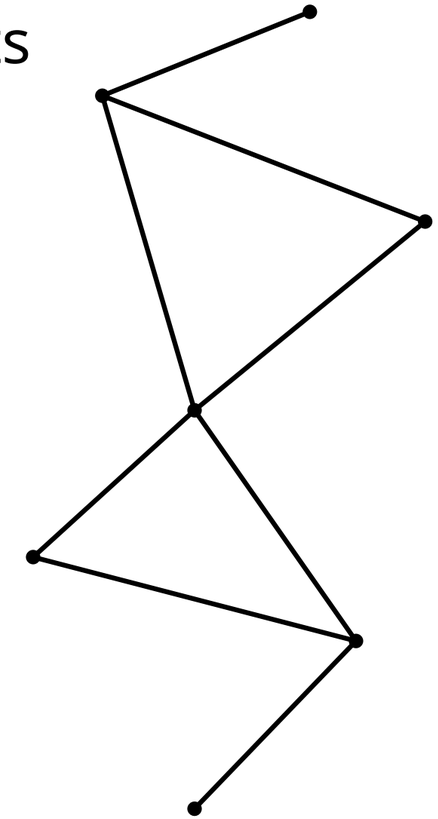
It consists of vertices, edges, and faces

Subdivisions

Vertices are the endpoints of the line segments

Edges are the interiors of the line segments

Faces are the interiors of connected two-dimensional regions that do not contain any point of any line segment

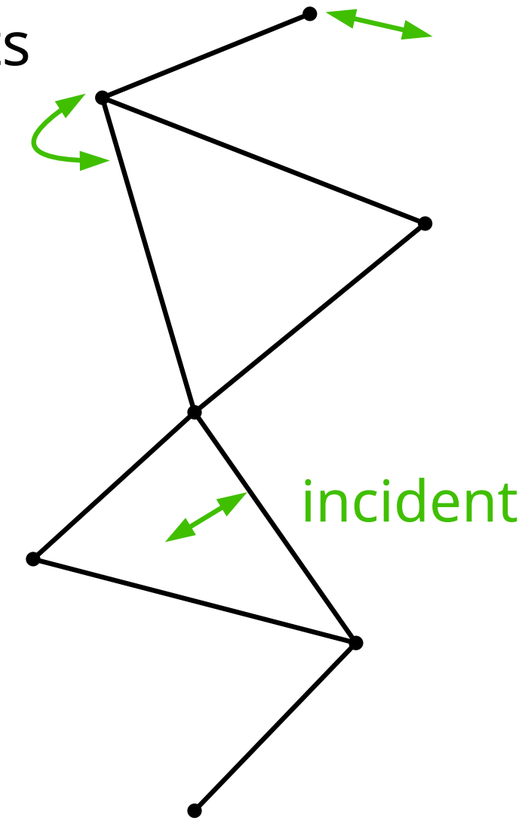


Subdivisions

Vertices are the endpoints of the line segments

Edges are the interiors of the line segments

Faces are the interiors of connected two-dimensional regions that do not contain any point of any line segment



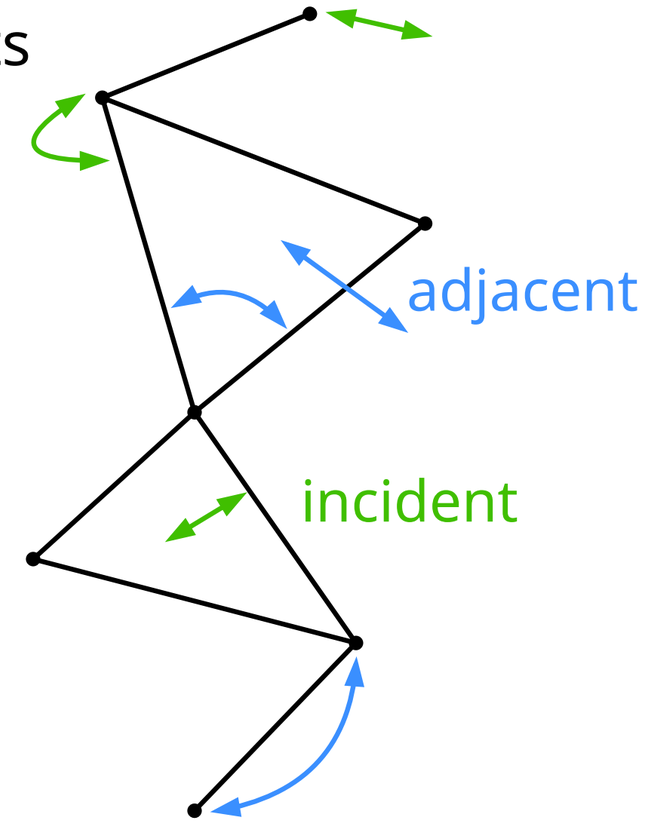
Objects of *different* dimensionality are incident or not

Subdivisions

Vertices are the endpoints of the line segments

Edges are the interiors of the line segments

Faces are the interiors of connected two-dimensional regions that do not contain any point of any line segment

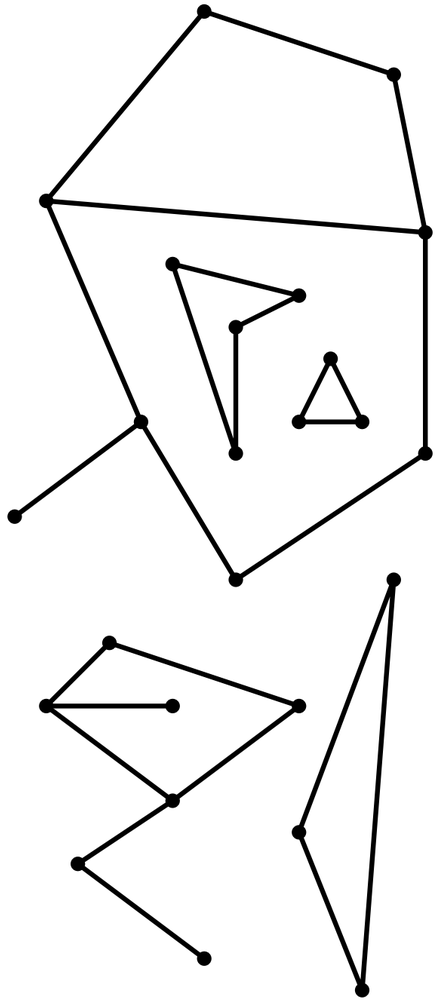


Objects of *different* dimensionality are incident or not

Objects of *the same* dimensionality are adjacent or not

Subdivisions

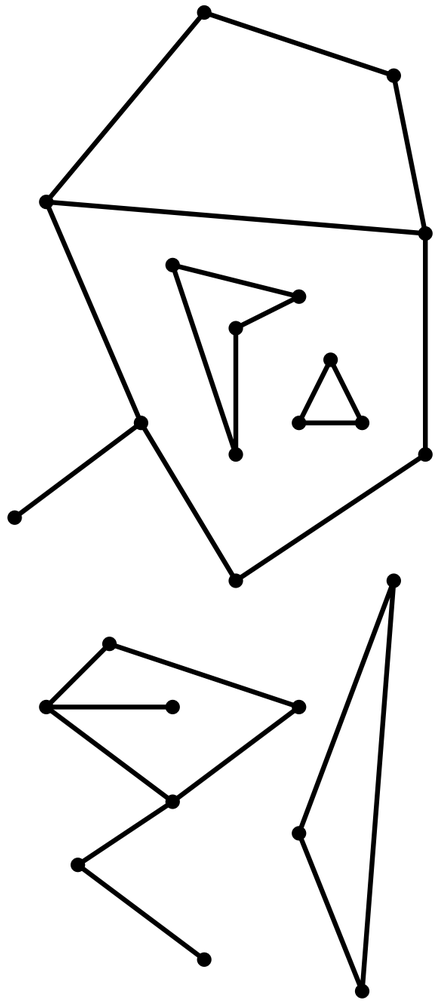
Exactly one face is unbounded, the outer face



Subdivisions

Exactly one face is unbounded, the outer face

Every other face is bounded and has an outer boundary consisting of vertices and edges

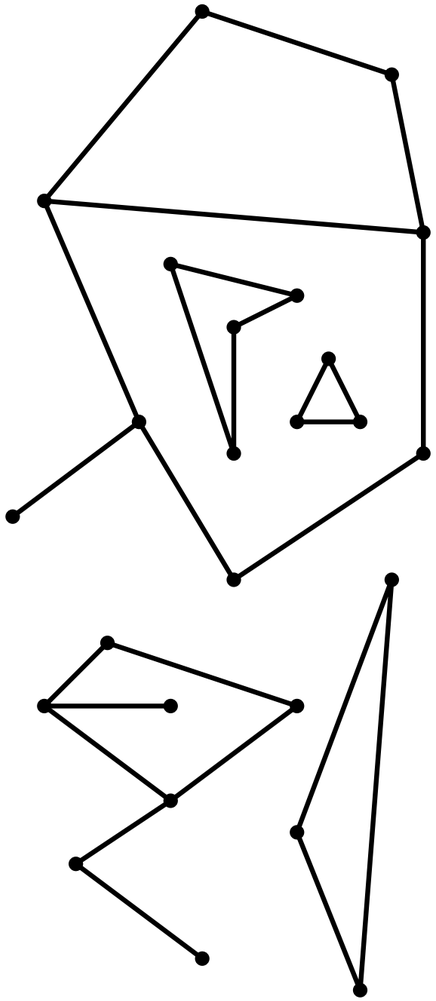


Subdivisions

Exactly one face is unbounded, the outer face

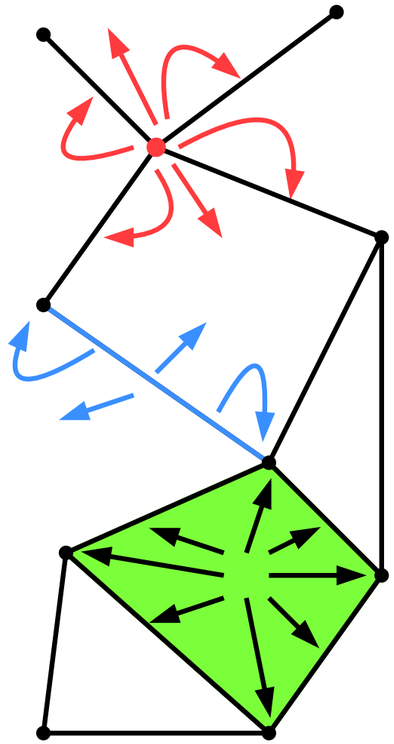
Every other face is bounded and has an outer boundary consisting of vertices and edges

Any face has zero or more inner boundaries



Representing subdivisions

A subdivision representation has a vertex-object class, an edge-object class, and a face-object class

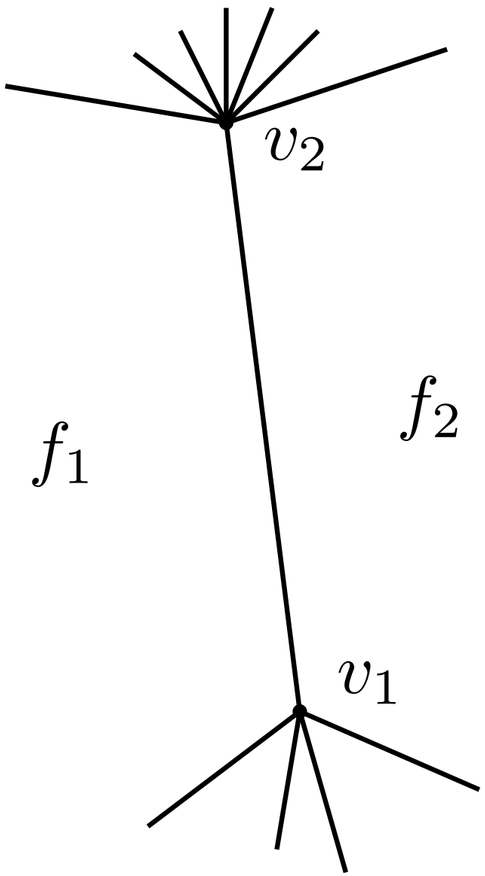


It is a pointer structure where objects can reach incident (or adjacent) objects easily

Representing subdivisions

Use the edge as the central object

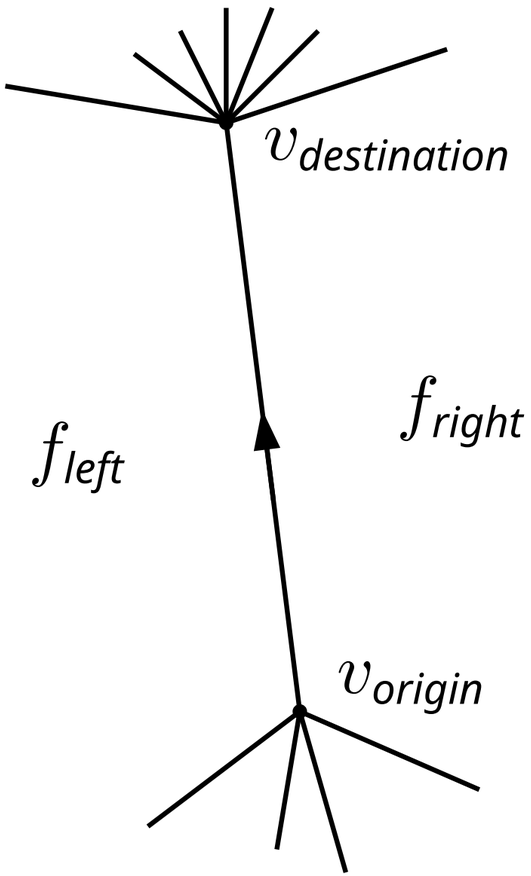
For any edge, exactly two vertices are incident, up to two faces are incident, and zero or more other edges are adjacent



Representing subdivisions

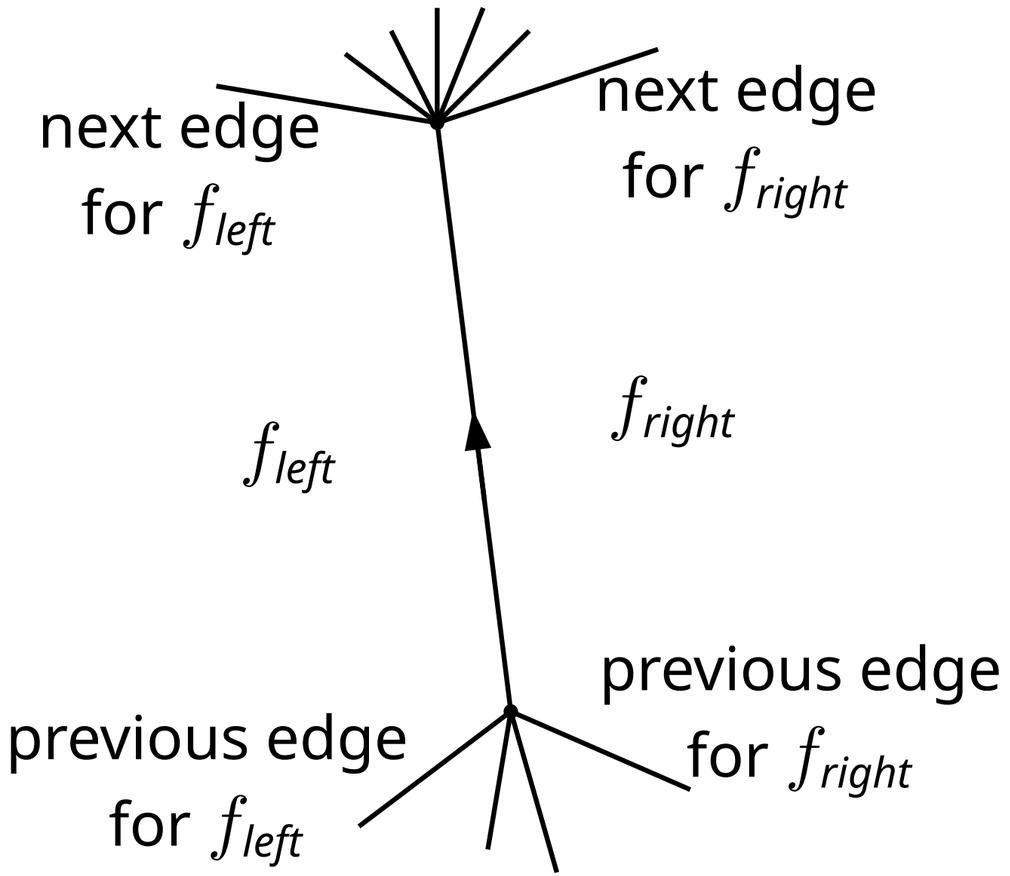
Use the edge as the central object,
and give it a direction

Now we can speak of origin, destination,
left face, and right face

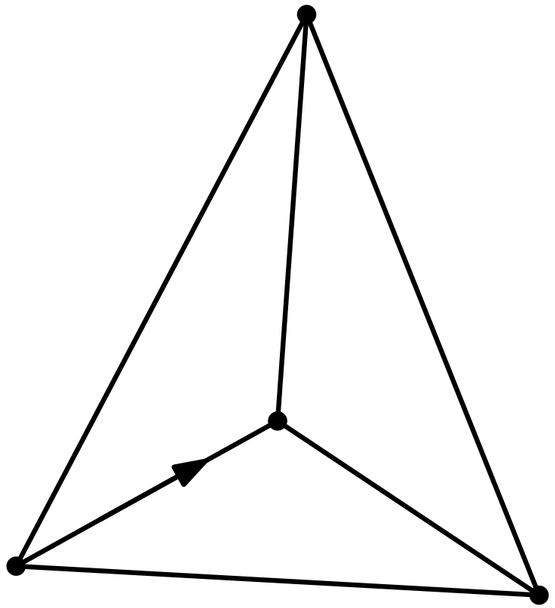


Representing subdivisions

Four edges are of special interest

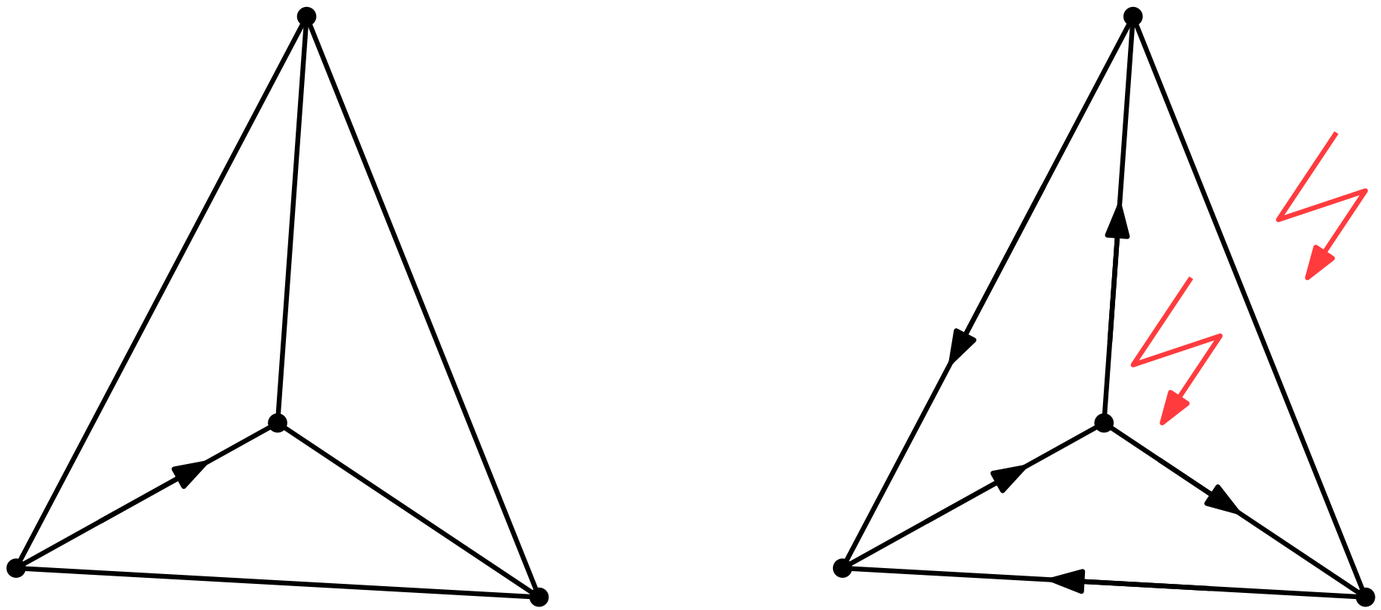


Representing subdivisions



It would be nice if we could traverse a boundary cycle by continuously following the next edge for f_{left} or f_{right}

Representing subdivisions

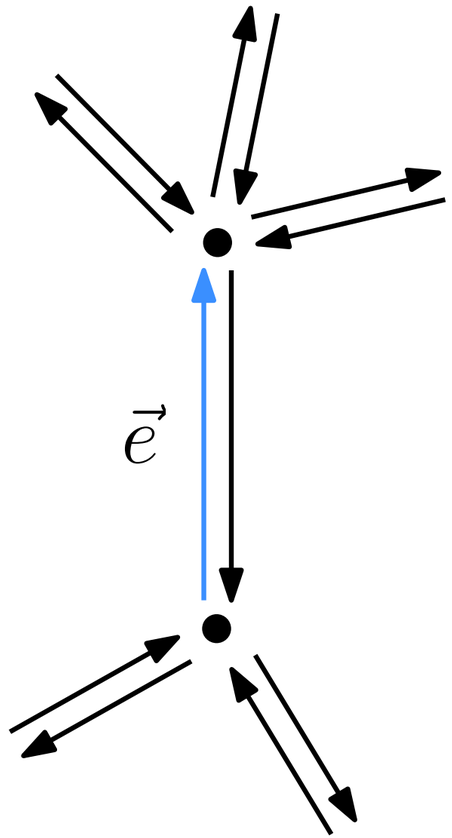


It would be nice if we could traverse a boundary cycle by continuously following the next edge for f_{left} or f_{right}

. . . but, no consistent edge orientation needs to exist

Representing subdivisions

We apply a trick/hack/impossibility: split every edge *length-wise*(!) into two half-edges

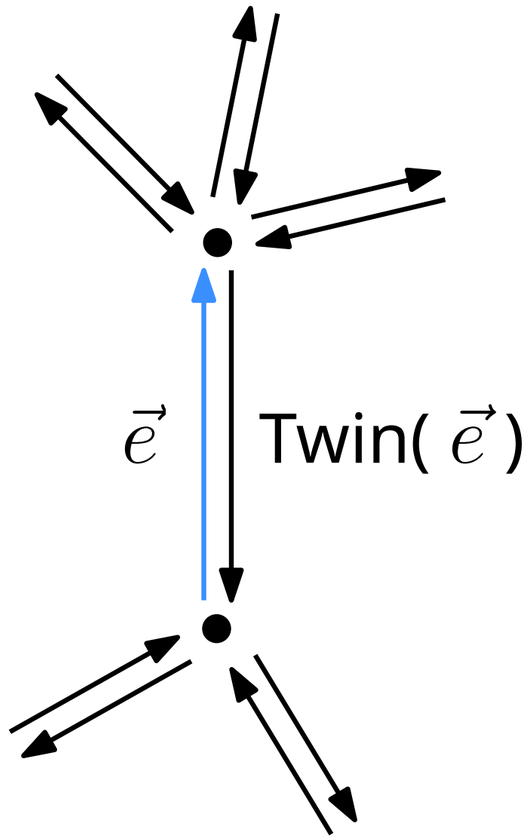


Representing subdivisions

We apply a trick/hack/impossibility: split every edge *length-wise*(!) into two half-edges

Every half-edge:

- has exactly one half-edge as its Twin
- is directed opposite to its Twin

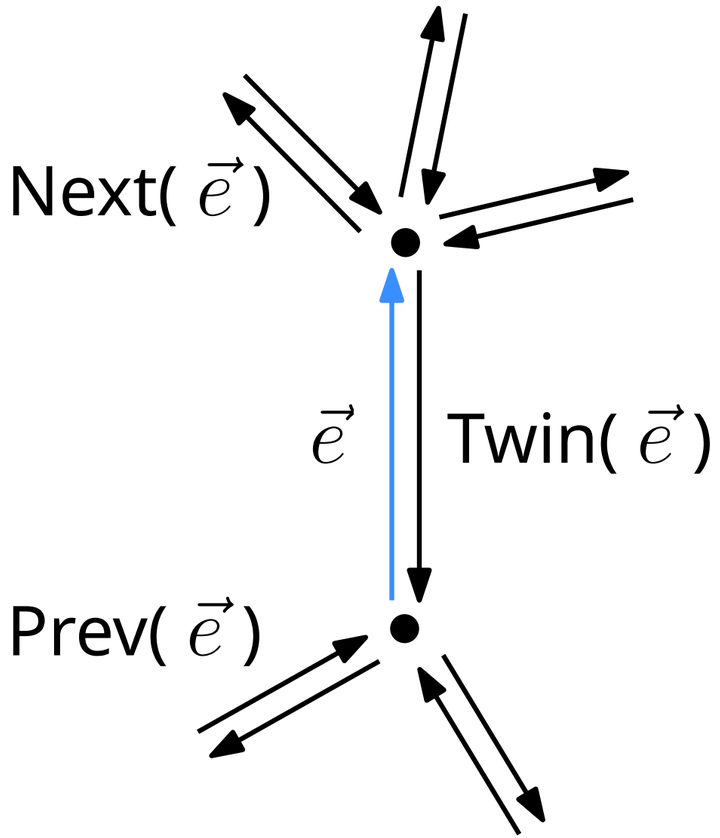


Representing subdivisions

We apply a trick/hack/impossibility: split every edge *length-wise*(!) into two half-edges

Every half-edge:

- has exactly one half-edge as its Twin
- is directed opposite to its Twin
- is incident to only one face (left)



The doubly-connected edge list

The doubly-connected edge list is a subdivision representation structure with an object for every vertex, every half-edge, and every face

A vertex object stores:

- Coordinates
- **IncidentEdge** (some half-edge leaving it)

A half-edge object stores:

- **Origin** (vertex)
- **Twin** (half-edge)
- **IncidentFace** (face)
- **Next** (half-edge in cycle of the incident face)
- **Prev** (half-edge in cycle of the incident face)

The doubly-connected edge list

The doubly-connected edge list is a subdivision representation structure with an object for every vertex, every half-edge, and every face

A vertex object stores:

- Coordinates
- **IncidentEdge** (some half-edge leaving it)

A half-edge object stores:

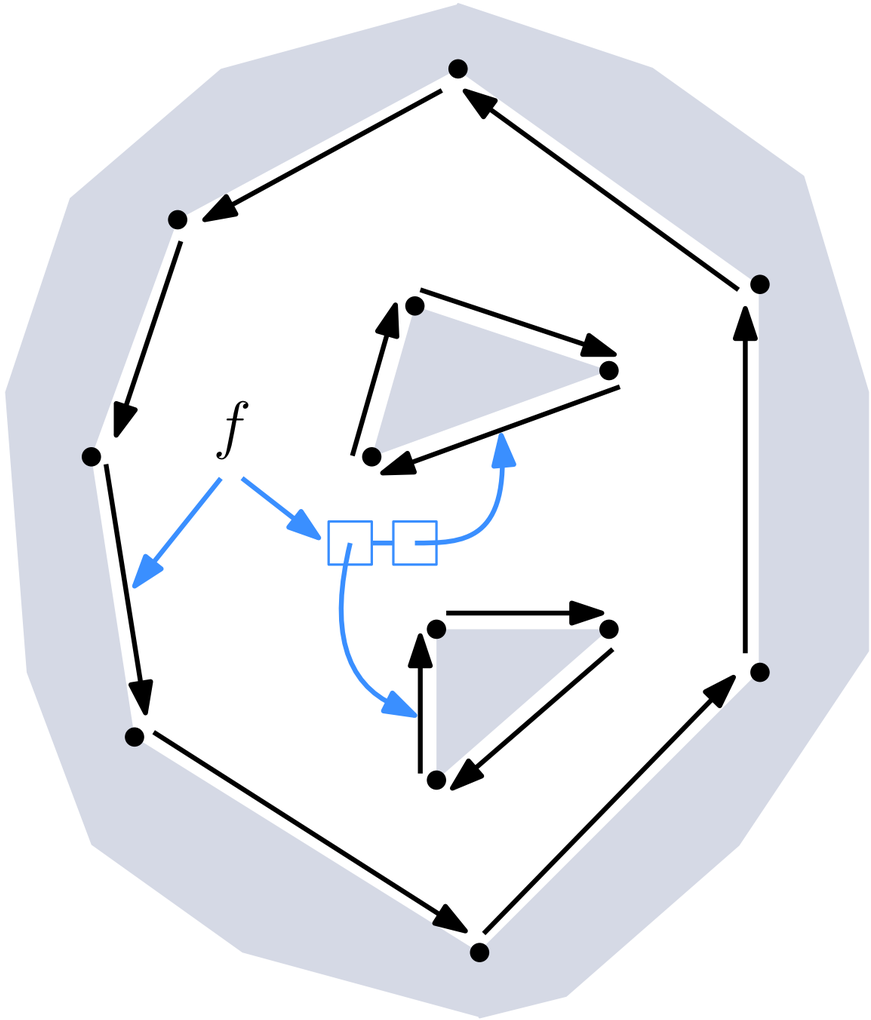
- **Origin** (vertex)
- **Twin** (half-edge)
- **IncidentFace** (face)
- **Next** (half-edge in cycle of the incident face)
- **Prev** (half-edge in cycle of the incident face)

Question: How can we report all incident edges at a vertex?

The doubly-connected edge list

A face object stores:

- **OuterComponent** (half-edge of outer cycle)
- **InnerComponents** (list of half-edges for the inner cycles bounding the face)



The doubly-connected edge list

A face object stores:

- **OuterComponent** (half-edge of outer cycle)
- **InnerComponents** (list of half-edges for the inner cycles bounding the face)

A vertex object stores:

- Coordinates
- **IncidentEdge** (some half-edge leaving it)

A half-edge object stores:

- **Origin** (vertex)
- **Twin** (half-edge)
- **IncidentFace** (face)
- **Next** (half-edge in cycle of the incident face)
- **Prev** (half-edge in cycle of the incident face)

Quiz

Which of the following equalities is not always true?

A: next(previous(e)) = e

B: IncidentFace(e) = IncidentFace(Next(e))

C: Twin(Prev(Twin(e))) = Next(e)

Quiz

Which of the following equalities is not always true?

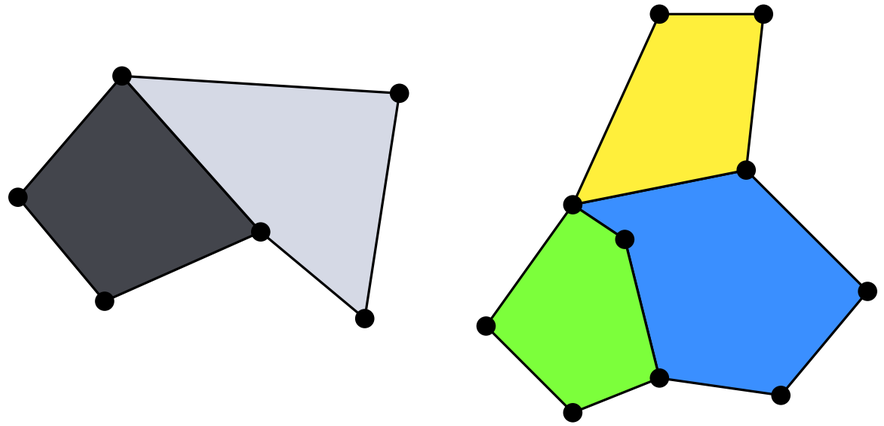
A: next(previous(e)) = e

B: IncidentFace(e) = IncidentFace(Next(e))

C: Twin(Prev(Twin(e))) = Next(e)

Map overlay problem

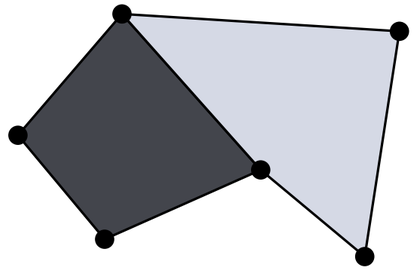
The map overlay problem for two subdivisions S_1 and S_2 is to compute a subdivision S that is the overlay of S_1 and S_2

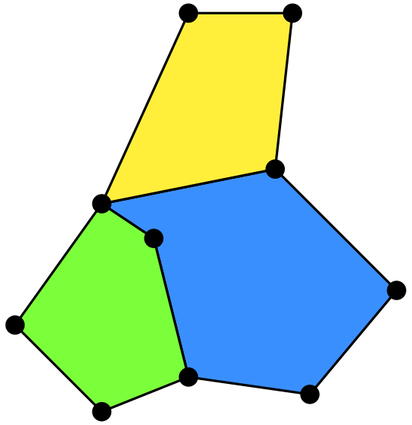


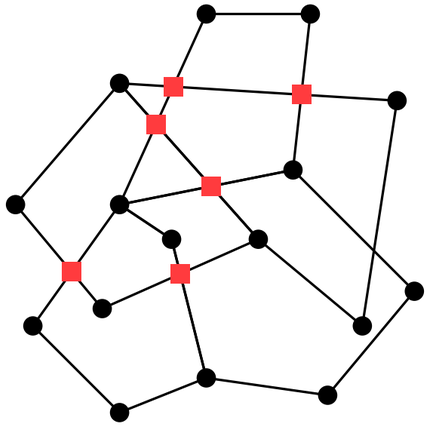
Map overlay problem

The map overlay problem for two subdivisions S_1 and S_2 is to compute a subdivision S that is the overlay of S_1 and S_2

All edges of S are (parts of) edges from S_1 and S_2. All vertices of S are also in S_1 or S_2, or intersections of edges from S_1 and S_2

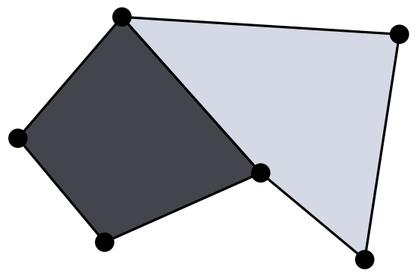


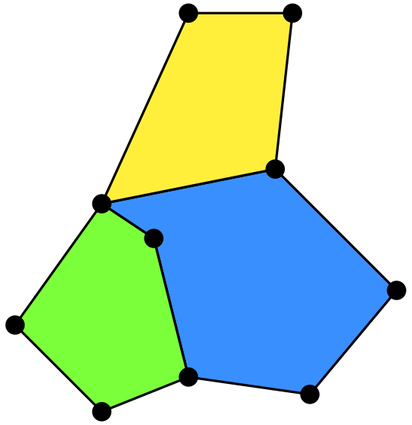




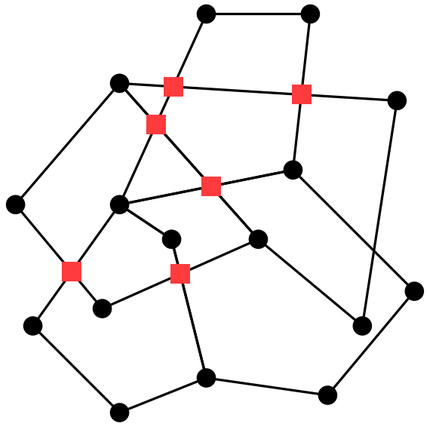
Map overlay problem

The map overlay problem for two subdivisions S_1 and S_2 is to compute a subdivision S that is the overlay of S_1 and S_2





All edges of S are (parts of) edges from S_1 and S_2. All vertices of S are also in S_1 or S_2, or intersections of edges from S_1 and S_2



The line-segment intersection algorithm can be extended to this problem using doubly-connected edge lists.
[Chapter 2.3]

Summary

Application: map overlay

Geometric problem: line segment intersection

Algorithmic technique: Plane sweep

Data structure: Doubly-connected edge list