Plane Sweep Approach

Claudio Mirolo

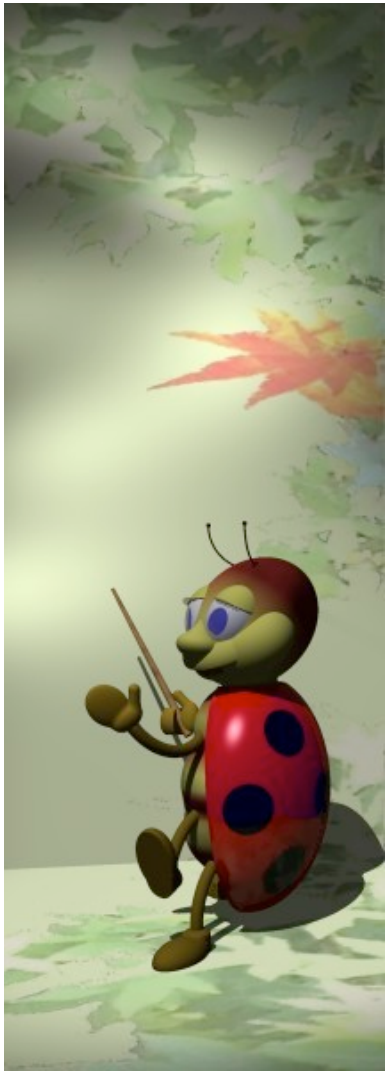
Dip. di Scienze Matematiche, Informatiche e Fisiche
Università di Udine, via delle Scienze 206 – Udine

claudio.mirolo@uniud.it

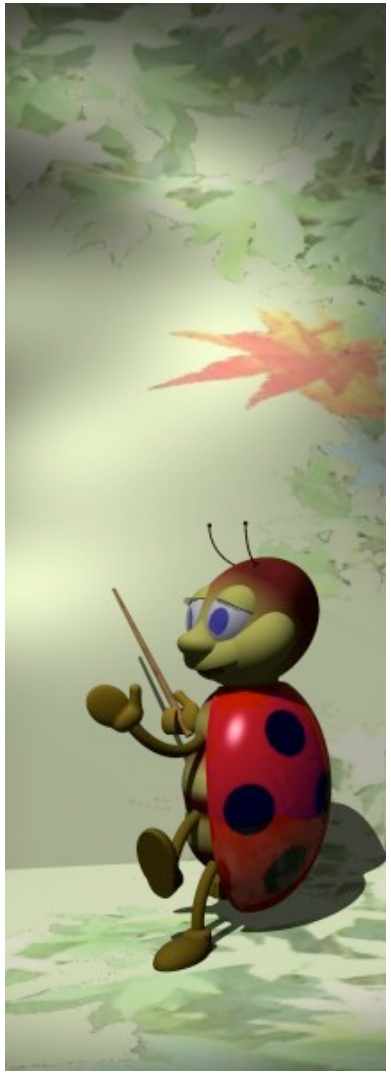
Computational Geometry

users.dimi.uniud.it/∼claudio.mirolo

Outline



Outline



Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs...
  - ...in connection with what it sees
  - critical: important qualitative, e.g. topological, change

Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs...
  - ...in connection with what it sees
  - critical: important qualitative, e.g. topological, change

Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs...
  - ...in connection with what it sees
  - critical: important qualitative, e.g. topological, change

Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs...
  - ...in connection with what it sees
  - critical: important qualitative, e.g. topological, change

Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs. . .
  - . . . in connection with what it sees
  - critical: important qualitative, e.g. topological, change

Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs. . .
  - . . . in connection with what it sees
  - critical: important qualitative, e.g. topological, change

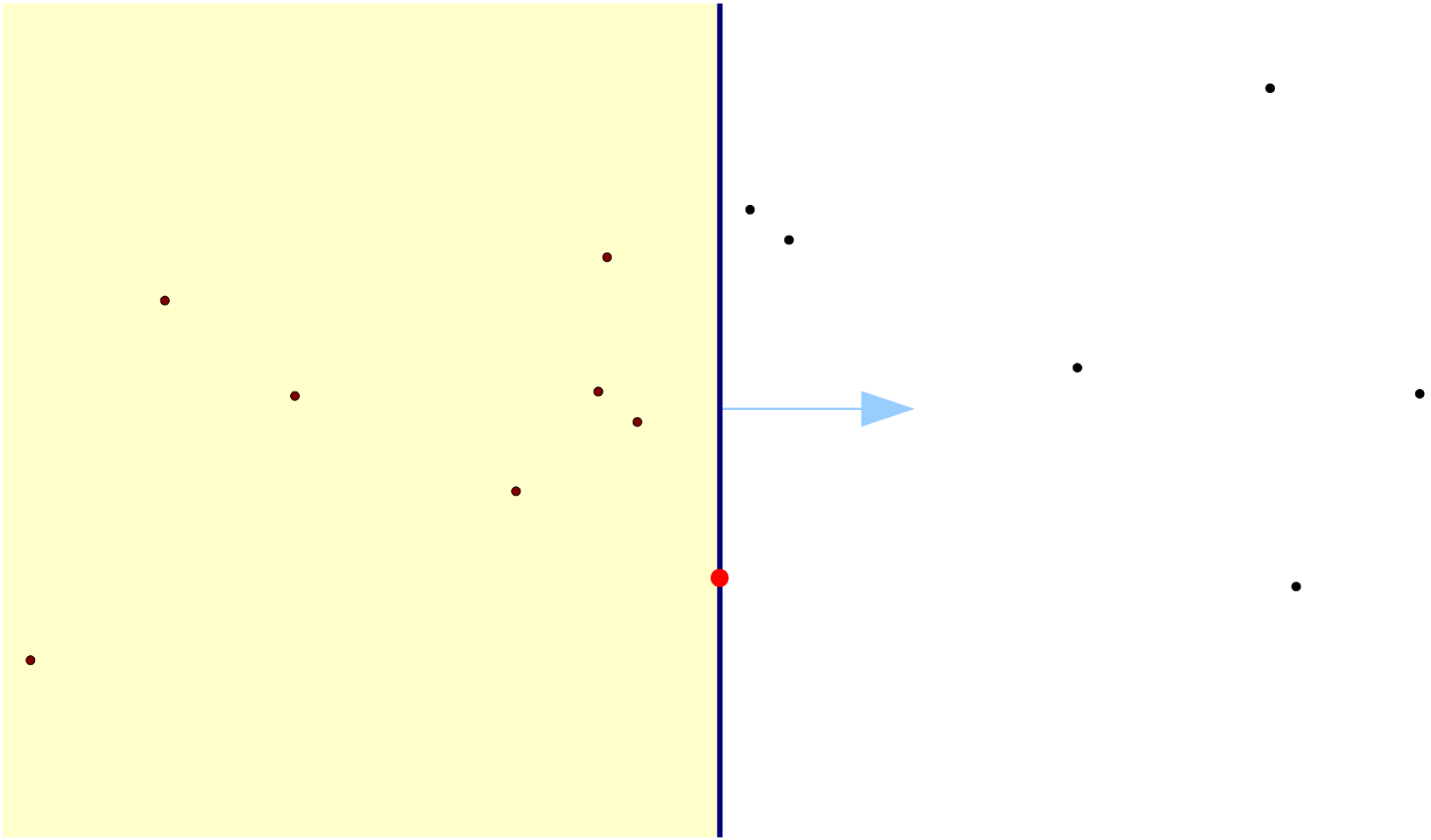
Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs. . .
  - . . . in connection with what it sees
  - critical: important qualitative, e.g. topological, change

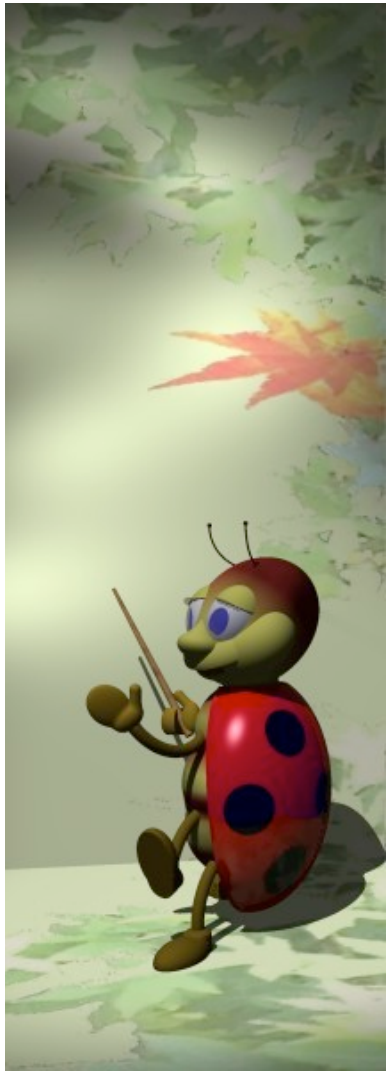
Plane sweep

- *Sweep line*:
  - sweeps the plane, e.g. moving left to right
  - keeps track of what it "sees" about the problem geometry
  - processing completed on the left
  - work in progress on the right
- *Events*:
  - sweep line positions where some critical change occurs. . .
  - . . . in connection with what it sees
  - critical: important qualitative, e.g. topological, change

Plane sweep



Outline



Line segment intersection problem

- Given a set S of n line segments in the plane
- Report all intersection points between segments in S
- Trivially $\Theta(n^2)$ for worst-case arrangements
- Interest in *output-sensitive* algorithms

Line segment intersection problem

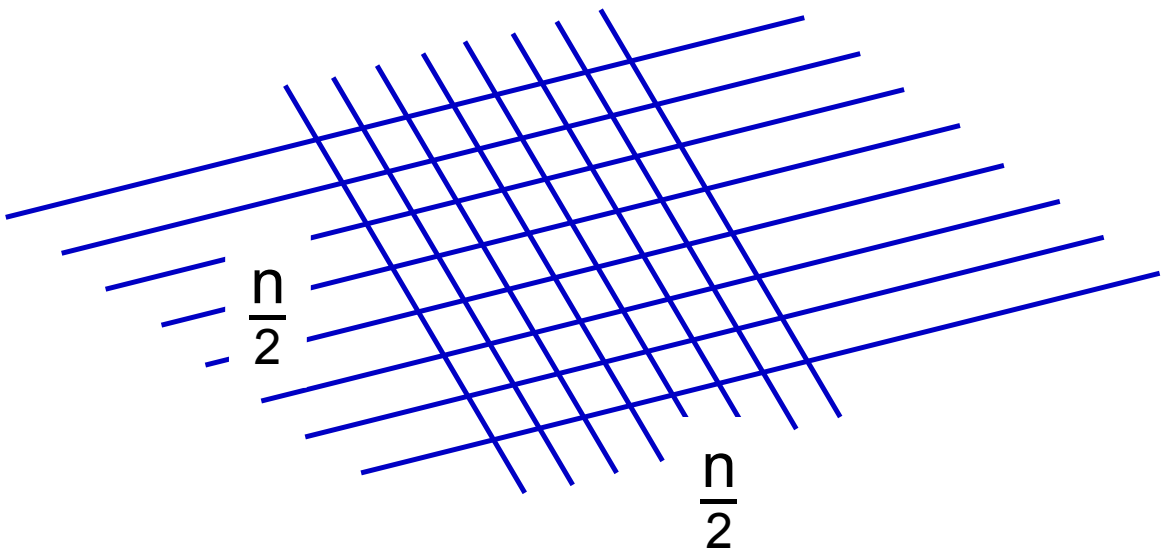
- Given a set S of n line segments in the plane
- Report all intersection points between segments in S
- Trivially $\Theta(n^2)$ for worst-case arrangements
- Interest in *output-sensitive* algorithms

Line segment intersection problem

- Given a set S of n line segments in the plane
- Report all intersection points between segments in S
- Trivially $\Theta(\ n^2\ )$ for worst-case arrangements
- Interest in *output-sensitive* algorithms

Line segment intersection problem

- Given a set S of n line segments in the plane
- Report all intersection points between segments in S
-
-



Line segment intersection problem

- Given a set S of n line segments in the plane
- Report all intersection points between segments in S
- Trivially $\Theta(\ n^2\ )$ for worst-case arrangements
- Interest in *output-sensitive* algorithms

First idea...

- Two line segments *may* intersect only if their projections onto (e.g.) the x-axis overlap
- The sweep line "sees" the line segments it crosses (whose projections do overlap)
- Critical changes occur at segment endpoints:
- *ENTER* and *EXIT* events

First idea...

- Two line segments *may* intersect only if their projections onto (e.g.) the x-axis overlap
- The sweep line "sees" the line segments it crosses (whose projections do overlap)
- Critical changes occur at segment endpoints:
- *ENTER* and *EXIT* events

First idea...

- Two line segments *may* intersect only if their projections onto (e.g.) the x-axis overlap
- The sweep line "sees" the line segments it crosses (whose projections do overlap)
- Critical changes occur at segment endpoints:
- *ENTER* and *EXIT* events

First idea...

- Two line segments *may* intersect only if their projections onto (e.g.) the x-axis overlap
- The sweep line "sees" the line segments it crosses (whose projections do overlap)
- Critical changes occur at segment endpoints:
- *ENTER* and *EXIT* events

Rudimentary algorithm

- At *ENTER* events, check for intersection of the "entering" segment against those crossed by the sweep line
- The list of crossed segments is updated at each event by either adding or removing a line segment
- Sorting endpoints: $O(\, n \log n \,)$
- However: $O(\, n^2 \,)$ checks in the worst case...
- ...even when there are very few intersections!

Rudimentary algorithm

- At *ENTER* events, check for intersection of the "entering" segment against those crossed by the sweep line
- The list of crossed segments is updated at each event by either adding or removing a line segment
- Sorting endpoints: $O(\, n \log n \,)$
- However: $O(\, n^2 \,)$ checks in the worst case...
- ...even when there are very few intersections!

Rudimentary algorithm

- At *ENTER* events, check for intersection of the "entering" segment against those crossed by the sweep line
- The list of crossed segments is updated at each event by either adding or removing a line segment
- Sorting endpoints: $O(\, n \log n \,)$
- However: $O(\, n^2 \,)$ checks in the worst case...
- ...even when there are very few intersections!

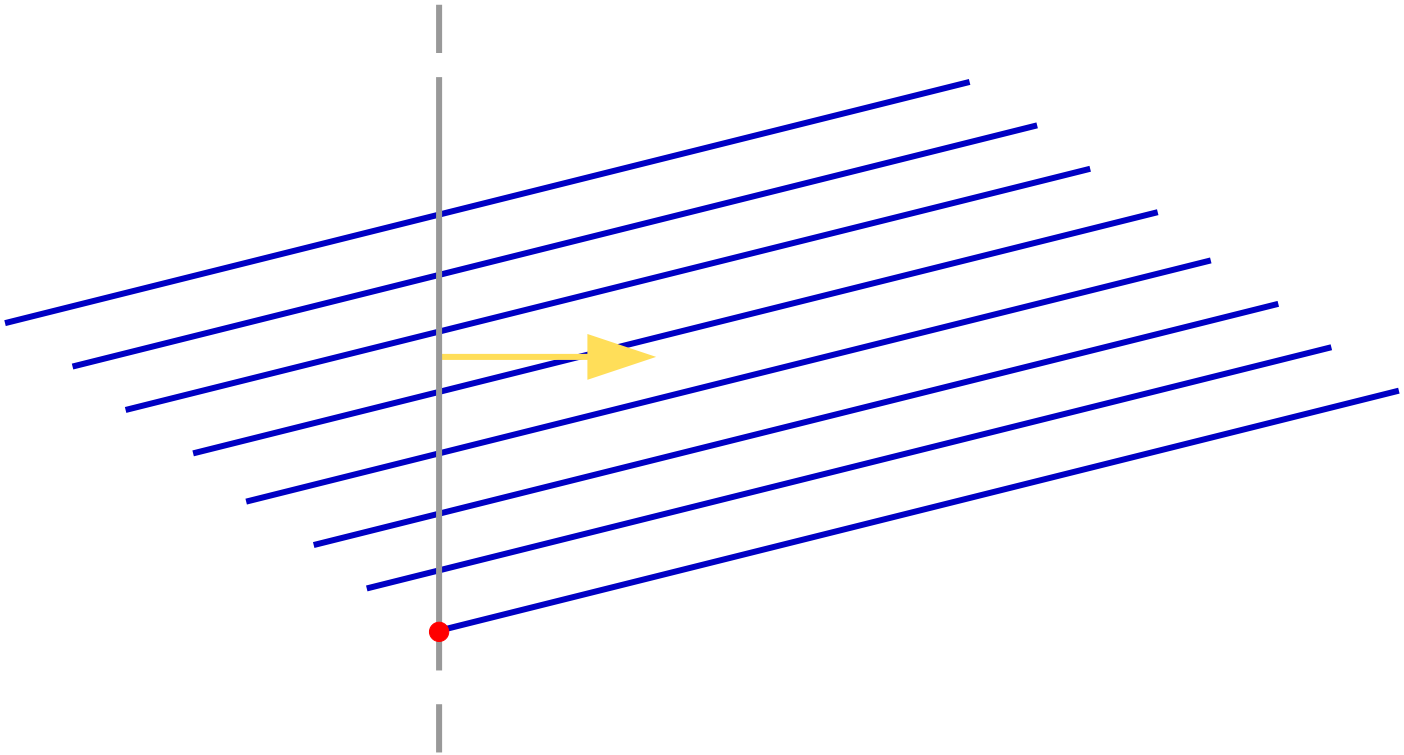
Rudimentary algorithm

- At *ENTER* events, check for intersection of the "entering" segment against those crossed by the sweep line
- The list of crossed segments is updated at each event by either adding or removing a line segment
- Sorting endpoints: $O(\, n \log n \,)$
- However: $O(\, n^2 \,)$ checks in the worst case...
- ...even when there are very few intersections!

Rudimentary algorithm

- At *ENTER* events, check for intersection of the "entering" segment against those crossed by the sweep line
- The list of crossed segments is updated at each event by either adding or removing a line segment
- Sorting endpoints: $O(\, n \log n \,)$
- However: $O(\, n^2 \,)$ checks in the worst case...
- ...even when there are very few intersections!

Rudimentary algorithm: Worst case arrangement



Second idea...

- At stage k, let L_k be the set of points where the sweep line crosses the segments of S
- In order for two line segments to intersect the corresponding points in L_k must be adjacent at some "previous" stage k
- The segments crossing the sweep line at stage k (S_k) can be organized according to the order of points in L_k (along the sweep line)

Second idea...

- At stage k, let L_k be the set of points where the sweep line crosses the segments of S
- In order for two line segments to intersect the corresponding points in L_k must be adjacent at some "previous" stage k
- The segments crossing the sweep line at stage k (S_k) can be organized according to the order of points in L_k (along the sweep line)

Second idea...

- At stage k, let L_k be the set of points where the sweep line crosses the segments of S
- In order for two line segments to intersect the corresponding points in L_k must be adjacent at some "previous" stage k
- The segments crossing the sweep line at stage k (S_k) can be organized according to the order of points in L_k (along the sweep line)

Shamos & Hoey (1976)

- Sort endpoints of line segments in S: $O(\ n \log n\ )$
- Arrange S_k as a balanced search tree (initially empty)
- *ENTER* event — add $s \in S$ into S_k
  when the sweep line reaches its left endpoint: $O(\ \log n\ )$
- *EXIT* event — remove $s \in S$ from S_k
  when the sweep line reaches its right endpoint: $O(\ \log n\ )$

Shamos & Hoey (1976)

- Sort endpoints of line segments in S: $O(\, n \log n \,)$
- Arrange S_k as a balanced search tree (initially empty)
- *ENTER* event — add $s \in S$ into S_k
  when the sweep line reaches its left endpoint: $O(\, \log n \,)$
- *EXIT* event — remove $s \in S$ from S_k
  when the sweep line reaches its right endpoint: $O(\, \log n \,)$

Shamos & Hoey (1976)

- Sort endpoints of line segments in S: $O(\, n \log n \,)$
- Arrange S_k as a balanced search tree (initially empty)
- *ENTER* event — add $s \in S$ into S_k
  when the sweep line reaches its left endpoint: $O(\, \log n \,)$
- *EXIT* event — remove $s \in S$ from S_k
  when the sweep line reaches its right endpoint: $O(\, \log n \,)$

Shamos & Hoey (1976)

- Sort endpoints of line segments in S: $O(\, n \log n \,)$
- Arrange S_k as a balanced search tree (initially empty)
- *ENTER* event — add $s \in S$ into S_k
  when the sweep line reaches its left endpoint: $O(\, \log n \,)$
- *EXIT* event — remove $s \in S$ from S_k
  when the sweep line reaches its right endpoint: $O(\, \log n \,)$

Shamos & Hoey (1976)

- Sort endpoints of line segments in S: $O(\, n \log n \,)$
- Arrange S_k as a balanced search tree (initially empty)
- *ENTER* event — add $s \in S$ into S_k
  when the sweep line reaches its left endpoint: $O(\, \log n \,)$
- *EXIT* event — remove $s \in S$ from S_k
  when the sweep line reaches its right endpoint: $O(\, \log n \,)$

Shamos & Hoey (1976)

- Sort endpoints of line segments in S: $O(\ n \log n\ )$
- Arrange S_k as a balanced search tree (initially empty)
- *ENTER* event — add $s \in S$ into S_k
  when the sweep line reaches its left endpoint: $O(\ \log n\ )$
- *EXIT* event — remove $s \in S$ from S_k
  when the sweep line reaches its right endpoint: $O(\ \log n\ )$

Intersection test

- When can new pairs of adjacent segments in S_k be found?
- *ENTER* event — test s for intersection with u and v if s is to be inserted into S_k between u and v: $O(\log n)$
- *EXIT* event — test u for intersection with v if s is found in S_k between u and v: $O(\log n)$
- Overall $2n$ steps: $O(n) \times O(\log n)$

Intersection test

- When can new pairs of adjacent segments in S_k be found?
- *ENTER* event — test s for intersection with u and v if s is to be inserted into S_k between u and v: $O(\log n)$
- *EXIT* event — test u for intersection with v if s is found in S_k between u and v: $O(\log n)$
- Overall $2n$ steps: $O(n) \times O(\log n)$

Intersection test

- When can new pairs of adjacent segments in S_k be found?
- *ENTER* event — test s for intersection with u and v if s is to be inserted into S_k between u and v: $O(\log n)$
- *EXIT* event — test u for intersection with v if s is found in S_k between u and v: $O(\log n)$
- Overall $2n$ steps: $O(n) \times O(\log n)$

Intersection test

- When can new pairs of adjacent segments in S_k be found?
- *ENTER* event — test s for intersection with u and v
  if s is to be inserted into S_k between u and v: $O(\log n)$
- *EXIT* event — test u for intersection with v
  if s is found in S_k between u and v: $O(\log n)$
- Overall $2n$ steps: $O(n) \times O(\log n)$

Intersection test

- When can new pairs of adjacent segments in S_k be found?
- *ENTER* event — test s for intersection with u and v
  if s is to be inserted into S_k between u and v: $O(\log n)$
- *EXIT* event — test u for intersection with v
  if s is found in S_k between u and v: $O(\log n)$
- Overall $2n$ steps: $O(n) \times O(\log n)$

Intersection test

- When can new pairs of adjacent segments in S_k be found?
- *ENTER* event — test s for intersection with u and v
  if s is to be inserted into S_k between u and v: $O(\log n)$
- *EXIT* event — test u for intersection with v
  if s is found in S_k between u and v: $O(\log n)$
- Overall $2n$ steps: $O(n) \times O(\log n)$

Operations on the tree structure

- $insert(s, T)$
- $find(s, T)$
- $remove(s, T)$
- $previous(s, T)$
- $next(s, T)$
- All running in $O(\log |T|) = O(\log n)$

Operations on the tree structure

- $insert(s, T)$
- $find(s, T)$
- $remove(s, T)$
- $previous(s, T)$
- $next(s, T)$
- All running in $O(\log |T|) = O(\log n)$

Operations on the tree structure

- $insert(s, T)$
- $find(s, T)$
- $remove(s, T)$
- $previous(s, T)$
- $next(s, T)$
- All running in $O(\log |T|) = O(\log n)$

Optimal intersection test algorithm

- Shamos & Hoey's $O(\, n \log n \,)$ algorithm for intersection test ...
- Do any two line segments in S intersect? (and if they do, report a "witness" pair)
- ... is optimal: $\Omega(\, n \log n \,)$ lower bound (worst case)
- *Element uniqueness* (see Dobkin & Lipton, 1975; 1979) can be reduced to this problem

Optimal intersection test algorithm

- Shamos & Hoey's $O(\, n \log n \,)$ algorithm for intersection test ...
- Do any two line segments in S intersect? (and if they do, report a "witness" pair)
- ... is optimal: $\Omega(\, n \log n \,)$ lower bound (worst case)
- *Element uniqueness* (see Dobkin & Lipton, 1975; 1979) can be reduced to this problem

Optimal intersection test algorithm

- Shamos & Hoey's $O(\, n \log n \,)$ algorithm for intersection test ...
- Do any two line segments in S intersect? (and if they do, report a "witness" pair)
- ... is optimal: $\Omega(\, n \log n \,)$ lower bound (worst case)
- *Element uniqueness* (see Dobkin & Lipton, 1975; 1979) can be reduced to this problem

Optimal intersection test algorithm

- Shamos & Hoey's $O(\, n \log n \,)$ algorithm for intersection test ...
- Do any two line segments in S intersect? (and if they do, report a "witness" pair)
- ... is optimal: $\Omega(\, n \log n \,)$ lower bound (worst case)
- *Element uniqueness* (see Dobkin & Lipton, 1975; 1979) can be reduced to this problem

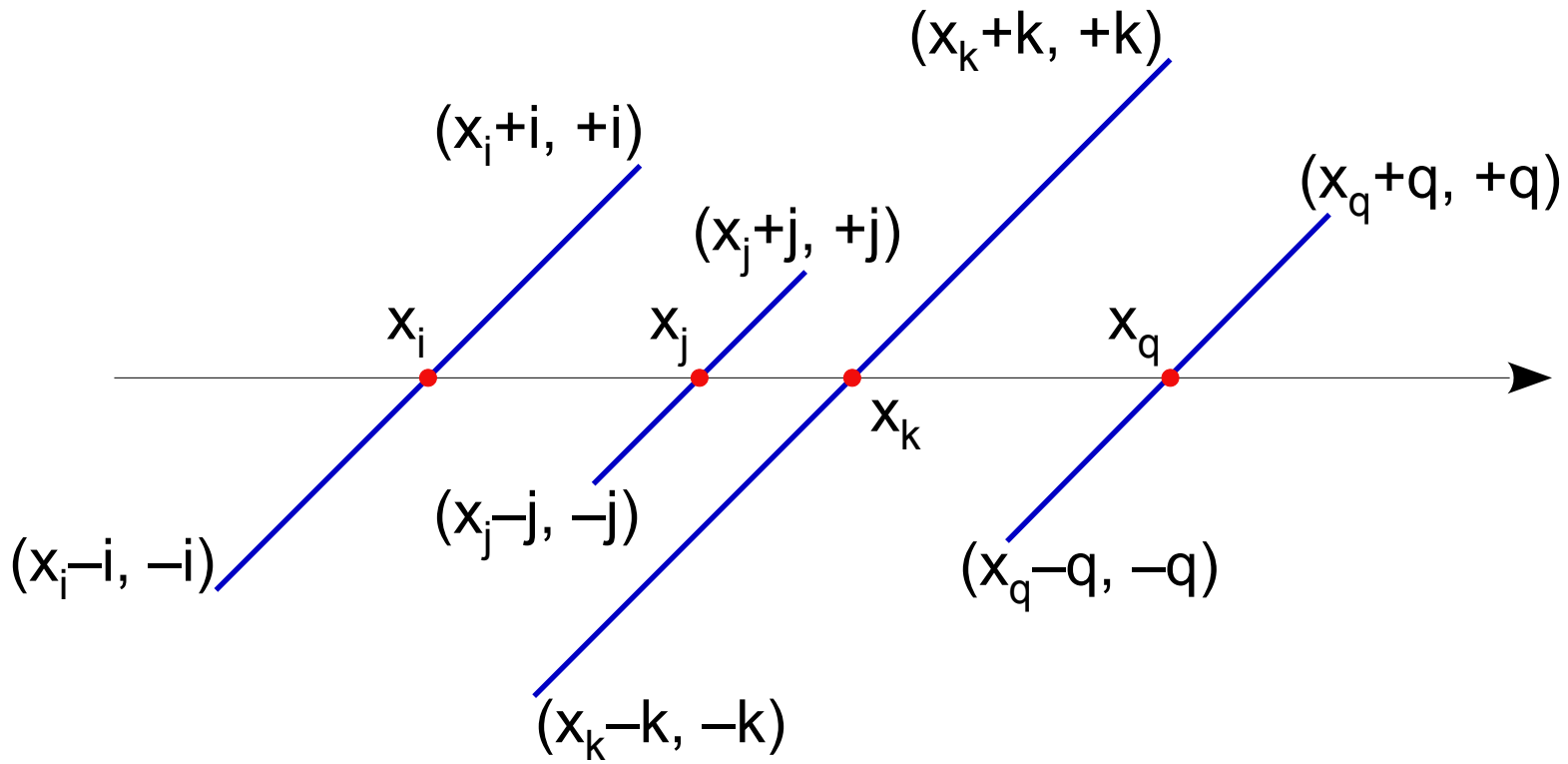
Optimal intersection test algorithm

- Shamos & Hoey's $O(\, n \log n \,)$ algorithm for intersection test ...
- Do any two line segments in S intersect? (and if they do, report a "witness" pair)
- ... is optimal: $\Omega(\, n \log n \,)$ lower bound (worst case)
- *Element uniqueness* (see Dobkin & Lipton, 1975; 1979) can be reduced to this problem

Optimal intersection test algorithm

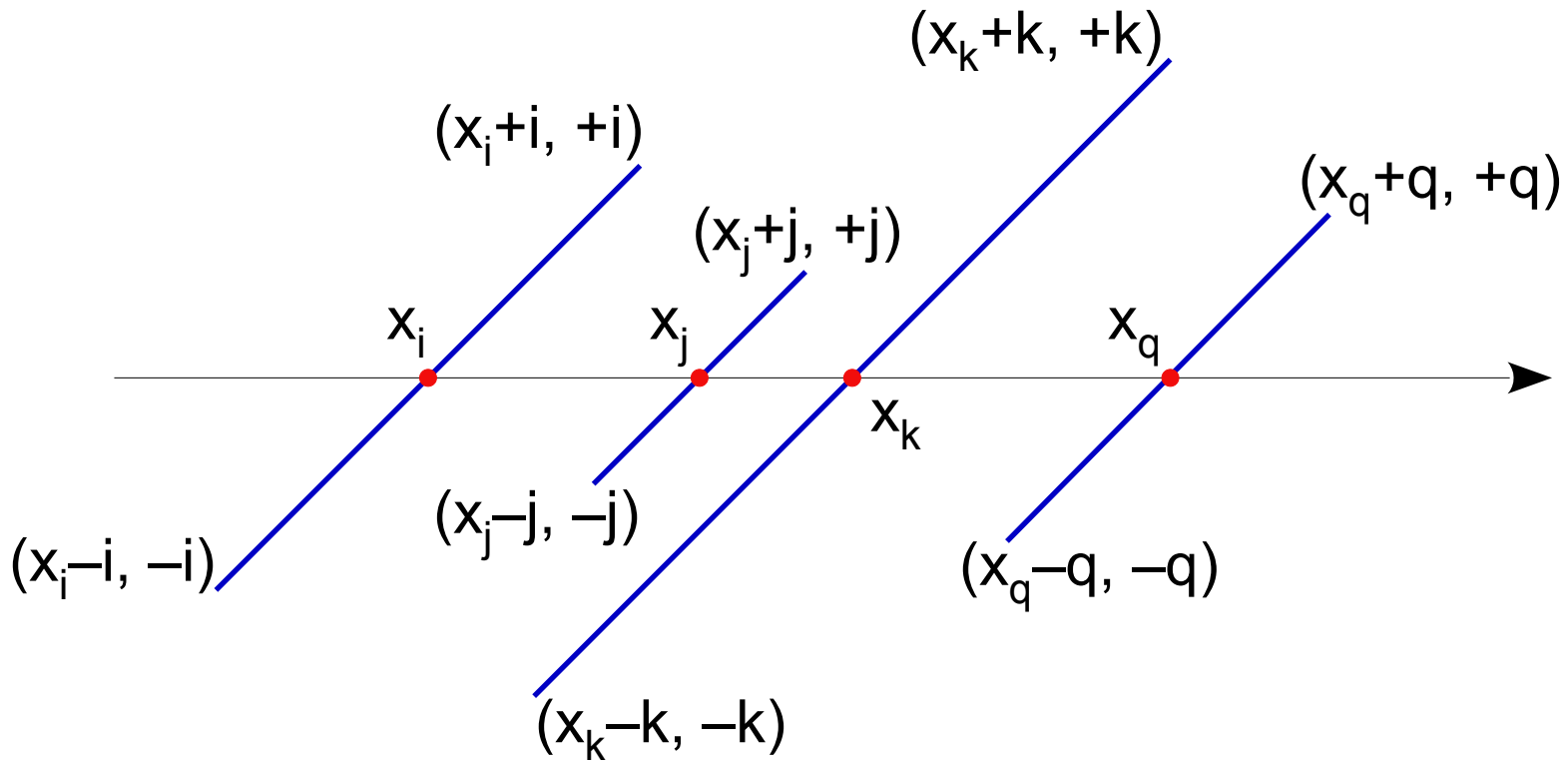
- Shamos & Hoey's $O(\, n \log n \,)$ algorithm for intersection test ...
- Do any two line segments in S intersect? (and if they do, report a "witness" pair)
- ... is optimal: $\Omega(\, n \log n \,)$ lower bound (worst case)
- *Element uniqueness* (see Dobkin & Lipton, 1975; 1979) can be reduced to this problem

Problem reduction to "Uniqueness"



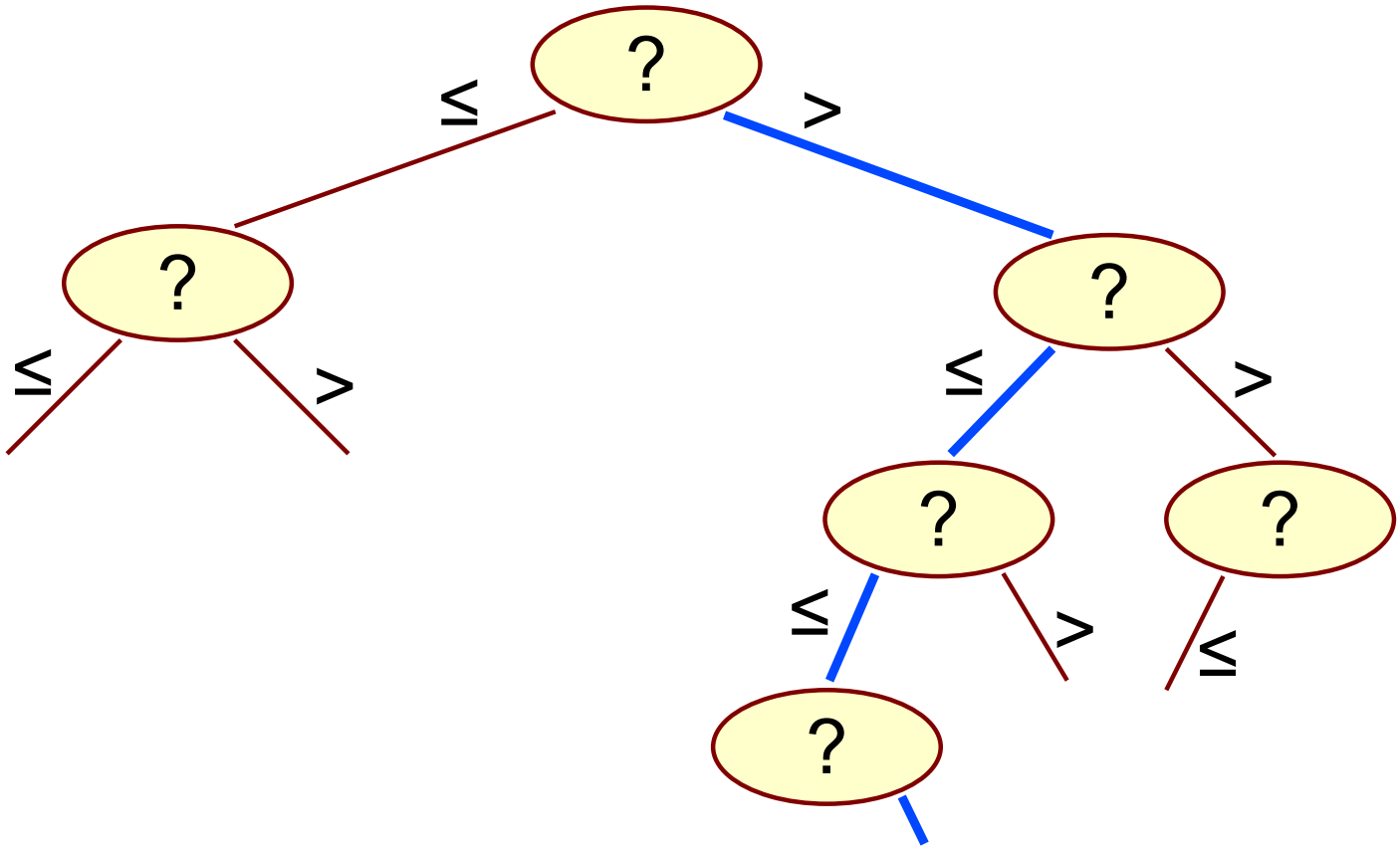
segment endpoints are all distinct, but segments overlap if x_i is the same

Problem reduction to "Uniqueness"



segment endpoints are all distinct, but segments overlap if x_i is the same

"Uniqueness": Algorithm decision tree



"Uniqueness": A bit of notation

- Collection of n numbers (linear order: <, =, >)
- $< x_1,\ x_2,\ \ldots\ x_n >$ and $< y_1,\ y_2,\ \ldots\ y_n >$ are two permutations of the same n-tuple
- $< z_1,\ z_2,\ \ldots\ z_n >$ such that $z_i = \alpha x_i + (1-\alpha) y_i$, where $\alpha \in [0,1]$
- i.e., $z_i = z_i(\alpha)$

"Uniqueness": A bit of notation

- Collection of n numbers (linear order: <, =, >)
- $< x_1,\ x_2,\ \ldots\ x_n >$ and $< y_1,\ y_2,\ \ldots\ y_n >$ are two permutations of the same n-tuple
- $< z_1,\ z_2,\ \ldots\ z_n >$ such that $z_i = \alpha x_i + (1-\alpha) y_i$, where $\alpha \in [0,1]$
- i.e., $z_i = z_i(\alpha)$

"Uniqueness": A bit of notation

- Collection of n numbers (linear order: <, =, >)
- $< x_1,\ x_2,\ \ldots\ x_n >$ and $< y_1,\ y_2,\ \ldots\ y_n >$ are two permutations of the same n-tuple
- $< z_1,\ z_2,\ \ldots\ z_n >$ such that $z_i = \alpha x_i + (1 - \alpha) y_i$, where $\alpha \in [0, 1]$
- i.e., $z_i = z_i(\alpha)$

"Uniqueness": A bit of notation

- Collection of n numbers (linear order: <, =, >)
- $< x_1,\ x_2,\ \ldots\ x_n >$ and $< y_1,\ y_2,\ \ldots\ y_n >$ are two permutations of the same n-tuple
- $< z_1,\ z_2,\ \ldots\ z_n >$ such that $z_i = \alpha x_i + (1 - \alpha) y_i$, where $\alpha \in [0, 1]$
- i.e., $z_i = z_i(\alpha)$

"Uniqueness": Property

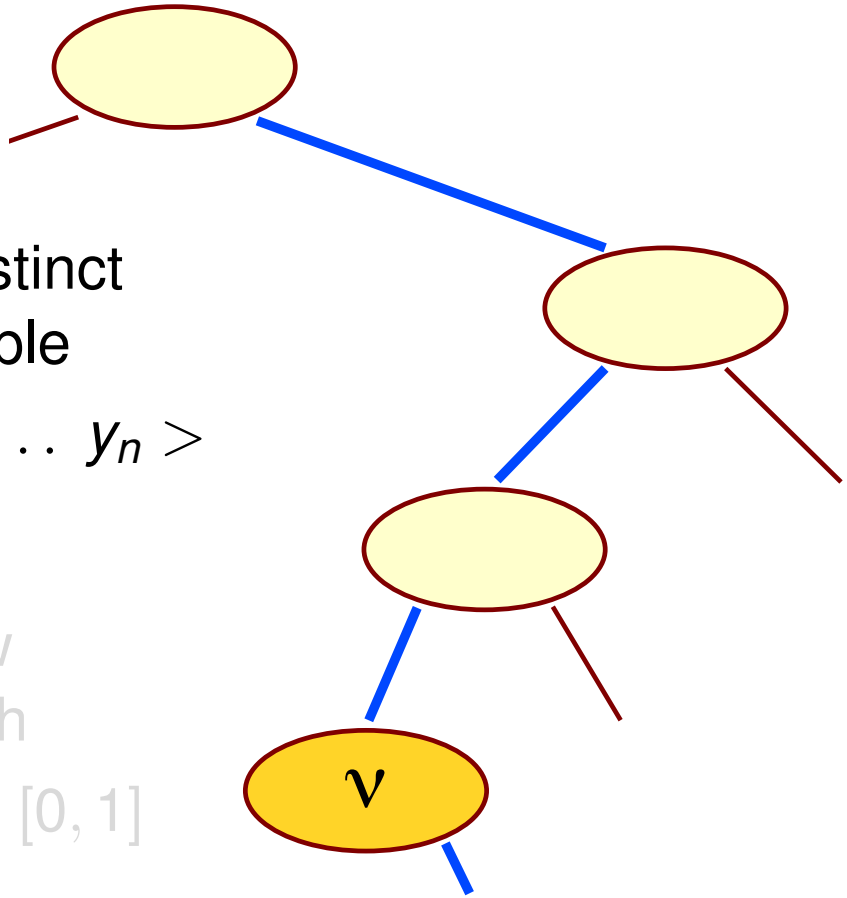
If the decision paths for two distinct permutations of the same n-tuple

$< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$

both reach node ν,

then the algorithm would follow the same path to ν starting with

$< z_1, z_2, \ldots z_n >$ for any $\alpha \in [0, 1]$



"Uniqueness": Property

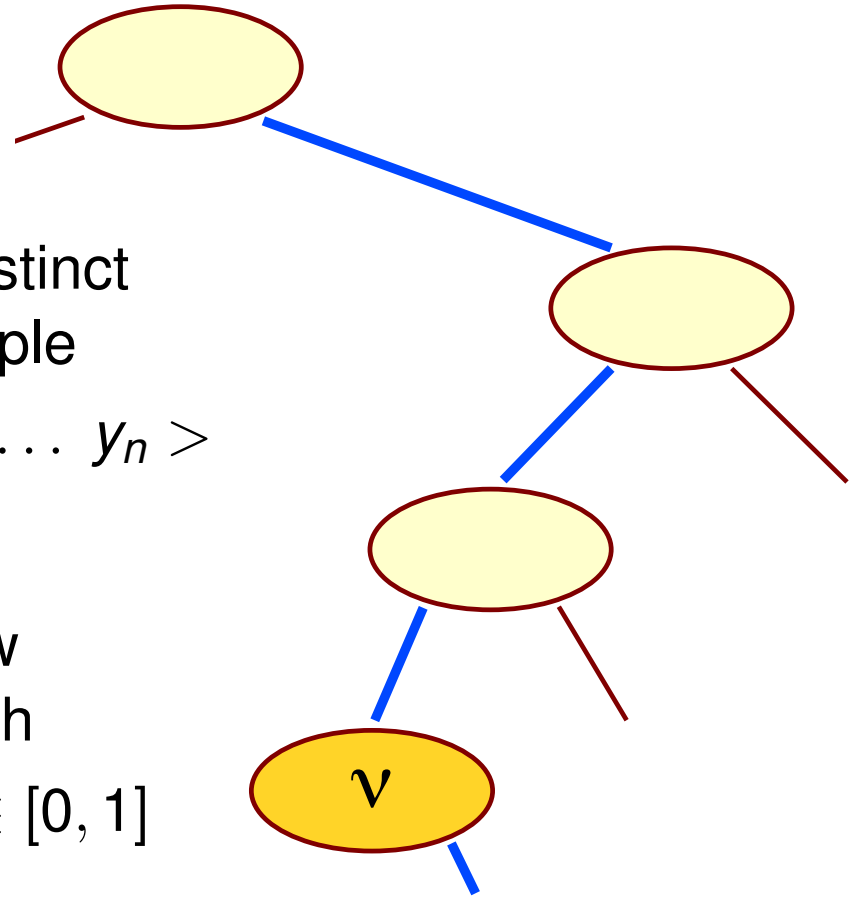
If the decision paths for two distinct permutations of the same n-tuple

$< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$

both reach node ν,

then the algorithm would follow the same path to ν starting with

$< z_1, z_2, \ldots z_n >$ for any $\alpha \in [0, 1]$

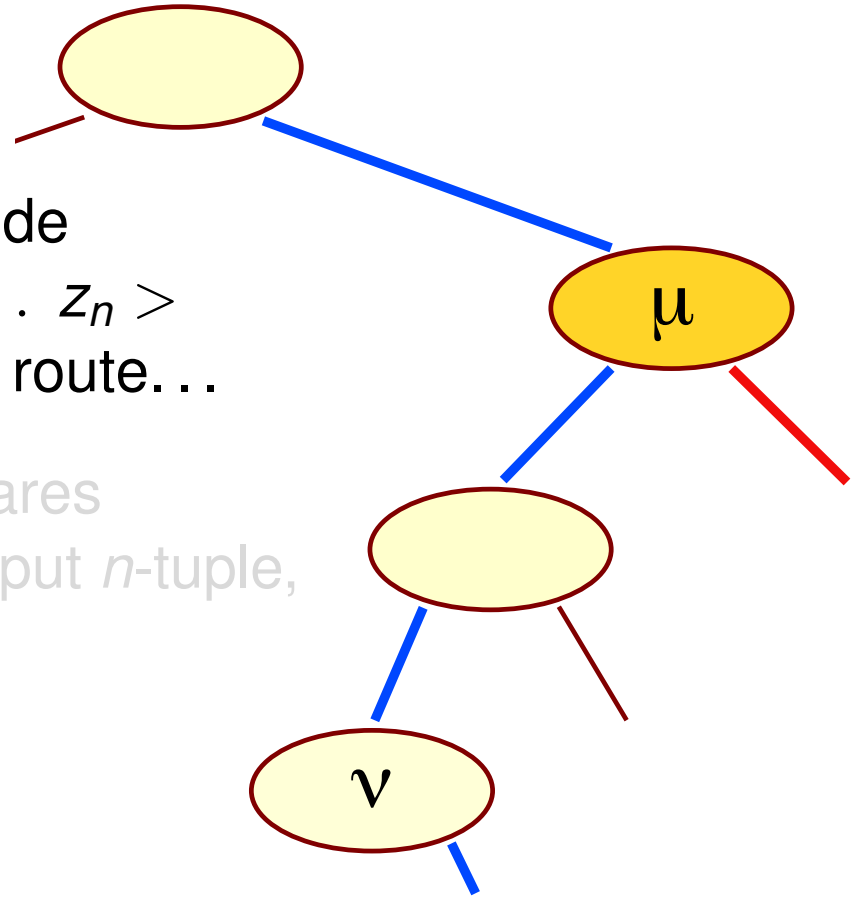


"Uniqueness": Contradiction

If not so, let μ be ν's parent node
where the path for $< z_1, z_2, \ldots z_n >$
is supposed to take a different route...

At node μ the algorithm compares
the i-th and j-th items of the input n-tuple,

but since $z_i = \alpha x_i + (1 - \alpha) y_i$
and $z_j = \alpha x_j + (1 - \alpha) y_j$,
comparing z_i and z_j cannot
result in a different decision

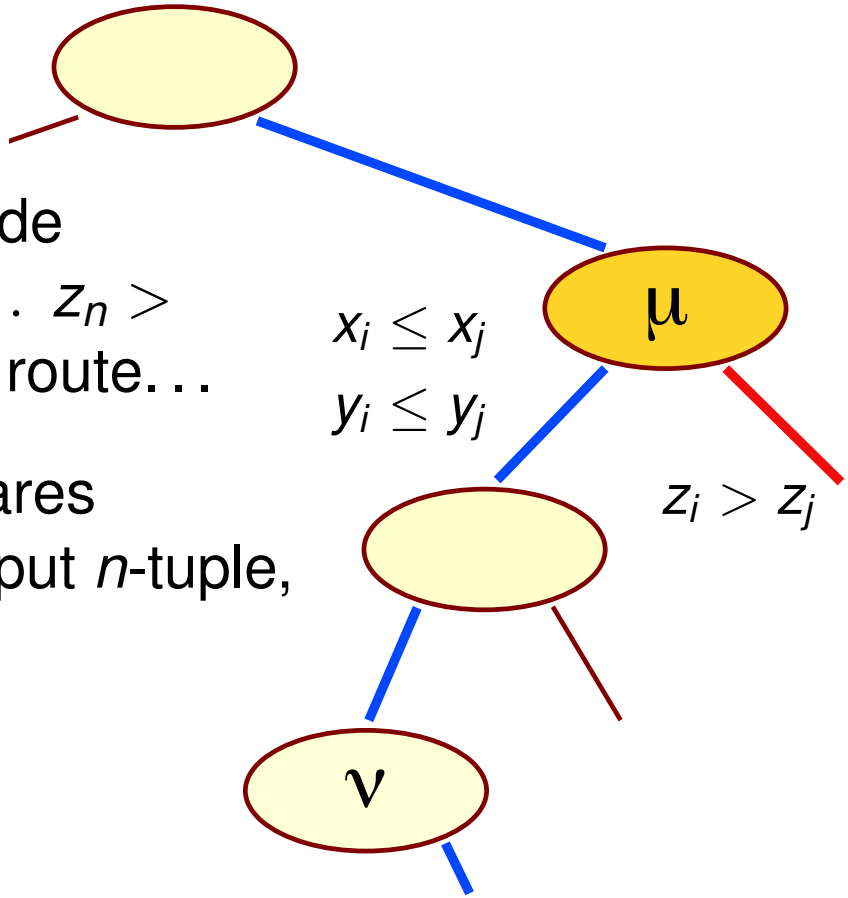


"Uniqueness": Contradiction

If not so, let μ be ν's parent node where the path for $< z_1, z_2, \ldots z_n >$ is supposed to take a different route...

At node μ the algorithm compares the i-th and j-th items of the input n-tuple,

but since $z_i = \alpha x_i + (1 - \alpha) y_i$ and $z_j = \alpha x_j + (1 - \alpha) y_j$, comparing z_i and z_j cannot result in a different decision

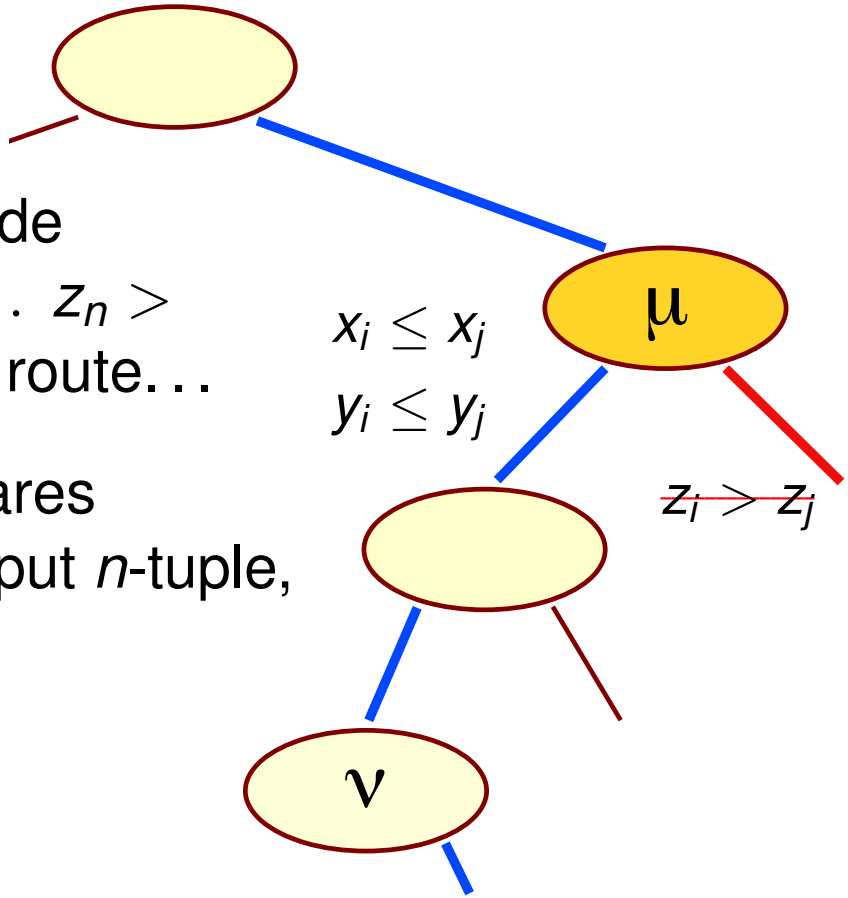


"Uniqueness": Contradiction

If not so, let μ be ν's parent node where the path for $< z_1, z_2, \ldots z_n >$ is supposed to take a different route...

At node μ the algorithm compares the i-th and j-th items of the input n-tuple,

but since $z_i = \alpha x_i + (1 - \alpha) y_i$ and $z_j = \alpha x_j + (1 - \alpha) y_j$, comparing z_i and z_j cannot result in a different decision



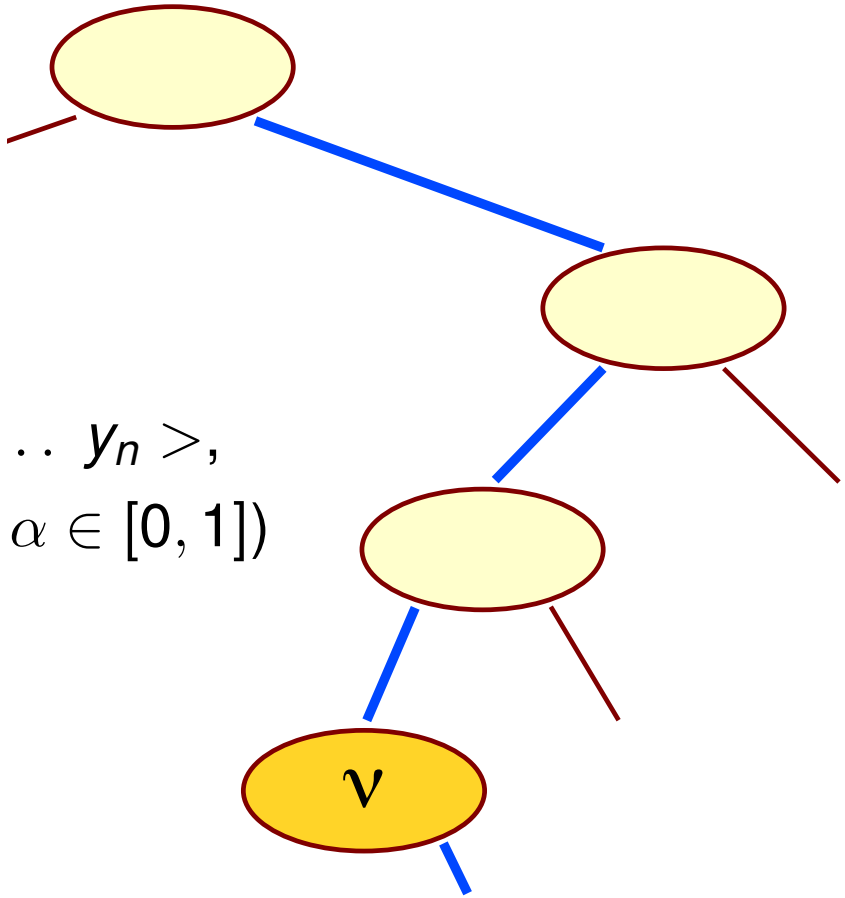
"Uniqueness": Implication

Thus, the input n-tuples

$< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$,

or $< z_1, z_2, \ldots z_n >$ (for any $\alpha \in [0, 1]$)

all lead to node ν ...



"Uniqueness": At node ν ...

- $< x_1, x_2, \ldots x_n >, < y_1, y_2, \ldots y_n >$
  are permutations of a same n-tuple of *distinct* items
- Hence, we can find i, j such that $x_i < x_j$ and $y_i > y_j$[1]
- By choosing $\alpha = \frac{y_i - y_j}{x_j - x_i + y_i - y_j}$ we have $z_i = z_j$
  (notice that the denominator is strictly positive)

[1] Let x_i be the smallest item changing its position, with destination j, then $x_i = y_j < x_j, y_i$.

"Uniqueness": At node ν ...

- $< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$
  are permutations of a same n-tuple of *distinct* items
- Hence, we can find i, j such that $x_i < x_j$ and $y_i > y_j$[1]
- By choosing $\alpha = \frac{y_i - y_j}{x_j - x_i + y_i - y_j}$ we have $z_i = z_j$
  (notice that the denominator is strictly positive)

[1] Let x_i be the smallest item changing its position, with destination j, then $x_i = y_j < x_j, y_i$.

"Uniqueness": At node ν ...

- $< x_1, x_2, \ldots x_n >, < y_1, y_2, \ldots y_n >$
  are permutations of a same *n*-tuple of *distinct* items
- Hence, we can find i, j such that $x_i < x_j$ and $y_i > y_j$[1]
- By choosing $\alpha = \frac{y_i - y_j}{x_j - x_i + y_i - y_j}$ we have $z_i = z_j$
  (notice that the denominator is strictly positive)

[1] Let x_i be the smallest item changing its position, with destination j, then $x_i = y_j < x_j, y_i$.

"Uniqueness": At node ν ...

- $< x_1, x_2, \ldots x_n >, < y_1, y_2, \ldots y_n >$
  are permutations of a same n-tuple of *distinct* items
- Hence, we can find i, j such that $x_i < x_j$ and $y_i > y_j$[1]
- By choosing $\alpha = \frac{y_i - y_j}{x_j - x_i + y_i - y_j}$ we have $z_i = z_j$
  (notice that the denominator is strictly positive)

[1] Let x_i be the smallest item changing its position, with destination j, then $x_i = y_j < x_j, y_i$.

"Uniqueness": At node ν ...

- $< x_1, x_2, \ldots x_n >, < y_1, y_2, \ldots y_n >$
  are permutations of a same n-tuple of *distinct* items
- Hence, we can find i, j such that $x_i < x_j$ and $y_i > y_j$[1]
- By choosing $\alpha = \frac{y_i - y_j}{x_j - x_i + y_i - y_j}$ we have $z_i = z_j$
  (notice that the denominator is strictly positive)

[1] Let x_i be the smallest item changing its position, with destination j, then $x_i = y_j < x_j, y_i$.

"Uniqueness": At node ν ...

- Starting from input:

  $< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$ (uniqueness holds)

  as well as $< z_1, z_2, \ldots z_n >$ (uniqueness does not hold) ...

- the algorithm always reaches node ν

- ν cannot be a leaf of the decision tree!

- $\Rightarrow$ there need to be as many decision-tree leaves as there are permutations of n-tuples

"Uniqueness": At node ν ...

- Starting from input:

  $< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$ (uniqueness holds)

  as well as $< z_1, z_2, \ldots z_n >$ (uniqueness does not hold) ...

- the algorithm always reaches node ν

- ν cannot be a leaf of the decision tree!

- $\Rightarrow$ there need to be as many decision-tree leaves as there are permutations of n-tuples

"Uniqueness": At node ν ...

- Starting from input:

  $< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$ (uniqueness holds)

  as well as $< z_1, z_2, \ldots z_n >$ (uniqueness does not hold) ...

- the algorithm always reaches node ν

- ν cannot be a leaf of the decision tree!

- $\Rightarrow$ there need to be as many decision-tree leaves as there are permutations of n-tuples

"Uniqueness": At node ν ...

- Starting from input:

  $< x_1, x_2, \ldots x_n >$, $< y_1, y_2, \ldots y_n >$ (uniqueness holds)

  as well as $< z_1, z_2, \ldots z_n >$ (uniqueness does not hold) ...

- the algorithm always reaches node ν

- ν cannot be a leaf of the decision tree!

- $\Rightarrow$ there need to be as many decision-tree leaves as there are permutations of n-tuples

Correctness

- If an intersection is reported
  then, of course, a pair of segments do intersect
- May intersections remain undetected?
- Consider the leftmost of all intersections,
  involving line segments u and v (u met before v)

Correctness

- If an intersection is reported
  then, of course, a pair of segments do intersect
- May intersections remain undetected?
- Consider the leftmost of all intersections,
  involving line segments u and v (u met before v)

Correctness

- If an intersection is reported
  then, of course, a pair of segments do intersect
- May intersections remain undetected?
- Consider the leftmost of all intersections,
  involving line segments u and v (u met before v)

Correctness

- If no other intersection is found in the meanwhile...
- Either v is a neighbor of u in S_k immediately after being added (intersection detected by processing *ENTER* event)
- Or u and v become neighbors after removing some other segment (intersection detected by processing *EXIT* event)

Correctness

- If no other intersection is found in the meanwhile...
- Either v is a neighbor of u in S_k immediately after being added (intersection detected by processing *ENTER* event)
- Or u and v become neighbors after removing some other segment (intersection detected by processing *EXIT* event)

Correctness

- If no other intersection is found in the meanwhile...
- Either v is a neighbor of u in S_k immediately after being added (intersection detected by processing *ENTER* event)
- Or u and v become neighbors after removing some other segment (intersection detected by processing *EXIT* event)

Degeneracies...

- What about segments sharing endpoints?
- And what about vertical segments?
- Or vertical *collinear* segments?
- Think of sorting in *lexicographic* order!

Degeneracies...

- What about segments sharing endpoints?
- And what about vertical segments?
- Or vertical *collinear* segments?
- Think of sorting in *lexicographic* order!

Degeneracies...

- What about segments sharing endpoints?
- And what about vertical segments?
- Or vertical *collinear* segments?
- Think of sorting in *lexicographic* order!

Degeneracies...

- What about segments sharing endpoints?
- And what about vertical segments?
- Or vertical *collinear* segments?
- Think of sorting in *lexicographic* order!

One more step forward

- Shamos & Hoey's algorithm reports a "witness" intersection
- How to proceed in order to find the other intersections?
- Key observation: critical changes at intersection points!
- New plane-sweep event: *SWAP* event
  (the order in S_k changes)

One more step forward

- Shamos & Hoey's algorithm reports a "witness" intersection
- How to proceed in order to find the other intersections?
- Key observation: critical changes at intersection points!
- New plane-sweep event: *SWAP* event
  (the order in S_k changes)

One more step forward

- Shamos & Hoey's algorithm reports a "witness" intersection
- How to proceed in order to find the other intersections?
- Key observation: critical changes at intersection points!
- New plane-sweep event: *SWAP* event (the order in S_k changes)

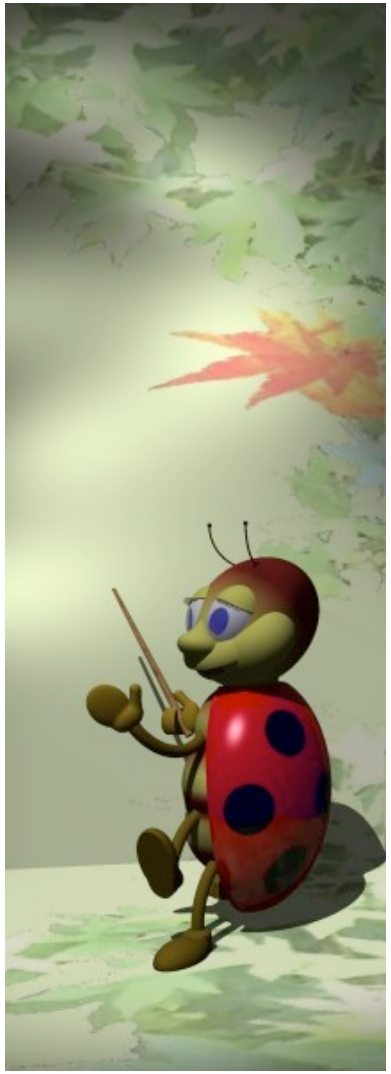
One more step forward

- Shamos & Hoey's algorithm reports a "witness" intersection
- How to proceed in order to find the other intersections?
- Key observation: critical changes at intersection points!
- New plane-sweep event: *SWAP* event
  (the order in S_k changes)

One more step forward

- Shamos & Hoey's algorithm reports a "witness" intersection
- How to proceed in order to find the other intersections?
- Key observation: critical changes at intersection points!
- New plane-sweep event: *SWAP* event
  (the order in S_k changes)

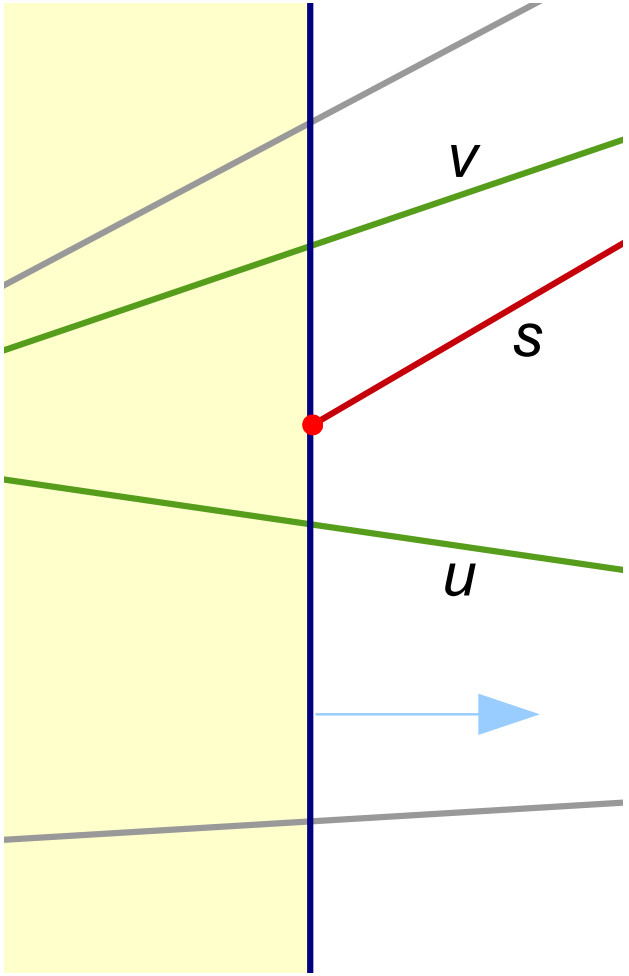
Outline



Bentley & Ottmann (1979)

ENTER event:

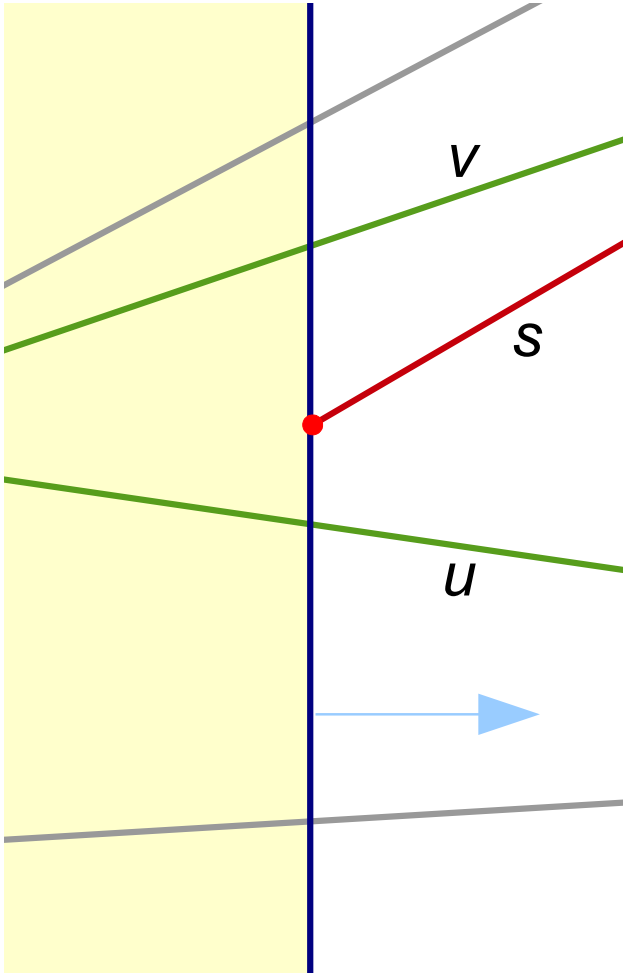
- leftmost endpoint of segment s
- falling between u and v
- new adjacencies: u–s, s–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

ENTER event:

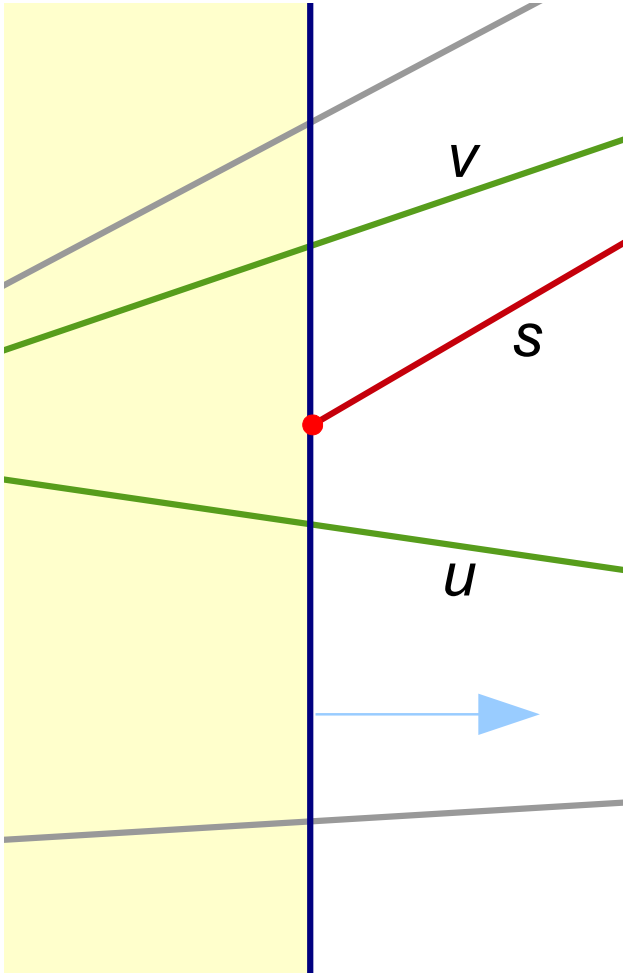
- leftmost endpoint of segment s
- falling between u and v
- new adjacencies: u–s, s–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

ENTER event:

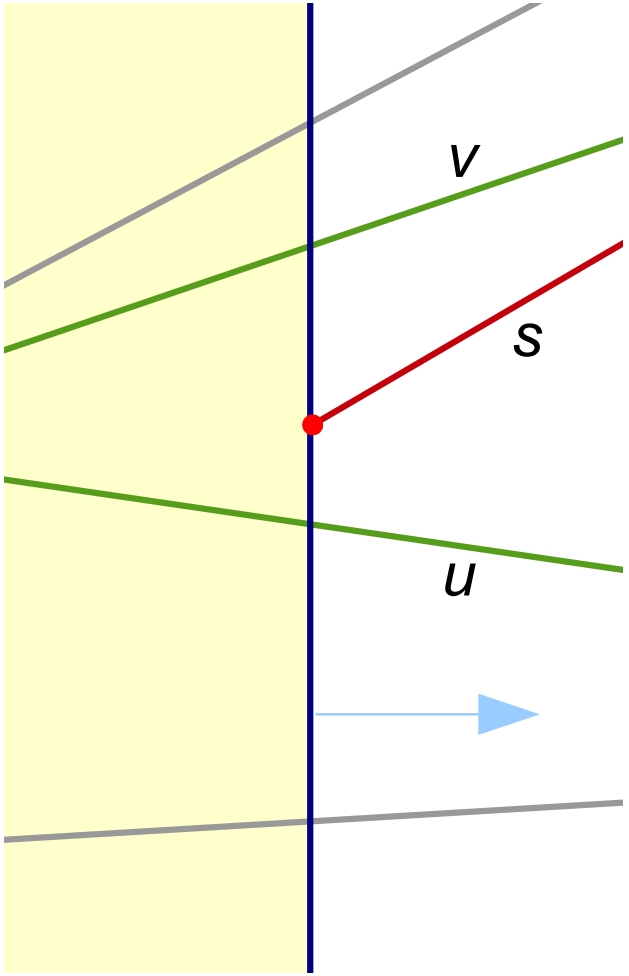
- leftmost endpoint of segment s
- falling between u and v
- new adjacencies: u–s, s–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

ENTER event:

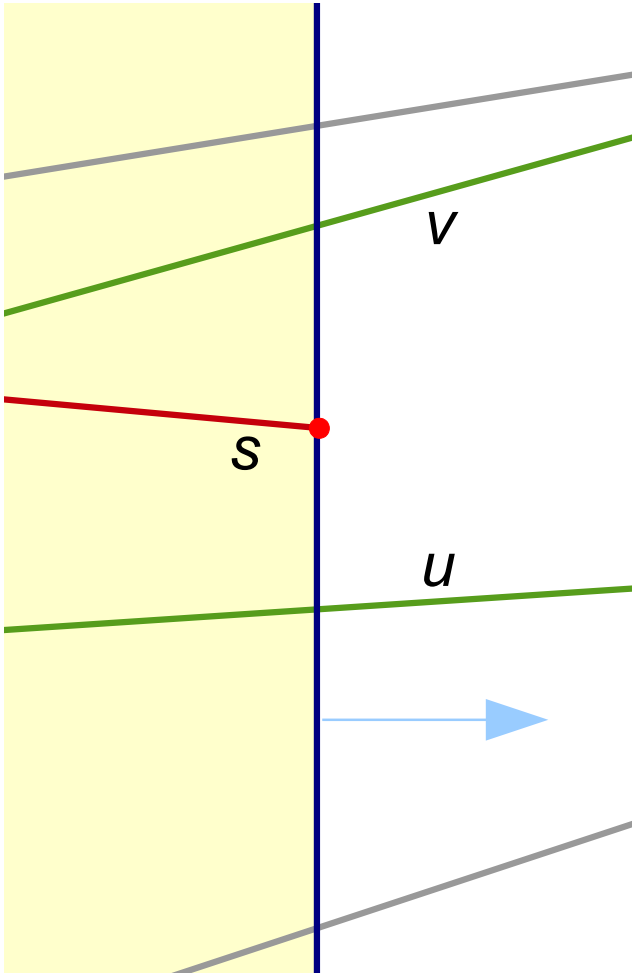
- leftmost endpoint of segment s
- falling between u and v
- new adjacencies: u–s, s–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

EXIT event:

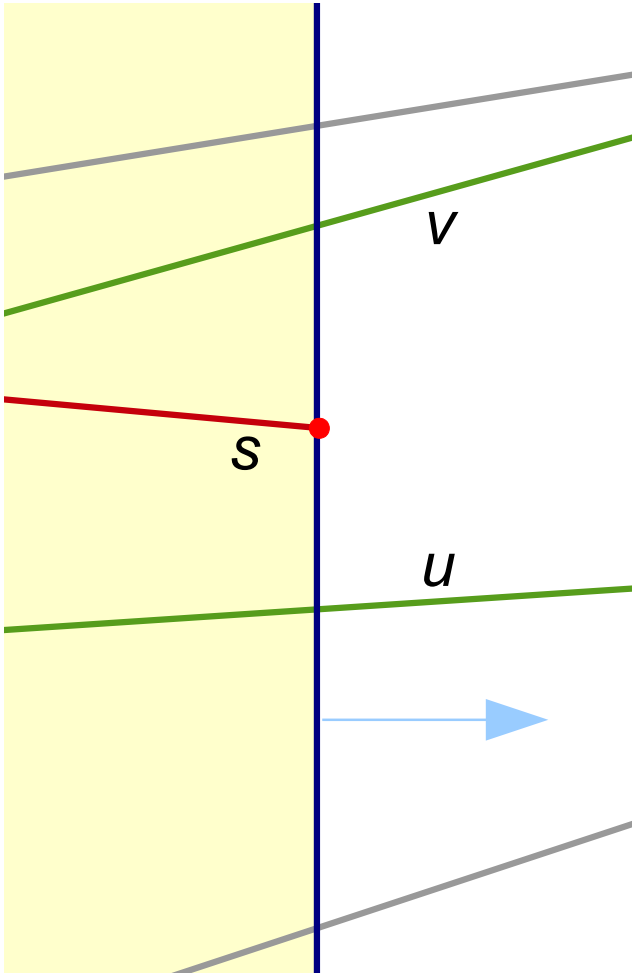
- rightmost endpoint of segment *s*
- falling between *u* and *v*
- new adjacency: *u*–*v*
- to be checked for intersection



Bentley & Ottmann (1979)

EXIT event:

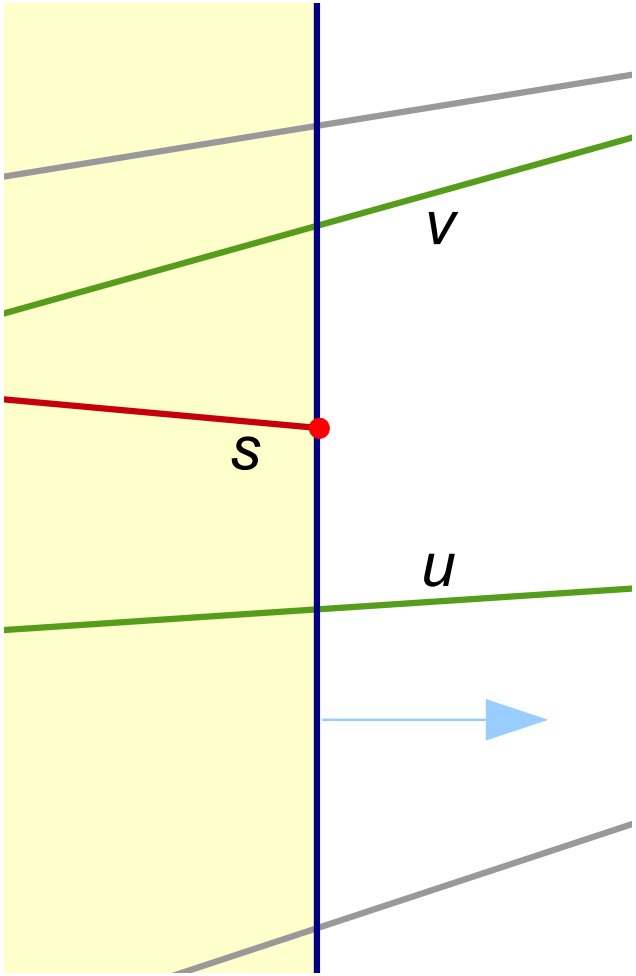
- rightmost endpoint of segment s
- falling between u and v
- new adjacency: u–v
- to be checked for intersection



Bentley & Ottmann (1979)

EXIT event:

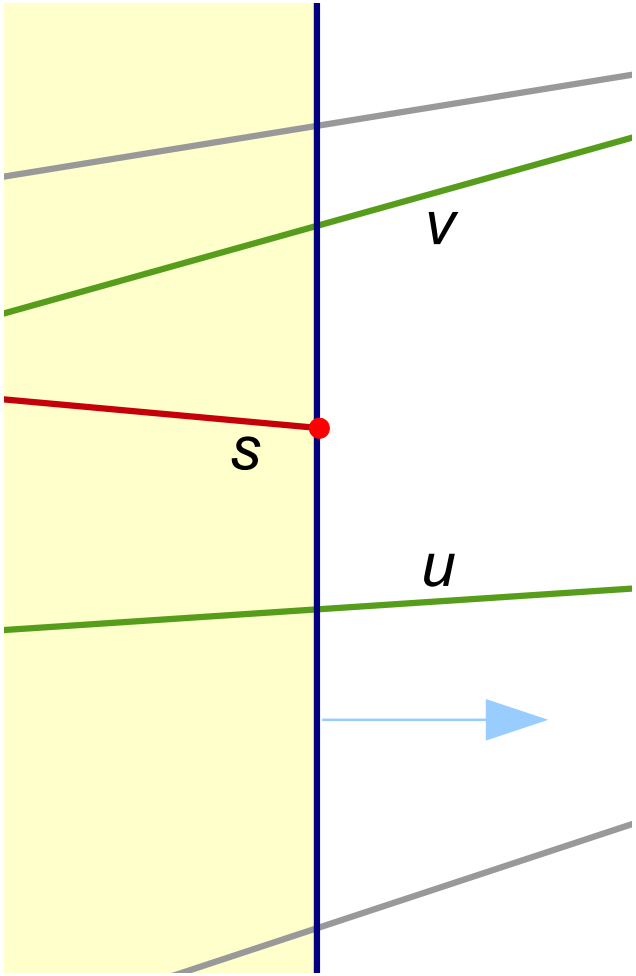
- rightmost endpoint of segment s
- falling between u and v
- new adjacency: u–v
- to be checked for intersection



Bentley & Ottmann (1979)

EXIT event:

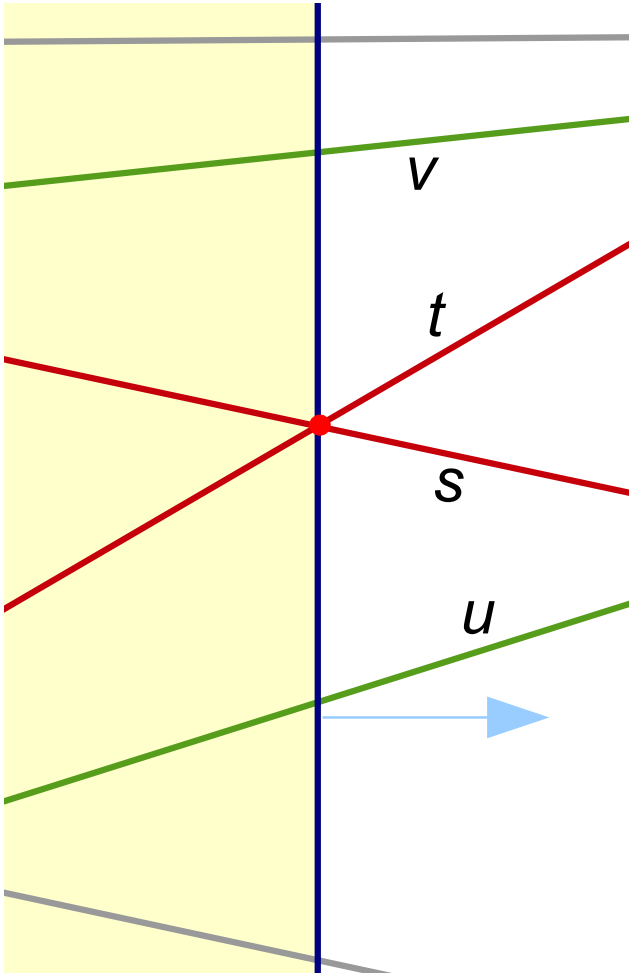
- rightmost endpoint of segment s
- falling between u and v
- new adjacency: u–v
- to be checked for intersection



Bentley & Ottmann (1979)

SWAP event:

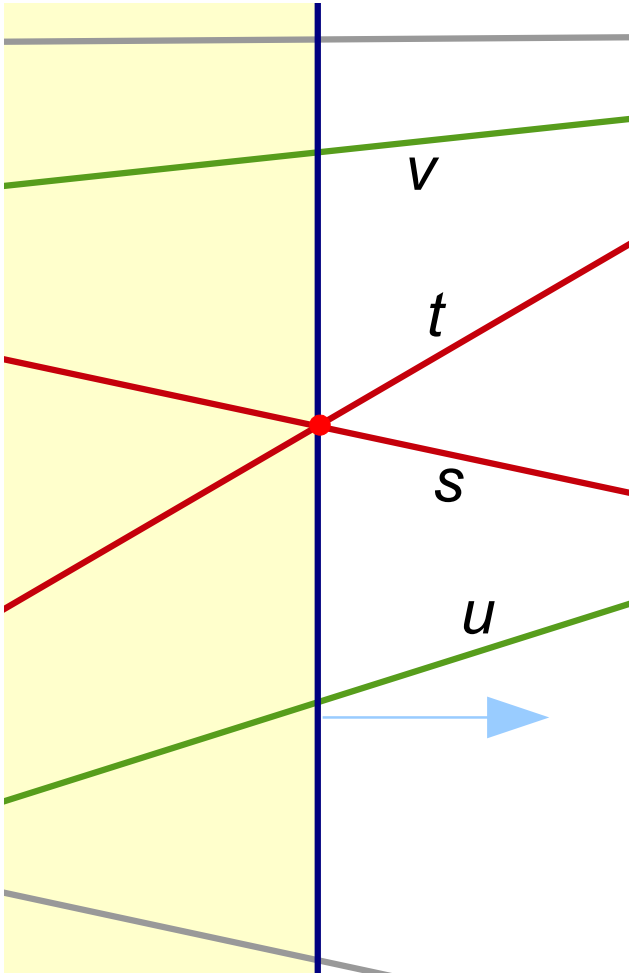
- crossing point of segments s and t
- falling between u and v
- new adjacencies: u–s, t–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

SWAP event:

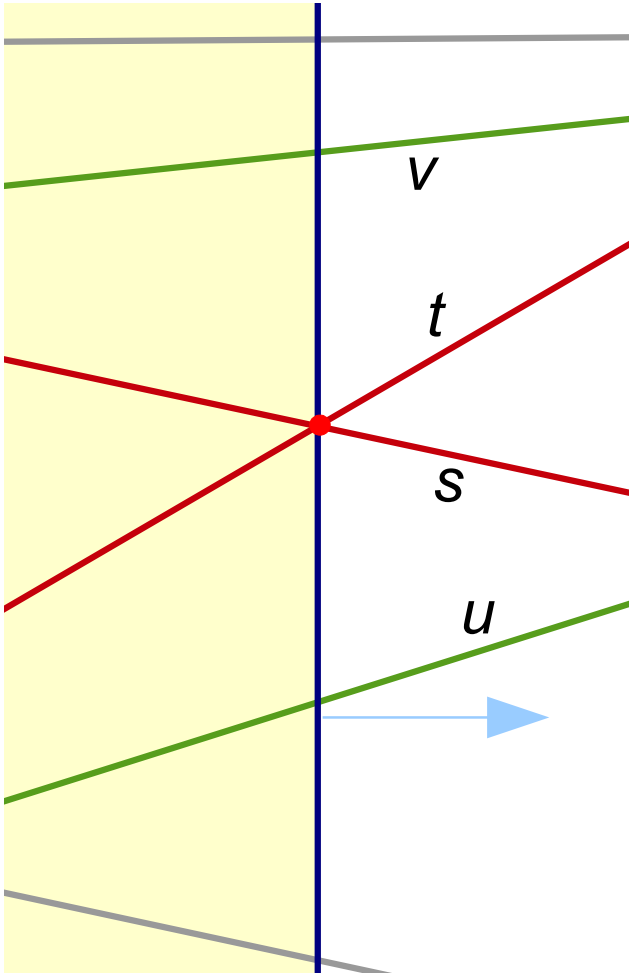
- crossing point of segments s and t
- falling between u and v
- new adjacencies: u–s, t–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

SWAP event:

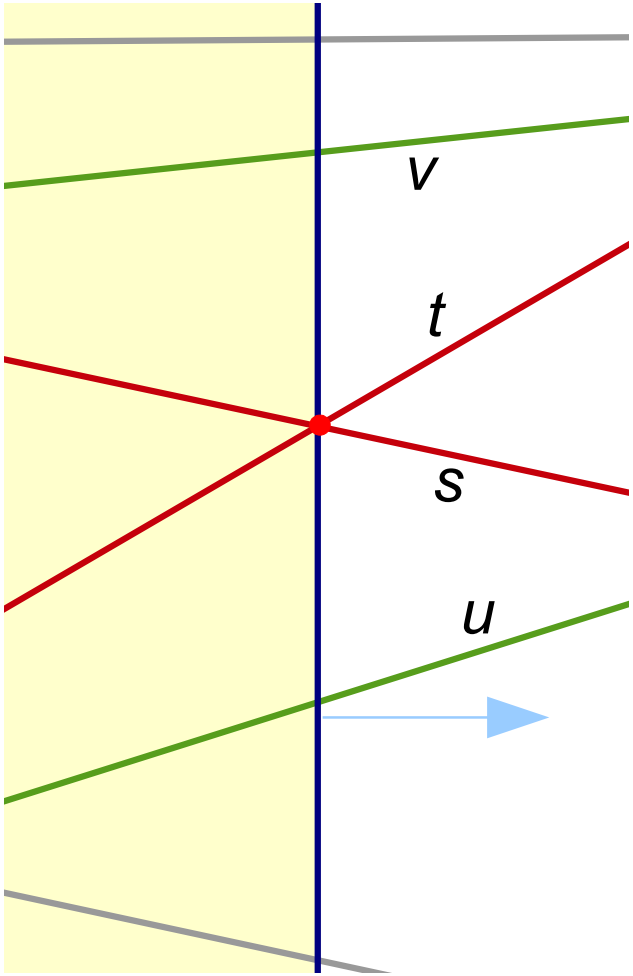
- crossing point of segments s and t
- falling between u and v
- new adjacencies: u–s, t–v
- both are to be checked for intersection



Bentley & Ottmann (1979)

SWAP event:

- crossing point of segments s and t
- falling between u and v
- new adjacencies: u–s, t–v
- both are to be checked for intersection



Provisional assumptions

- No vertical segments
- Segments do not overlap
- No three segments meet at a common point
- Endpoints are not shared between segments
- No (other) intersections with endpoints

Provisional assumptions

- No vertical segments
- Segments do not overlap
- No three segments meet at a common point
- Endpoints are not shared between segments
- No (other) intersections with endpoints

Provisional assumptions

- No vertical segments
- Segments do not overlap
- No three segments meet at a common point
- Endpoints are not shared between segments
- No (other) intersections with endpoints

Provisional assumptions

- No vertical segments
- Segments do not overlap
- No three segments meet at a common point
- Endpoints are not shared between segments
- No (other) intersections with endpoints

Provisional assumptions

- No vertical segments
- Segments do not overlap
- No three segments meet at a common point
- Endpoints are not shared between segments
- No (other) intersections with endpoints

Further remarks

- *Invariant*: all intersection points on the left side of the sweep line have been computed
- Intersection events may be found more than once
- Only intersection events on the right side of the sweep line need to be considered
- *Event identity* needs however to be tested
- This will also allow to deal with shared endpoints, three or more segments meeting in a common point. . .

Further remarks

- *Invariant*: all intersection points on the left side of the sweep line have been computed
- Intersection events may be found more than once
- Only intersection events on the right side of the sweep line need to be considered
- *Event identity* needs however to be tested
- This will also allow to deal with shared endpoints, three or more segments meeting in a common point. . .

Further remarks

- *Invariant*: all intersection points on the left side of the sweep line have been computed
- Intersection events may be found more than once
- Only intersection events on the right side of the sweep line need to be considered
- *Event identity* needs however to be tested
- This will also allow to deal with shared endpoints, three or more segments meeting in a common point. . .

Further remarks

- *Invariant*: all intersection points on the left side of the sweep line have been computed
- Intersection events may be found more than once
- Only intersection events on the right side of the sweep line need to be considered
- *Event identity* needs however to be tested
- This will also allow to deal with shared endpoints, three or more segments meeting in a common point. . .

Further remarks

- *Invariant*: all intersection points on the left side of the sweep line have been computed
- Intersection events may be found more than once
- Only intersection events on the right side of the sweep line need to be considered
- *Event identity* needs however to be tested
- This will also allow to deal with shared endpoints, three or more segments meeting in a common point. . .

Data structure refinement

- *SWAP* events are interleaved with *ENTER*/*EXIT* events...
- ...but the corresponding points are not known in advance
- Refinement: dynamic event sorting via *priority queue*
- Overall cost of operations on the event queue:
  $O(\ |E| \log n\ )$, where E is the set of all processed events

Data structure refinement

- *SWAP* events are interleaved with *ENTER*/*EXIT* events...
- ...but the corresponding points are not known in advance
- Refinement: dynamic event sorting via *priority queue*
- Overall cost of operations on the event queue:
  $O(\ |E| \log n\ )$, where E is the set of all processed events

Data structure refinement

- *SWAP* events are interleaved with *ENTER*/*EXIT* events...
- ...but the corresponding points are not known in advance
- Refinement: dynamic event sorting via *priority queue*
- Overall cost of operations on the event queue:
  $O(\ |E| \log n\ )$, where E is the set of all processed events

Data structure refinement

- *SWAP* events are interleaved with *ENTER*/*EXIT* events...
- ...but the corresponding points are not known in advance
- Refinement: dynamic event sorting via *priority queue*
- Overall cost of operations on the event queue:
  $O(\ |E| \log n\ )$, where E is the set of all processed events

Correctness of the approach

- Observation: the structure connected to the sweep line is representative of the geometry within a vertical strip between two consecutive event points
- Since segment intersections are event points, segments crossing the strip can be sorted from the bottom upwards
- In the strip just to the left of the intersection between u and v, segments u and v must be adjacent

Correctness of the approach

- Observation: the structure connected to the sweep line is representative of the geometry within a vertical strip between two consecutive event points
- Since segment intersections are event points, segments crossing the strip can be sorted from the bottom upwards
- In the strip just to the left of the intersection between u and v, segments u and v must be adjacent

Correctness of the approach

- Observation: the structure connected to the sweep line is representative of the geometry within a vertical strip between two consecutive event points
- Since segment intersections are event points, segments crossing the strip can be sorted from the bottom upwards
- In the strip just to the left of the intersection between u and v, segments u and v must be adjacent

Correctness of the approach

- Then, there must be a former event where u and v become adjacent for the first time
- This holds for any pair of intersecting segments. . .
- Hence, all intersections can be detected and reported
- Geometric interpretation of lexicographic order relative to the sweep line

Correctness of the approach

- Then, there must be a former event where u and v become adjacent for the first time
- This holds for any pair of intersecting segments...
- Hence, all intersections can be detected and reported
- Geometric interpretation of lexicographic order relative to the sweep line

Correctness of the approach

- Then, there must be a former event where u and v become adjacent for the first time
- This holds for any pair of intersecting segments...
- Hence, all intersections can be detected and reported
- Geometric interpretation of lexicographic order relative to the sweep line

Correctness of the approach

- Then, there must be a former event where u and v become adjacent for the first time
- This holds for any pair of intersecting segments...
- Hence, all intersections can be detected and reported
- Geometric interpretation of lexicographic order relative to the sweep line

Putting things together

- *Event queue*:
  - either priority queue or balanced search tree
  - insert / fetch in $O(\log |E|) = O(\log n^2) = O(\log n)$
- *Sweep line structure*:
  - balanced search tree
  - insert / find / remove / previous / next in $O(\log n)$

Putting things together

- *Event queue*:
  - either priority queue or balanced search tree
  - insert / fetch in $O(\log |E|) = O(\log n^2) = O(\log n)$
- *Sweep line structure*:
  - balanced search tree
  - insert / find / remove / previous / next in $O(\log n)$

Putting things together

- *Event queue*:
  - either priority queue or balanced search tree
  - insert / fetch in $O(\log |E|) = O(\log n^2) = O(\log n)$
- *Sweep line structure*:
  - balanced search tree
  - insert / find / remove / previous / next in $O(\log n)$

Putting things together

- *Event queue*:
  - either priority queue or balanced search tree
  - insert / fetch in $O(\log |E|) = O(\log n^2) = O(\log n)$
- *Sweep line structure*:
  - balanced search tree
  - insert / find / remove / previous / next in $O(\log n)$

Main steps

- Event queue initialization (all *ENTER*/*EXIT* events): $O(\ n \log n\ )$
- For each event: $O(\ n + k\ )$ iterations for k intersections
  - possible repeats of SWAP events can be ascribed to one of the $O(\ n + k\ )$ "previous" EXIT/SWAP events
- Process event: $O(\ \log n\ )$
  - event queue: one *fetch* and possibly one or two *insert*
  - sweep line structure: no more than two *remove*, one *find*, two *insert*, one *previous*, one *next*
  - two remove + two (re)insert for *SWAP* events

Main steps

- Event queue initialization (all *ENTER*/*EXIT* events): $O(\ n \log n\ )$
- For each event: $O(\ n + k\ )$ iterations for k intersections
  - possible repeats of SWAP events can be ascribed to one of the $O(\ n + k\ )$ "previous" EXIT/SWAP events
- Process event: $O(\ \log n\ )$
  - event queue: one *fetch* and possibly one or two *insert*
  - sweep line structure: no more than two *remove*, one *find*, two *insert*, one *previous*, one *next*
  - two remove + two (re)insert for *SWAP* events

Main steps

- Event queue initialization (all *ENTER*/*EXIT* events): $O(\ n \log n\ )$
- For each event: $O(\ n + k\ )$ iterations for k intersections
  - possible repeats of SWAP events can be ascribed to one of the $O(\ n + k\ )$ "previous" EXIT/SWAP events
- Process event: $O(\ \log n\ )$
  - event queue: one *fetch* and possibly one or two *insert*
  - sweep line structure: no more than two *remove*, one *find*, two *insert*, one *previous*, one *next*
  - two remove + two (re)insert for *SWAP* events

Main steps

- Event queue initialization (all *ENTER*/*EXIT* events): $O(\ n \log n\ )$
- For each event: $O(\ n + k\ )$ iterations for k intersections
  - possible repeats of SWAP events can be ascribed to one of the $O(\ n + k\ )$ "previous" EXIT/SWAP events
- Process event: $O(\ \log n\ )$
  - event queue: one *fetch* and possibly one or two *insert*
  - sweep line structure: no more than two *remove*, one *find*, two *insert*, one *previous*, one *next*
  - two remove + two (re)insert for *SWAP* events

Main steps

- Event queue initialization (all *ENTER*/*EXIT* events): $O(\ n \log n\ )$
- For each event: $O(\ n + k\ )$ iterations for k intersections
  - possible repeats of SWAP events can be ascribed to one of the $O(\ n + k\ )$ "previous" EXIT/SWAP events
- Process event: $O(\ \log n\ )$
  - event queue: one *fetch* and possibly one or two *insert*
  - sweep line structure: no more than two *remove*, one *find*, two *insert*, one *previous*, one *next*
  - two remove + two (re)insert for *SWAP* events

Main steps

- Event queue initialization (all *ENTER*/*EXIT* events): $O(\ n \log n\ )$
- For each event: $O(\ n + k\ )$ iterations for k intersections
  - possible repeats of SWAP events can be ascribed to one of the $O(\ n + k\ )$ "previous" EXIT/SWAP events
- Process event: $O(\ \log n\ )$
  - event queue: one *fetch* and possibly one or two *insert*
  - sweep line structure: no more than two *remove*, one *find*, two *insert*, one *previous*, one *next*
  - two remove + two (re)insert for *SWAP* events

Computational costs

- Running costs: $O((n+k) \log n)$
- Storage costs: how to lower the storage required for the event queue from $O(|E|)$ to $O(n)$?
- Keep only intersection events relative to adjacent segments in the sweep line structure: $O(n)$
- . . . if removed they will be re-inserted before processing the intersection event!
- event queue $\rightarrow$ balanced search tree

Computational costs

- Running costs: $O((n+k) \log n)$
- Storage costs: how to lower the storage required for the event queue from $O(|E|)$ to $O(n)$?
- Keep only intersection events relative to adjacent segments in the sweep line structure: $O(n)$
- . . . if removed they will be re-inserted before processing the intersection event!
- event queue $\rightarrow$ balanced search tree

Computational costs

- Running costs: $O(\ (n+k)\log n\ )$
- Storage costs: how to lower the storage required for the event queue from $O(\ |E|\ )$ to $O(\ n\ )$?
- Keep only intersection events relative to adjacent segments in the sweep line structure: $O(\ n\ )$
- . . . if removed they will be re-inserted before processing the intersection event!
- event queue $\rightarrow$ balanced search tree

Computational costs

- Running costs: $O(\ (n+k)\log n\ )$
- Storage costs: how to lower the storage required for the event queue from $O(\ |E|\ )$ to $O(\ n\ )$?
- Keep only intersection events relative to adjacent segments in the sweep line structure: $O(\ n\ )$
- . . . if removed they will be re-inserted before processing the intersection event!
- event queue $\rightarrow$ balanced search tree

Computational costs

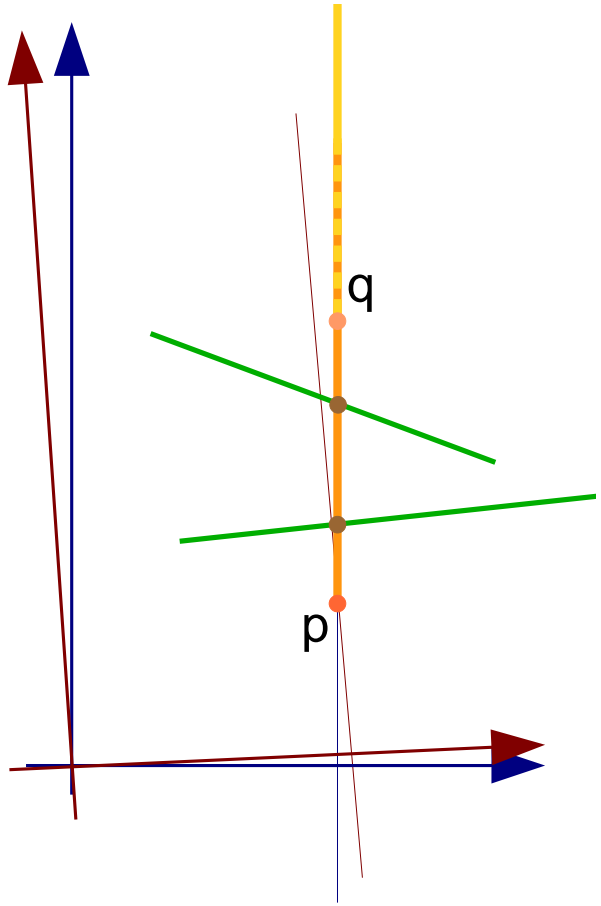
- Running costs: $O(\ (n+k)\log n\ )$
- Storage costs: how to lower the storage required for the event queue from $O(\ |E|\ )$ to $O(\ n\ )$?
- Keep only intersection events relative to adjacent segments in the sweep line structure: $O(\ n\ )$
- . . . if removed they will be re-inserted before processing the intersection event!
- event queue $\rightarrow$ balanced search tree

Degeneracies...

- What about vertical segments?
- Again, think of a *lexicographic* order!
- Points shared by more than two segments?
- (overlapping segments are not contemplated)

Degeneracies...

- What about vertical segments?
- Again, think of a *lexicographic* order!
- Points shared by more than two segm
- (overlapping segments are not conten



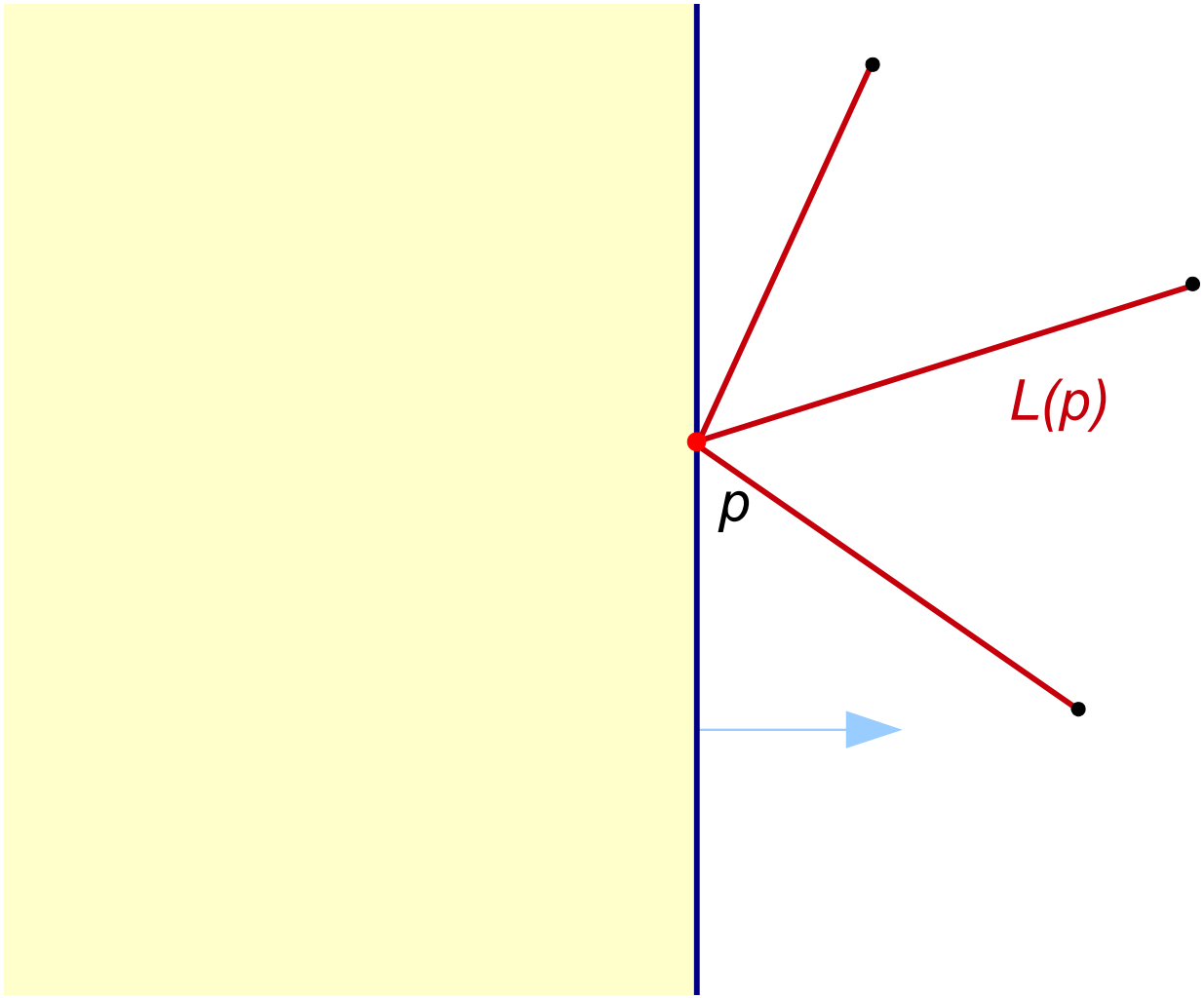
Degeneracies...

- What about vertical segments?
- Again, think of a *lexicographic* order!
- Points shared by more than two segments?
- (overlapping segments are not contemplated)

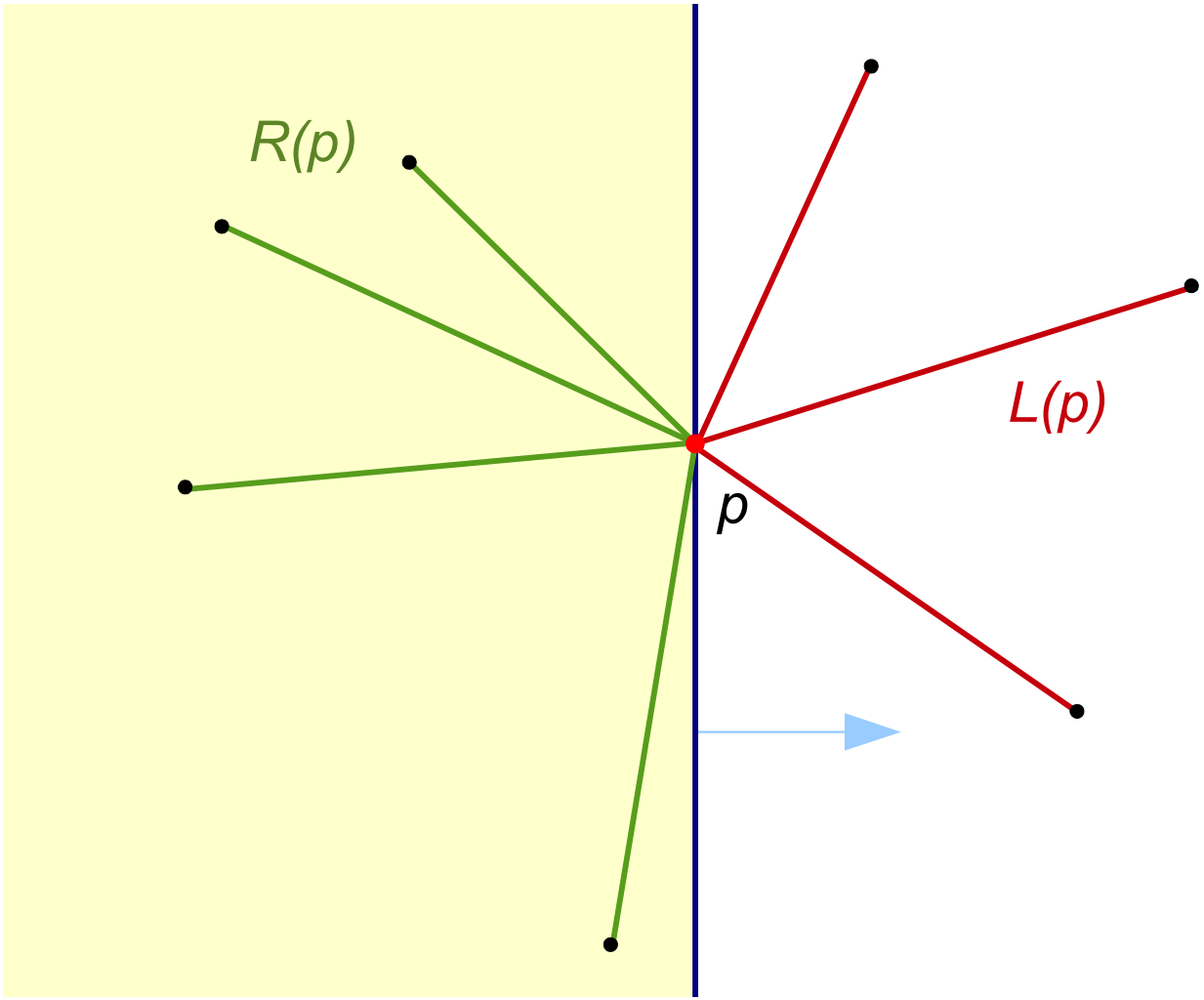
Degeneracies...

- What about vertical segments?
- Again, think of a *lexicographic* order!
- Points shared by more than two segments?
- (overlapping segments are not contemplated)

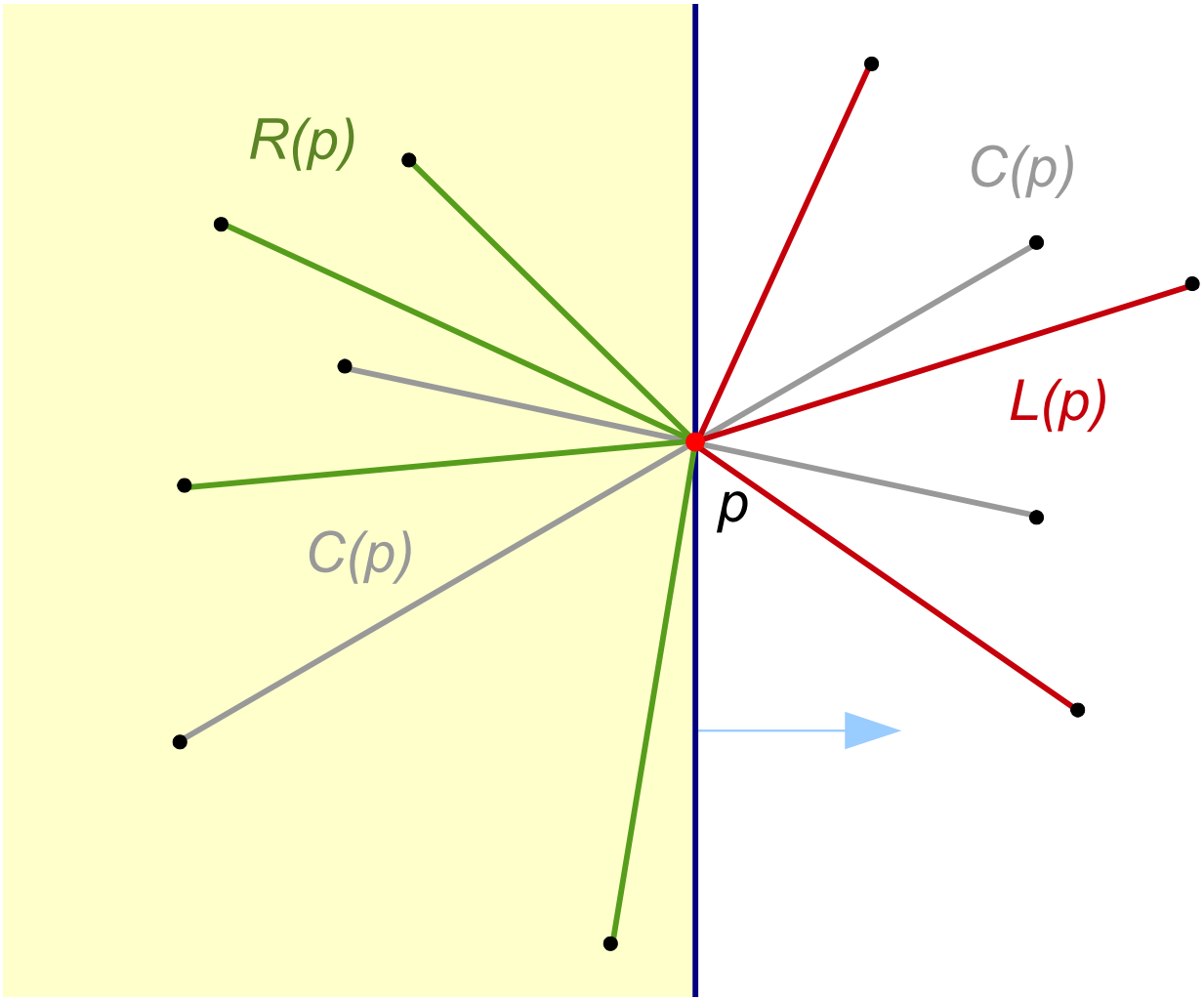
Several segments meet at a common (end)point



Several segments meet at a common (end)point



Several segments meet at a common (end)point



Several segments meet at a common (end)point

- *ENTER*, *EXIT* and *SWAP* events at p are adjacent in the event queue (search tree)
- They may be merged into a *HYBRID* event to be updated dynamically
- When event at p is processed, segments in $R(p) \cup C(p)$ are adjacent in the sweep line structure

Several segments meet at a common (end)point

- *ENTER*, *EXIT* and *SWAP* events at p are adjacent in the event queue (search tree)
- They may be merged into a *HYBRID* event to be updated dynamically
- When event at p is processed, segments in $R(p) \cup C(p)$ are adjacent in the sweep line structure

Several segments meet at a common (end)point

- *ENTER*, *EXIT* and *SWAP* events at p are adjacent in the event queue (search tree)
- They may be merged into a *HYBRID* event to be updated dynamically
- When event at p is processed, segments in $R(p) \cup C(p)$ are adjacent in the sweep line structure

Processing *HYBRID* events

- If $|L(p) \cup R(p) \cup C(p)| > 1$ then report intersection point p
- Remove segments in $R(p) \cup C(p)$ from the sweep line structure
- Insert segments in $L(p) \cup C(p)$ into the sweep line structure
- Order of segments in $C(p)$ is thus reversed

Processing *HYBRID* events

- If $|L(p) \cup R(p) \cup C(p)| > 1$ then report intersection point p
- Remove segments in $R(p) \cup C(p)$ from the sweep line structure
- Insert segments in $L(p) \cup C(p)$ into the sweep line structure
- Order of segments in $C(p)$ is thus reversed

Processing *HYBRID* events

- If $|L(p) \cup R(p) \cup C(p)| > 1$ then report intersection point p
- Remove segments in $R(p) \cup C(p)$ from the sweep line structure
- Insert segments in $L(p) \cup C(p)$ into the sweep line structure
- Order of segments in $C(p)$ is thus reversed

Processing *HYBRID* events

- If $|L(p) \cup R(p) \cup C(p)| > 1$ then report intersection point p
- Remove segments in $R(p) \cup C(p)$ from the sweep line structure
- Insert segments in $L(p) \cup C(p)$ into the sweep line structure
- Order of segments in $C(p)$ is thus reversed

Processing *HYBRID* events

- If $|L(p) \cup C(p)| = 0$ then check for intersection between the two segments just below and just above those in $R(p)$
- Otherwise check for intersections relative to the two new adjacencies in the sweep line structure
- Either insert a new event or find and possibly update an existing event in the event queue accordingly

Processing *HYBRID* events

- If $|L(p) \cup C(p)| = 0$ then check for intersection between the two segments just below and just above those in $R(p)$
- Otherwise check for intersections relative to the two new adjacencies in the sweep line structure
- Either insert a new event or find and possibly update an existing event in the event queue accordingly

Processing *HYBRID* events

- If $|L(p) \cup C(p)| = 0$ then check for intersection between the two segments just below and just above those in $R(p)$
- Otherwise check for intersections relative to the two new adjacencies in the sweep line structure
- Either insert a new event or find and possibly update an existing event in the event queue accordingly

Stricter cost bound

- From $O(\,(n+k)\log n\,)$ ...
- ... to $O(\,(n+i)\log n\,)$
- where i = number of distinct intersection points
- Analysis based on *Euler's formula*
- Observe that $i < k$ for points shared by several segments!

Stricter cost bound

- From $O((n+k)\log n)$ ...
- ... to $O((n+i)\log n)$
- where i = number of distinct intersection points
- Analysis based on *Euler's formula*
- Observe that $i < k$ for points shared by several segments!

Stricter cost bound

- From $O((n+k)\log n)$ ...
- ... to $O((n+i)\log n)$
- where i = number of distinct intersection points
- Analysis based on *Euler's formula*
- Observe that $i < k$ for points shared by several segments!

Stricter cost bound

- From $O(\ (n+k)\log n\ )$ ...
- ... to $O(\ (n+i)\log n\ )$
- where i = number of distinct intersection points
- Analysis based on *Euler's formula*
- Observe that $i < k$ for points shared by several segments!

Stricter cost bound

- From $O(\ (n+k)\log n\ )$ ...
- ... to $O(\ (n+i)\log n\ )$
- where i = number of distinct intersection points
- Analysis based on *Euler's formula*
- Observe that $i < k$ for points shared by several segments!

Analysis based on *Euler*'s formula

- For each event point p, $O(\ |L(p) \cup R(p) \cup C(p)|\ )$ $O(\ \log n\ )$ operations on the tree structures...
- Overall, $O(\ \sum_p\ |L(p) \cup R(p) \cup C(p)|\ )$ such operations
- $= O(\ \sum_p\ deg(p)\ )$ by seeing the arrangement of segments as defining a planar graph (p: vertices)
- And since each graph edge contributes 1 to the degree of two graph vertices:

$$\sum_p\ deg(p)\ \leq\ 2E\ =\ O(\ V\ )\ =\ O(\ 2n+i\ )$$

Analysis based on *Euler*'s formula

- For each event point p, $O(\ |L(p) \cup R(p) \cup C(p)|\ )$ $O(\ \log n\ )$ operations on the tree structures...
- Overall, $O(\ \sum_p\ |L(p) \cup R(p) \cup C(p)|\ )$ such operations
- $= O(\ \sum_p\ deg(p)\ )$ by seeing the arrangement of segments as defining a planar graph (p: vertices)
- And since each graph edge contributes 1 to the degree of two graph vertices:

$$\sum_p deg(p) \leq 2E = O(\ V\ ) = O(\ 2n + i\ )$$

Analysis based on *Euler*'s formula

- For each event point p, $O(\ |L(p) \cup R(p) \cup C(p)|\ )$ $O(\ \log n\ )$ operations on the tree structures...
- Overall, $O(\ \sum_p\ |L(p) \cup R(p) \cup C(p)|\ )$ such operations
- $= O(\ \sum_p\ deg(p)\ )$ by seeing the arrangement of segments as defining a planar graph (p: vertices)
- And since each graph edge contributes 1 to the degree of two graph vertices:

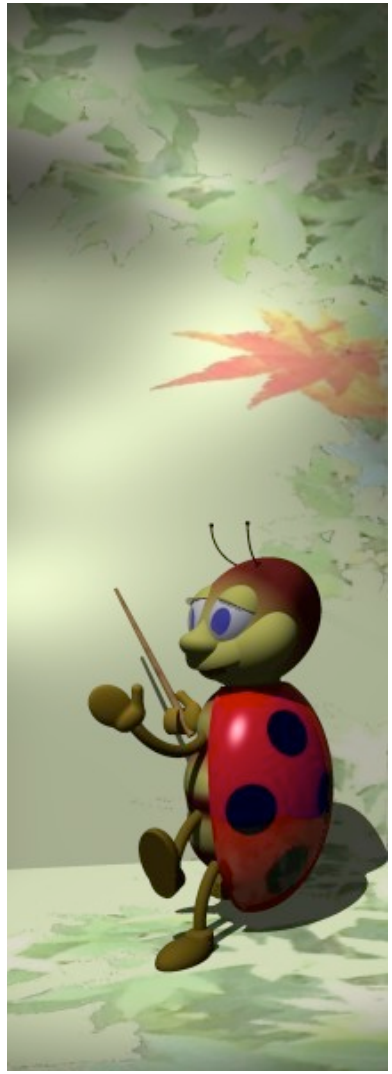
$$\sum_p deg(p) \leq 2E = O(\ V\ ) = O(\ 2n + i\ )$$

Analysis based on *Euler*'s formula

- For each event point p, $O(\ |L(p) \cup R(p) \cup C(p)|\ )$ $O(\ \log n\ )$ operations on the tree structures...
- Overall, $O(\ \sum_p\ |L(p) \cup R(p) \cup C(p)|\ )$ such operations
- $=\ O(\ \sum_p\ deg(p)\ )$ by seeing the arrangement of segments as defining a planar graph (p: vertices)
- And since each graph edge contributes 1 to the degree of two graph vertices:

$$\sum_p\ deg(p)\ \leq\ 2E\ =\ O(\ V\ )\ =\ O(\ 2n+i\ )$$

Outline



Aiming at optimality

- Optimal *output-sensitive* algorithms
- Running time: $O(\, n \log n \, + \, k \,)$
- The optimal trend can be met
  in the worst case (Chazelle & Edelsbrunner, 1988)
  or in the expected case (Mulmuley, 1990)
- More complex structure

Aiming at optimality

- Optimal *output-sensitive* algorithms
- Running time: $O(\, n \log n \, + \, k \,)$
- The optimal trend can be met
  in the worst case (Chazelle & Edelsbrunner, 1988)
  or in the expected case (Mulmuley, 1990)
- More complex structure

Aiming at optimality

- Optimal *output-sensitive* algorithms
- Running time: $O(\, n \log n \, + \, k \,)$
- The optimal trend can be met
  in the worst case (Chazelle & Edelsbrunner, 1988)
  or in the expected case (Mulmuley, 1990)
- More complex structure

Aiming at optimality

- Optimal *output-sensitive* algorithms
- Running time: $O(\ n \log n\ +\ k\ )$
- The optimal trend can be met
  in the worst case (Chazelle & Edelsbrunner, 1988)
  or in the expected case (Mulmuley, 1990)
- More complex structure

Chazelle & Edelsbrunner (1988/1992)

- Mainly of theoretical interest
- Sorage: $O(n + k)$
- Can be (and has been) improved to $O(n)$

Chazelle & Edelsbrunner (1988/1992)

- Mainly of theoretical interest
- Sorage: $O(n + k)$
- Can be (and has been) improved to $O(n)$

Chazelle & Edelsbrunner (1988/1992)

- Mainly of theoretical interest
- Sorage: $O(\ n\ +\ k\ )$
- Can be (and has been) improved to $O(\ n\ )$

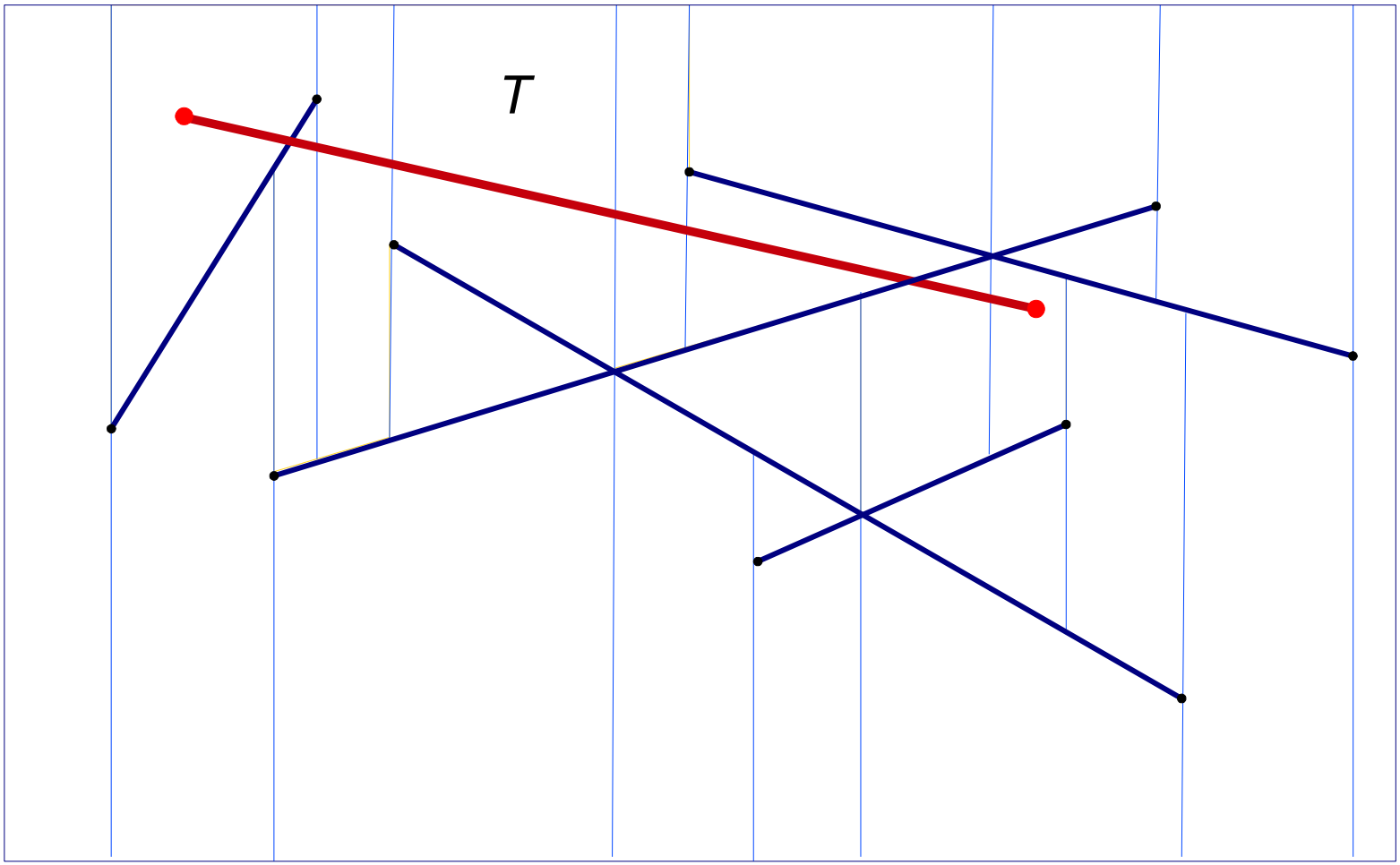
Mulmuley (1990)

- Interesting randomized incremental approach
- Based on a trapezoidal decomposition of the plane

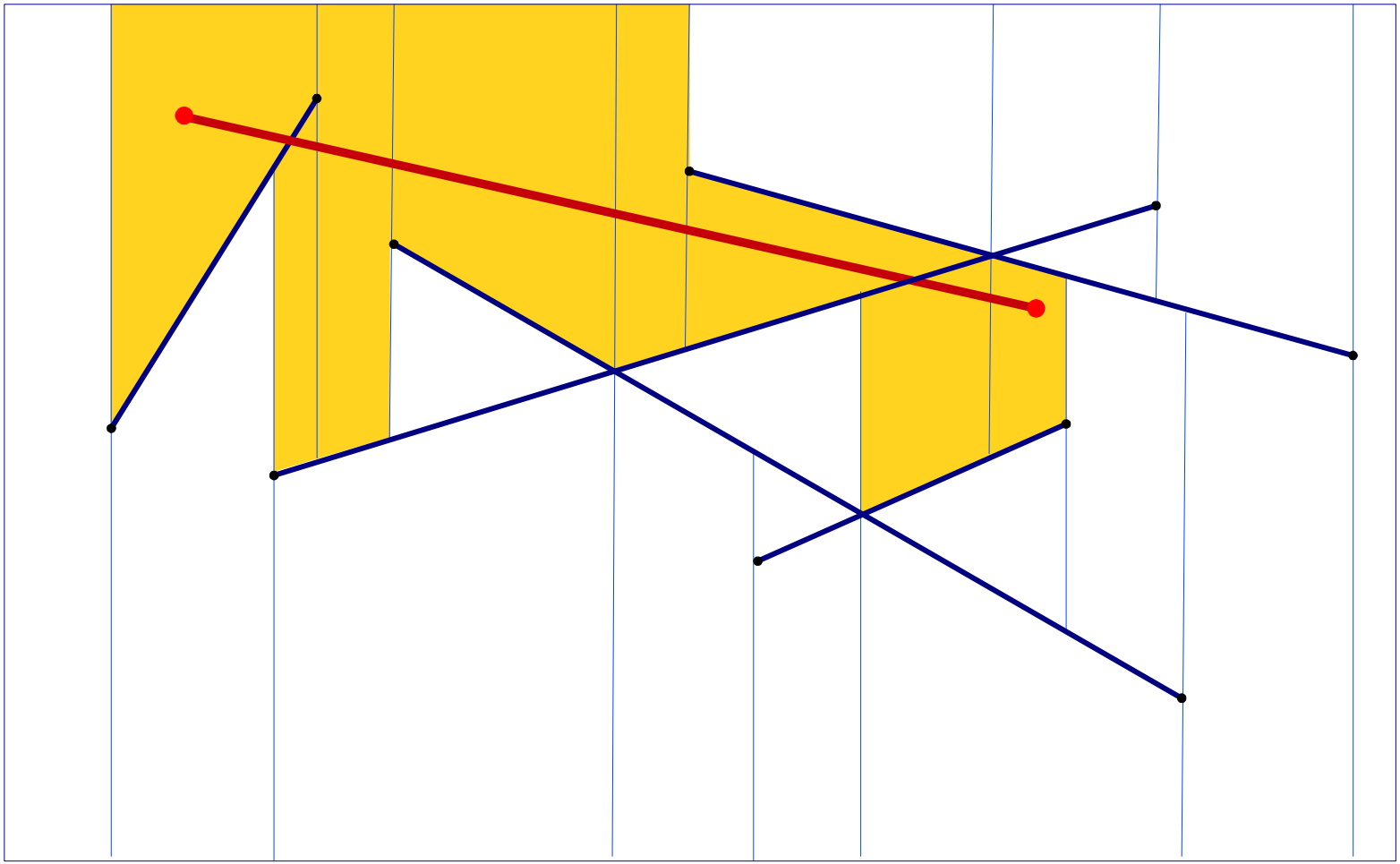
Mulmuley (1990)

- Interesting randomized incremental approach
- Based on a trapezoidal decomposition of the plane

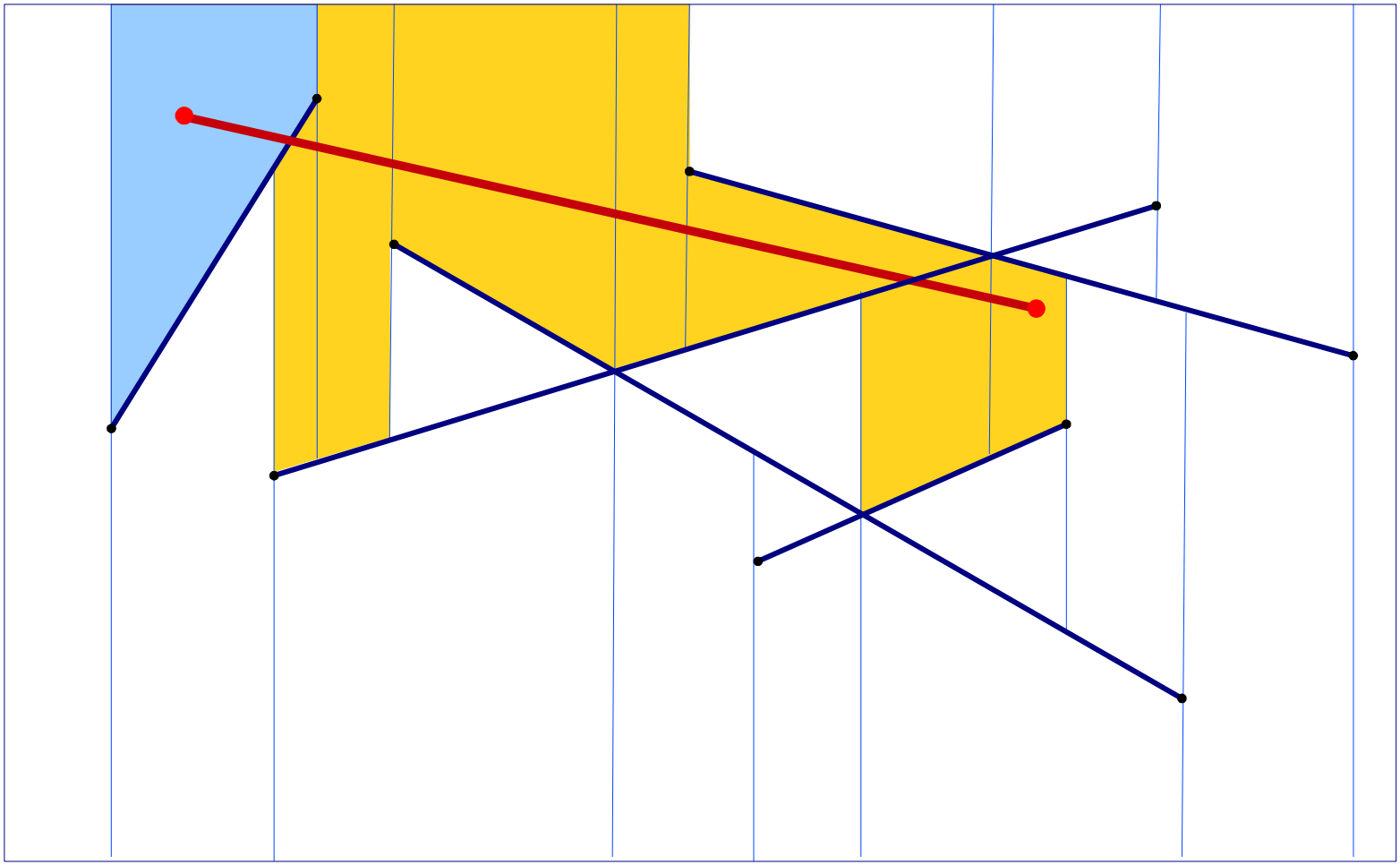
Mulmuley: Trapezoidal subdivision



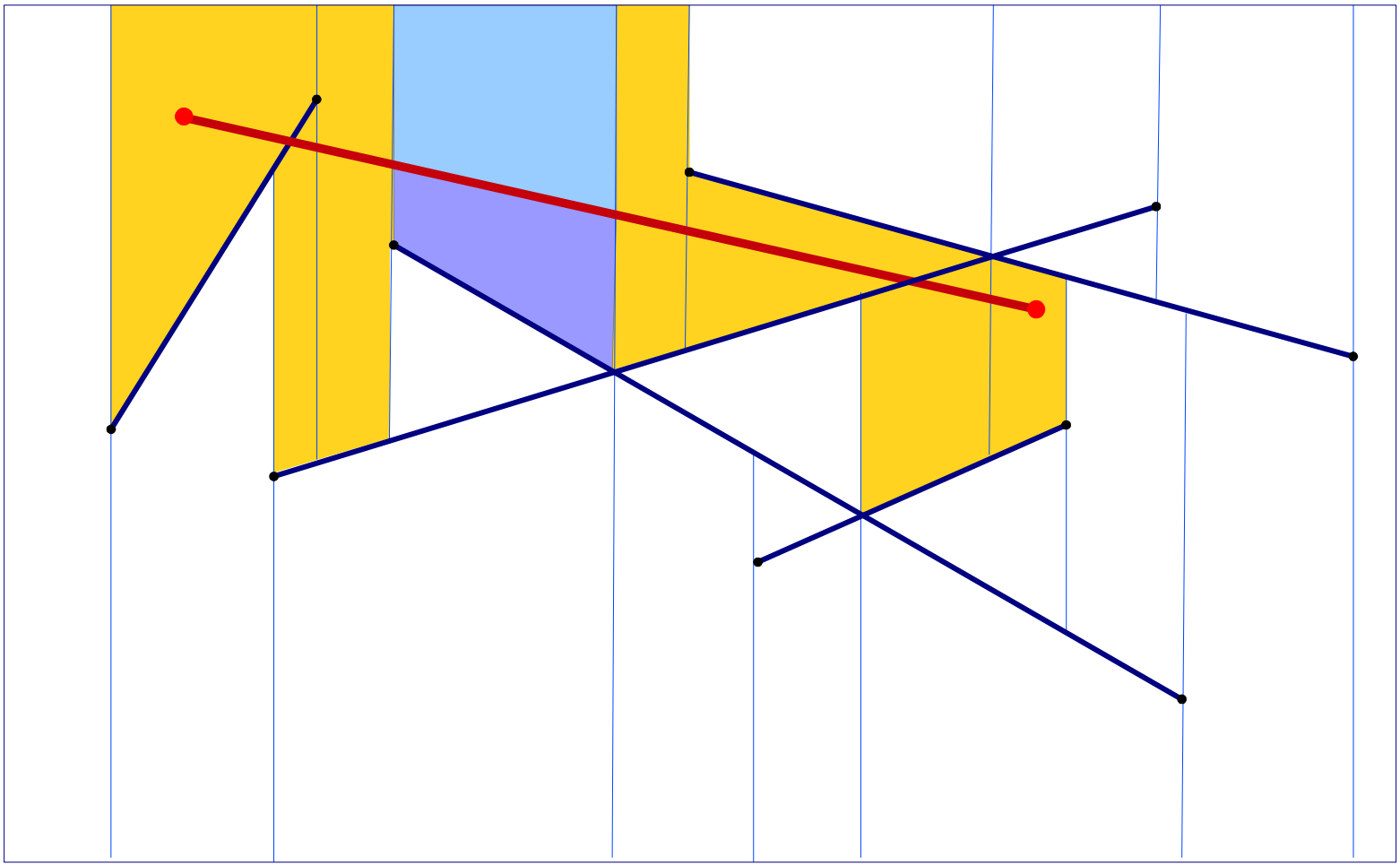
Mulmuley: Conflicts



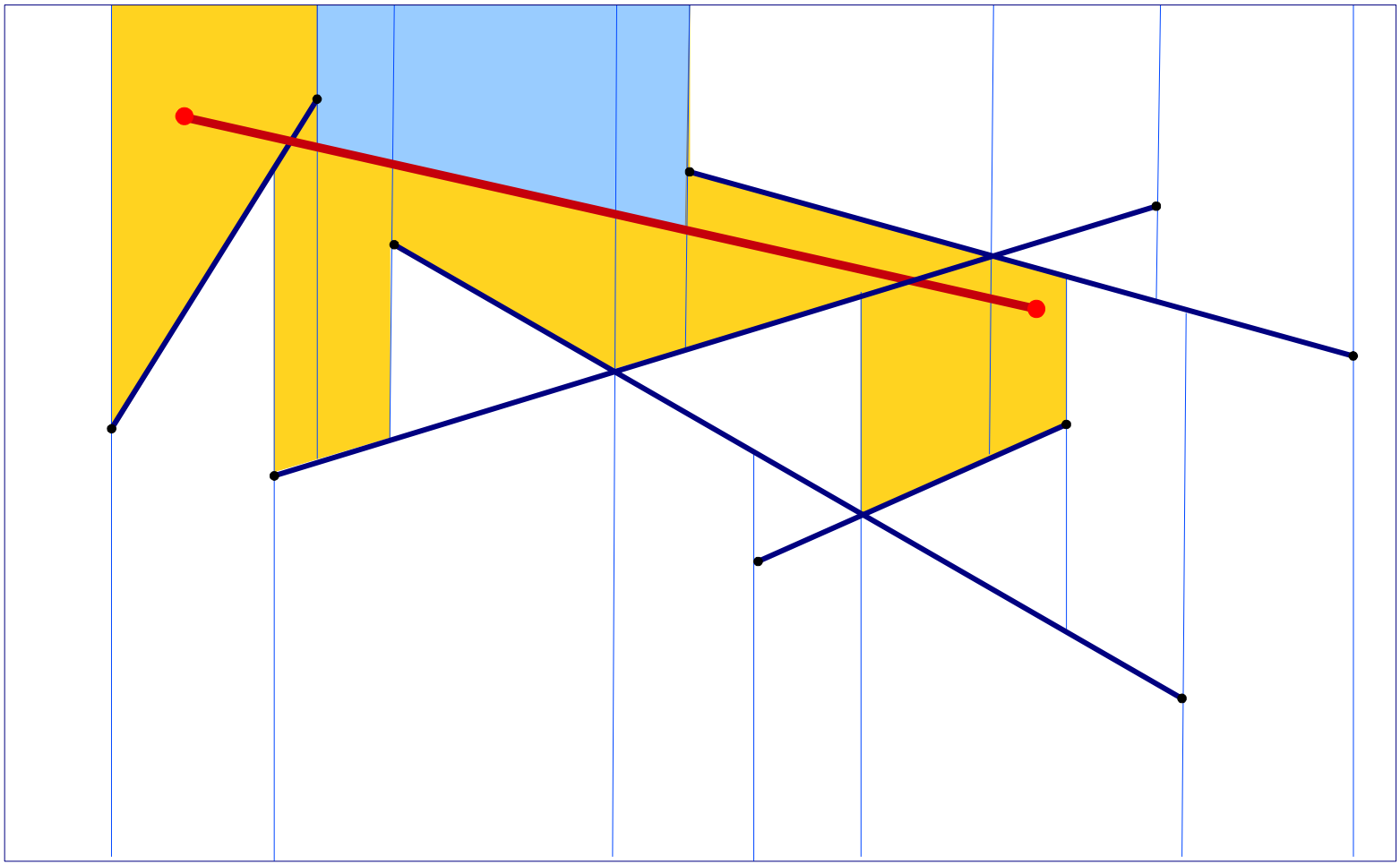
Mulmuley: Point location



Mulmuley: Split



Mulmuley: Merge



Randomized incremental approach

- Trapezoidal decomposition of a rectangular area: T
- Incremental updates: add new segment s
- Conflicts for $s \cap T$: point location / navigate / split / merge
- Influence graph: decomposition "history"
- Expected storage: $O(\ n\ +\ k\ )$

Randomized incremental approach

- Trapezoidal decomposition of a rectangular area: T
- Incremental updates: add new segment s
- Conflicts for $s \cap T$: point location / navigate / split / merge
- Influence graph: decomposition "history"
- Expected storage: $O(n + k)$

Randomized incremental approach

- Trapezoidal decomposition of a rectangular area: T
- Incremental updates: add new segment s
- Conflicts for $s \cap T$: point location / navigate / split / merge
- Influence graph: decomposition "history"
- Expected storage: $O(n + k)$

Randomized incremental approach

- Trapezoidal decomposition of a rectangular area: T
- Incremental updates: add new segment s
- Conflicts for $s \cap T$: point location / navigate / split / merge
- Influence graph: decomposition “history”
- Expected storage: $O(n + k)$

Randomized incremental approach

- Trapezoidal decomposition of a rectangular area: T
- Incremental updates: add new segment s
- Conflicts for $s \cap T$: point location / navigate / split / merge
- Influence graph: decomposition “history”
- Expected storage: $O(\ n\ +\ k\ )$

Randomized incremental approach

- Expected running time: complex analysis
- Assumption: *random sampling*
- Need for a suitable structure
  to navigate the *trapezoidal map*
- E.g. DCEL: Doubly Connected Edge List

Randomized incremental approach

- Expected running time: complex analysis
- Assumption: *random sampling*
- Need for a suitable structure to navigate the *trapezoidal map*
- E.g. DCEL: Doubly Connected Edge List

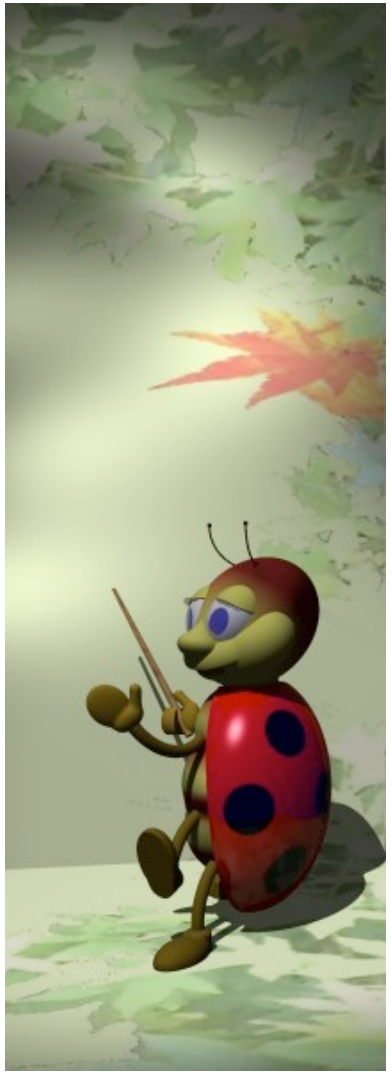
Randomized incremental approach

- Expected running time: complex analysis
- Assumption: *random sampling*
- Need for a suitable structure
  to navigate the *trapezoidal map*
- E.g. DCEL: Doubly Connected Edge List

Randomized incremental approach

- Expected running time: complex analysis
- Assumption: *random sampling*
- Need for a suitable structure
  to navigate the *trapezoidal map*
- E.g. DCEL: Doubly Connected Edge List

Outline



Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line
- Running time costs?
- Storage requirements?

Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line
- Running time costs?
- Storage requirements?

Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line
- Running time costs?
- Storage requirements?

Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line
- Running time costs?
- Storage requirements?

Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line (enriched with *previous*/*next* links)
- Running time costs? $O(\, n \log n \, + \, k \,)$
- Storage requirements?

Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line (enriched with *previous*/*next* links)
- Running time costs? $O(\, n \log n \, + \, k \,)$
- Storage requirements?

Horizontal and vertical segments

- Bentley & Ottmann (1979)
- Events: endpoints of horizontal segments (*ENTER*/*EXIT*), lower endpoint of vertical segments (*VERTICAL*)
- Sweep line structure: horizontal segments crossing the sweep line (enriched with *previous*/*next* links)
- Running time costs? $O(\, n \log n \, + \, k \,)$
- Storage requirements? $O(\, n \,)$

Just counting intersections

- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but. . .
- How could we count intersections in $O(\log n)$ per event?
- Add. . .

  . . .
- Running time costs?

Just counting intersections

- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but. . .
- How could we count intersections in $O(\log n)$ per event?
- Add. . .

  . . .
- Running time costs?

Just counting intersections

- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but. . .
- How could we count intersections in $O(\log n)$ per event?
- Add. . .

  . . .
- Running time costs?

Just counting intersections

- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but...
- How could we count intersections in $O(\log n)$ per event?
- Add...

  ...
- Running time costs?

Just counting intersections

- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but. . .
- How could we count intersections in $O(\log n)$ per event?
- Add counting information to nodes of the search tree, e.g. size of each subtree stored at its root
- Running time costs?

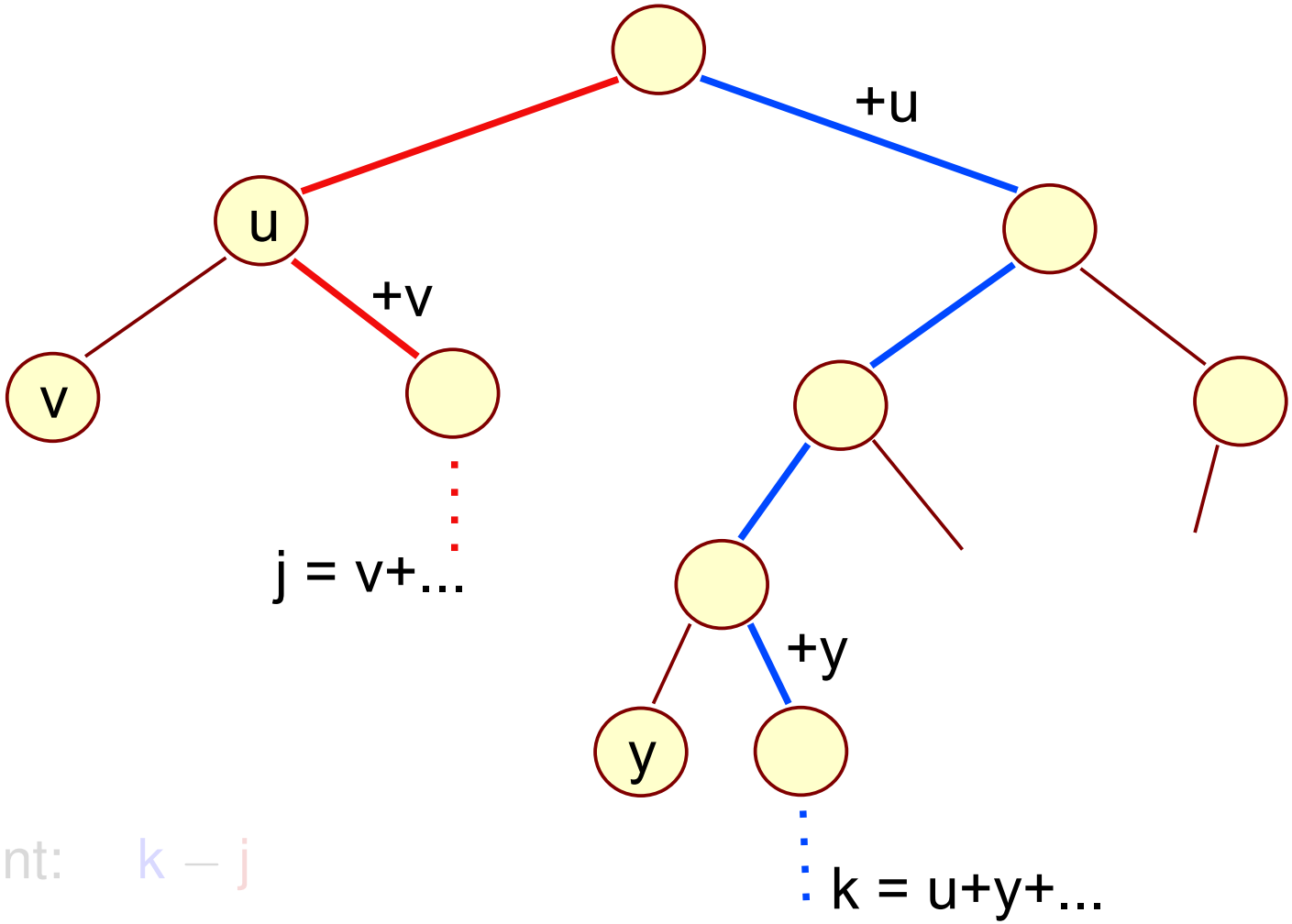
Just counting intersections

- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but. . .
- How could we count intersections in $O(\log n)$ per event?
- Add counting information to nodes of the search tree, e.g. size of each subtree stored at its root
- Running time costs?

Just counting intersections

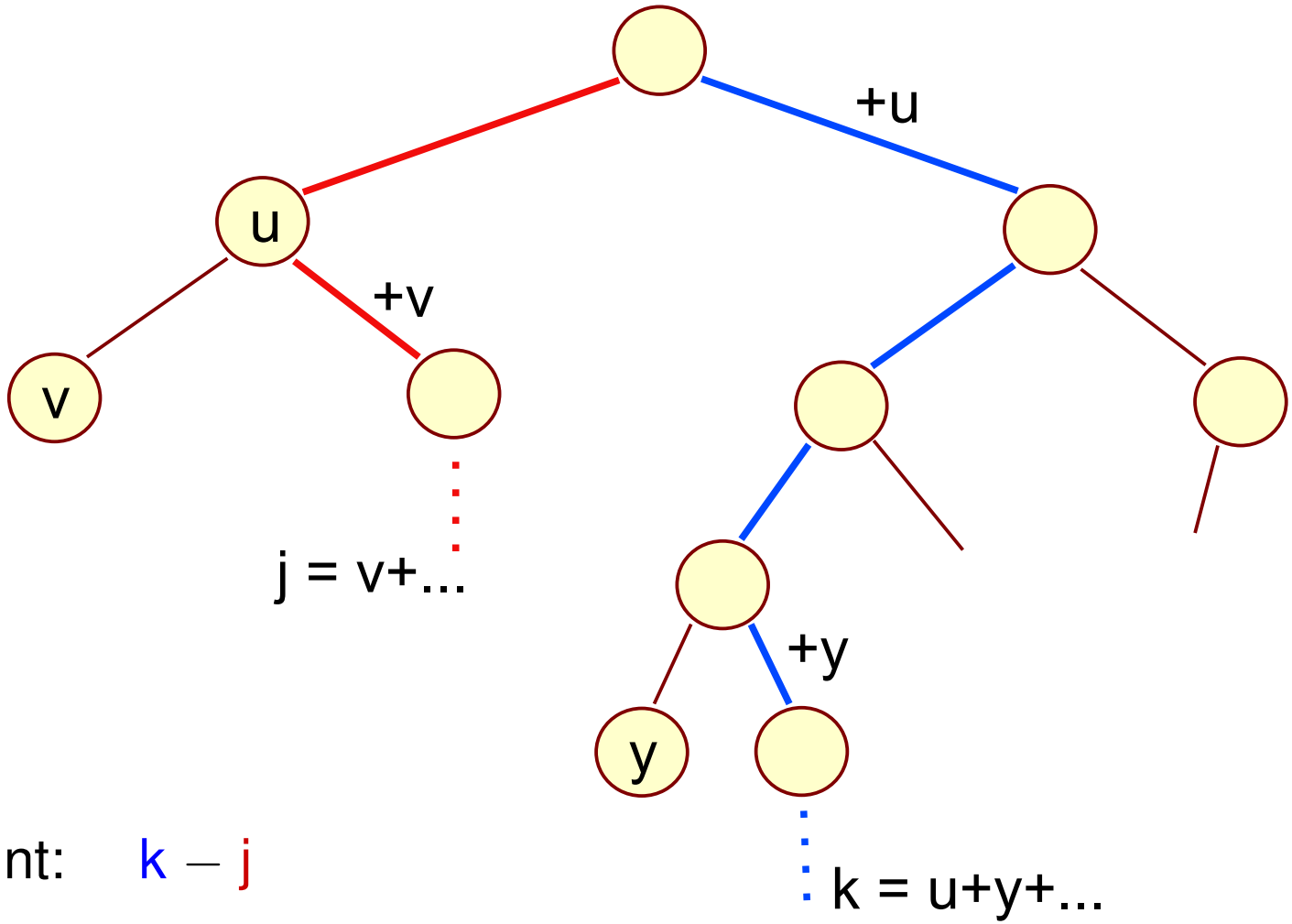
- Bentley & Ottmann (1979)
- Events: as above
- Sweep line structure: as above but...
- How could we count intersections in $O(\log n)$ per event?
- Add counting information to nodes of the search tree, e.g. size of each subtree stored at its root
- Running time costs? $O(n \log n)$

Node counting information: segments “below”



Count: k − j

Node counting information: segments “below”



Count: $k - j$

Node counting information

- Insert segment: $+1$ for each node along the path
- Remove segment: -1 for each node along the path
- Local adjustments when rebalancing the tree:

$$\textit{node}\ \text{count} = \textit{left son}\ \text{count} + \textit{right son}\ \text{count}\ [+1]$$

Node counting information

- Insert segment: $+1$ for each node along the path
- Remove segment: -1 for each node along the path
- Local adjustments when rebalancing the tree:

$$\textit{node}\ \text{count} = \textit{left son}\ \text{count} + \textit{right son}\ \text{count}\ [+\,1]$$

Node counting information

- Insert segment: $+1$ for each node along the path
- Remove segment: -1 for each node along the path
- Local adjustments when rebalancing the tree:

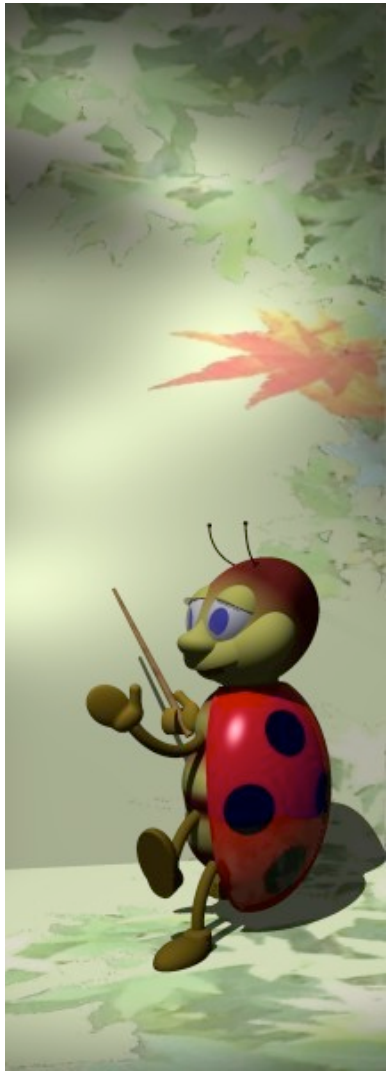
  node count = *left son* count + *right son* count [+ 1]

Node counting information

- Insert segment: $+1$ for each node along the path
- Remove segment: -1 for each node along the path
- Local adjustments when rebalancing the tree:

  node count = *left son* count + *right son* count [+ 1]

Outline



References


M.I. Shamos & D.J. Hoey (1976)
Geometric intersection problems
Proc. Conf. Foundations of Computer Science


J.L. Bentley & T.A. Ottmann (1979)
Algorithms for Reporting and Counting Geometric Intersections
IEEE Trans. on Computer, C-28(9)


K. Mulmuley (1990)
A Fast Planar Partition Algorithm, I
J. Symbolic Computation, 10

References


B. Chazelle & H. Edelsbrunner (1992)
An Optimal Algorithm for Intersecting Line Segments in the Plane
J. ACM, 39(1)


D.P. Dobkin & R.J. Lipton (1979)
On the Complexity of Computations under Varying Sets of Primitives
J. of Computer and System Sciences, 18