Point Location in Trapezoidal Maps

Claudio Mirolo

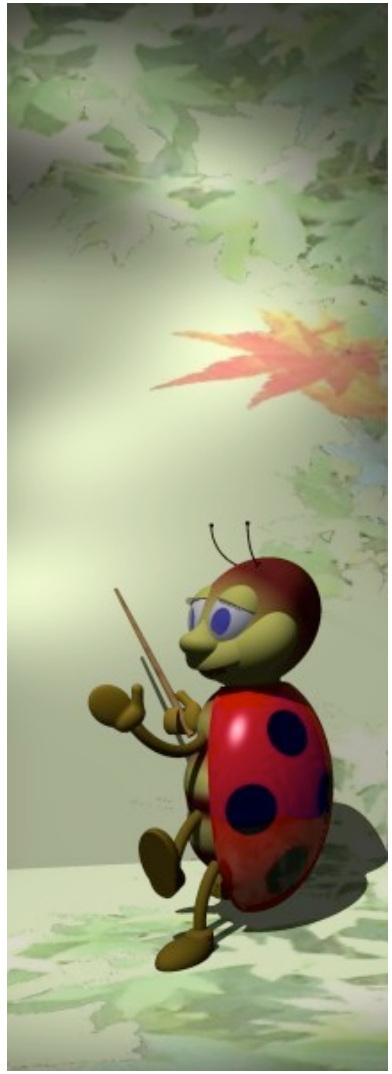
Dip. di Scienze Matematiche, Informatiche e Fisiche
Università di Udine, via delle Scienze 206 – Udine

claudio.mirolo@uniud.it

Computational Geometry

users.dimi.uniud.it/∼claudio.mirolo

Outline



Planar *point location* problem

- For a planar subdivsion S with n edges
- Given a query point q
- Report the face f (edge e, vertex v) of S such that $q \in f$ ($q \in e$, $q = v$)
- Efficiently! (*Preprocessing*)

Planar *point location* problem

- For a planar subdivsion S with n edges
- Given a query point q
- Report the face f (edge e, vertex v) of S such that $q \in f$ ($q \in e$, $q = v$)
- Efficiently! (*Preprocessing*)

Planar *point location* problem

- For a planar subdivsion S with n edges
- Given a query point q
- Report the face f (edge e, vertex v) of S such that $q \in f$ ($q \in e$, $q = v$)
- Efficiently! (*Preprocessing*)

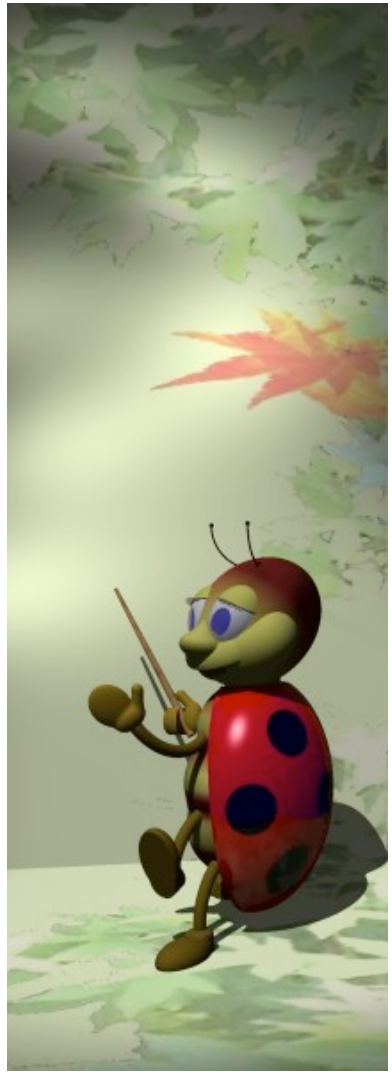
Planar *point location* problem

- For a planar subdivsion S with n edges
- Given a query point q
- Report the face f (edge e, vertex v) of S such that $q \in f$ ($q \in e$, $q = v$)
- Efficiently! (*Preprocessing*)

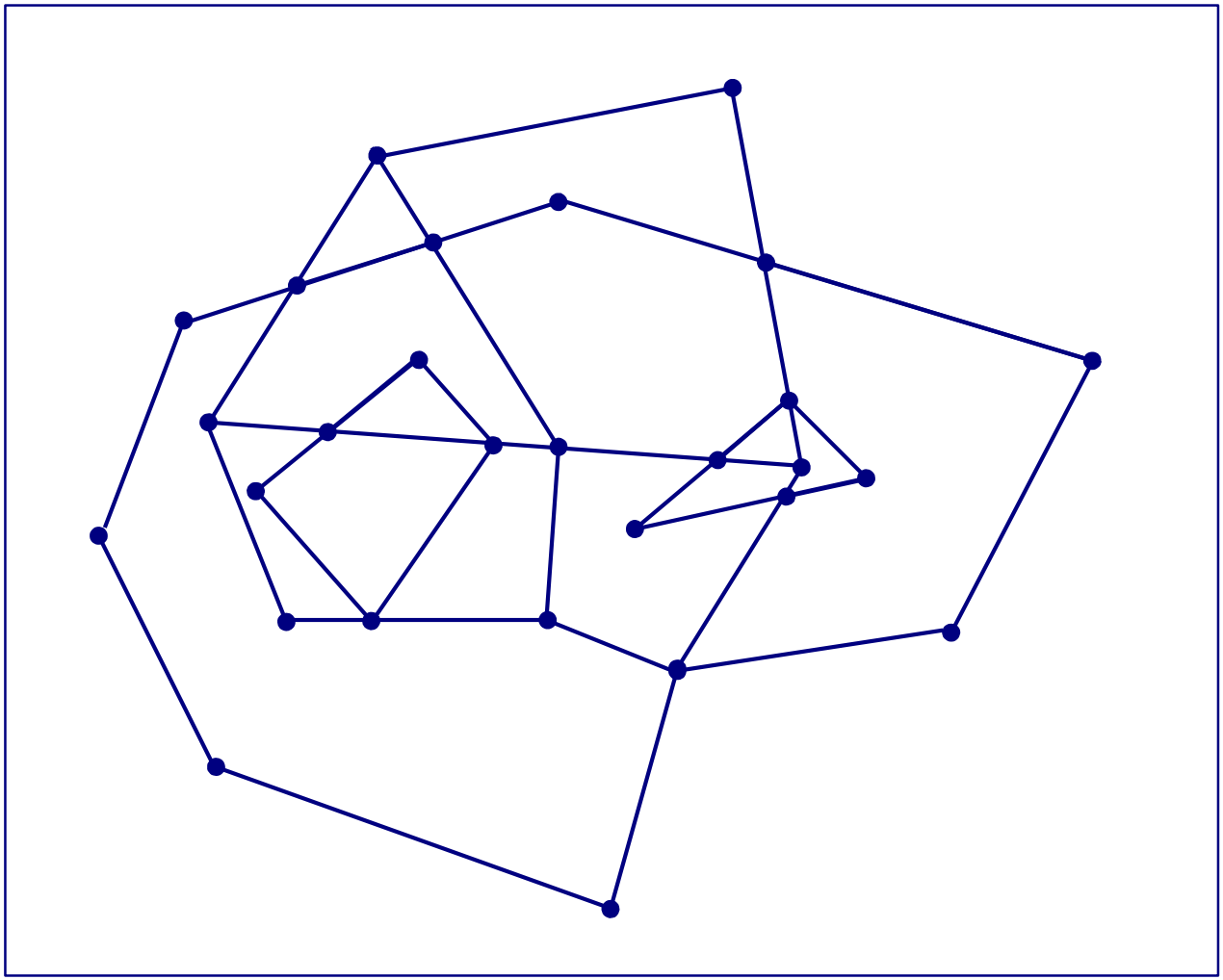
Planar *point location* problem

- For a planar subdivsion S with n edges
- Given a query point q
- Report the face f (edge e, vertex v) of S such that $q \in f$ ($q \in e$, $q = v$)
- Efficiently! (*Preprocessing*)

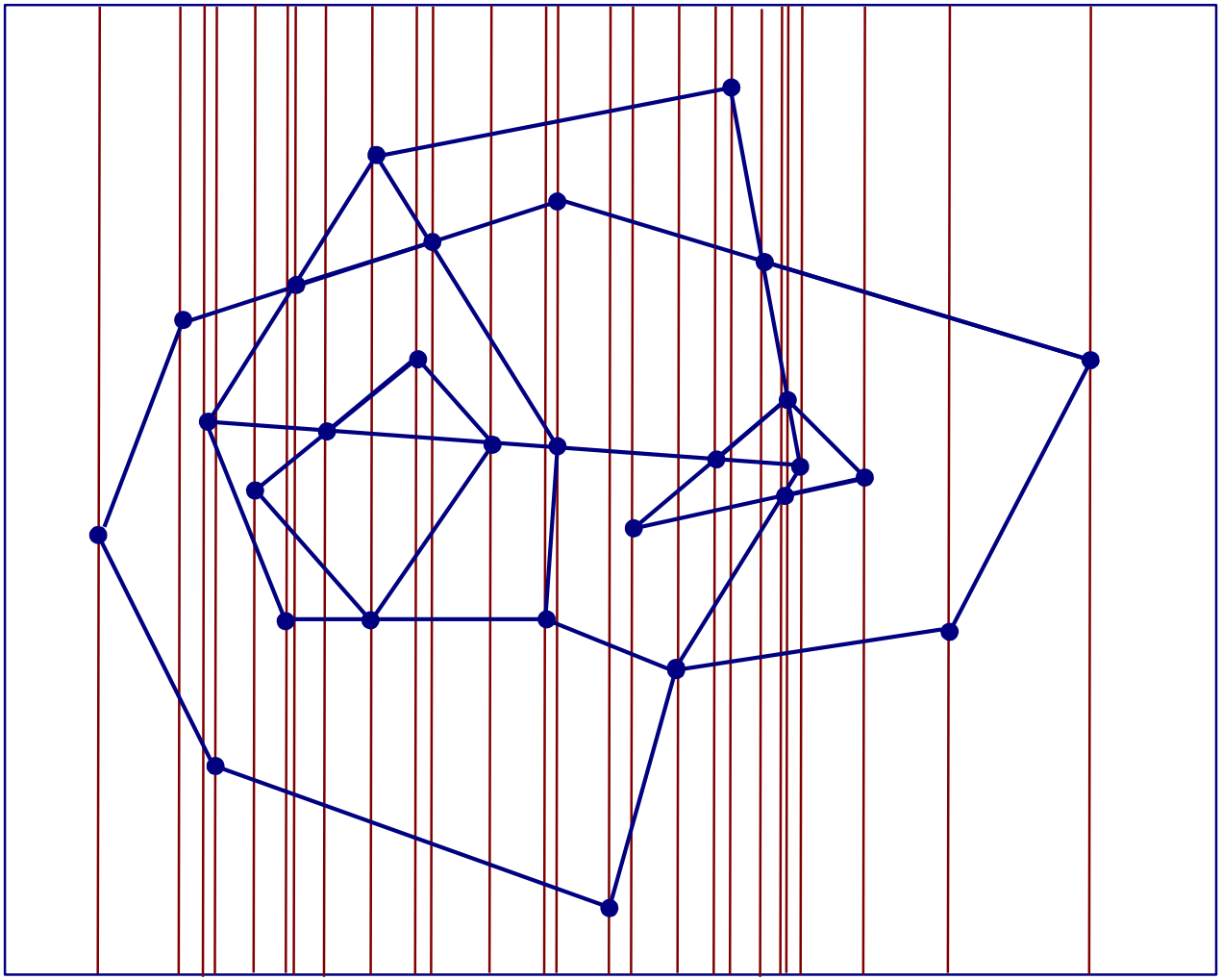
Outline



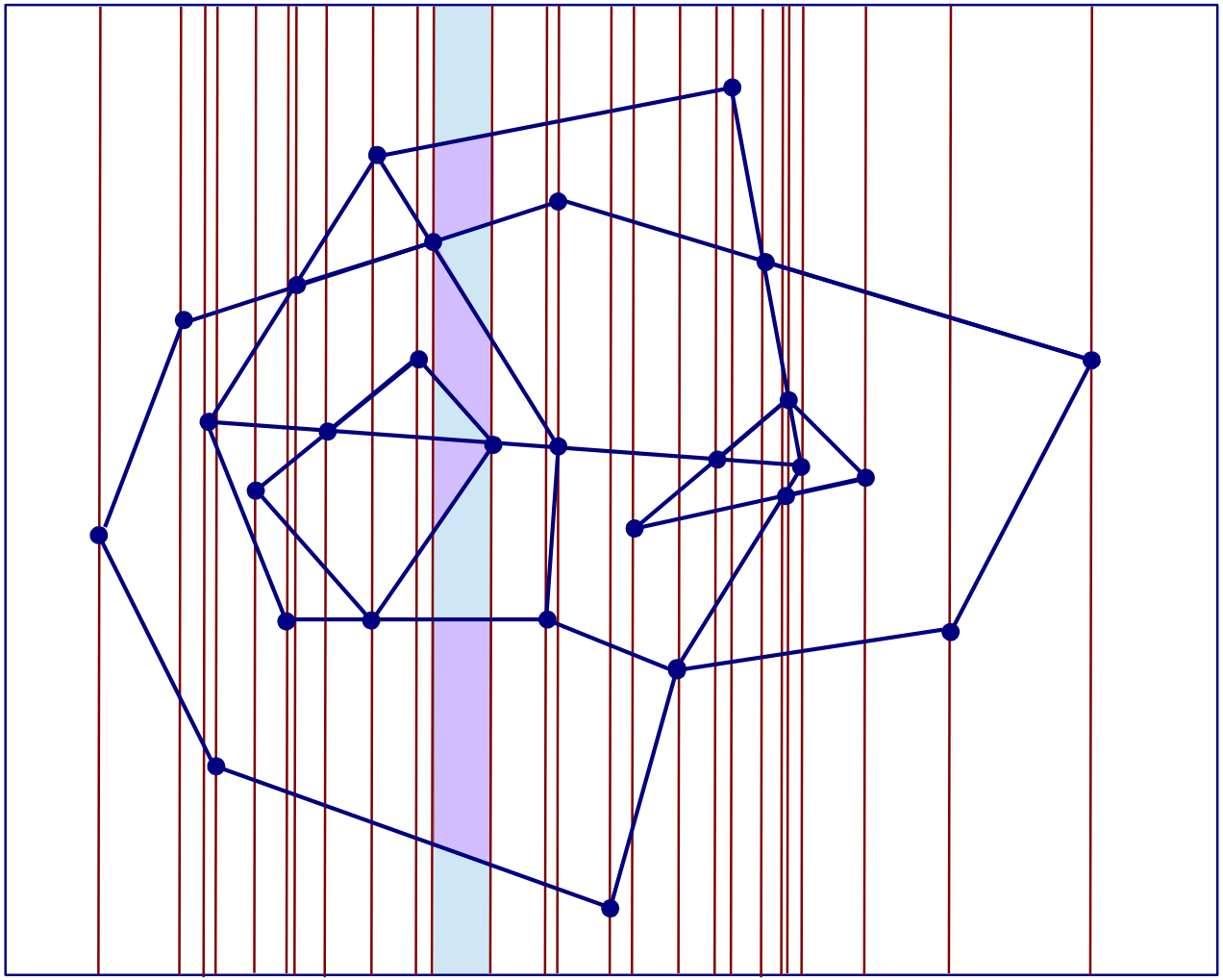
Naïve trapezoidal map of a planar subdivision



Naïve trapezoidal map of a planar subdivision



Naïve trapezoidal map of a planar subdivision: *slab*



Preprocessing

- *Bounding box* (just for the sake of simplicity)
- Vertical lines are drawn through all vertices
- Vertical *slabs* are sorted left to right (array, BST . . .)
- *Trapezoids* within a slab are sorted bottom to top (arrays, BSTs . . .)

Preprocessing

- *Bounding box* (just for the sake of simplicity)
- Vertical lines are drawn through all vertices
- Vertical *slabs* are sorted left to right (array, BST . . .)
- *Trapezoids* within a slab are sorted bottom to top (arrays, BSTs . . .)

Preprocessing

- *Bounding box* (just for the sake of simplicity)
- Vertical lines are drawn through all vertices
- Vertical *slabs* are sorted left to right
  (array, BST . . .)
- *Trapezoids* within a slab are sorted bottom to top
  (arrays, BSTs . . .)

Preprocessing

- *Bounding box* (just for the sake of simplicity)
- Vertical lines are drawn through all vertices
- Vertical *slabs* are sorted left to right
  (array, BST . . .)
- *Trapezoids* within a slab are sorted bottom to top
  (arrays, BSTs . . .)

Preprocessing

- *Bounding box* (just for the sake of simplicity)
- Vertical lines are drawn through all vertices
- Vertical *slabs* are sorted left to right
  (array, BST . . .)
- *Trapezoids* within a slab are sorted bottom to top
  (arrays, BSTs . . .)

Point location

- Binary search for the *slab* containing the query point q
- Binary search within the slab
  for the *trapezoid* containing the query point q
- No more than $2n + 1$ slabs
  and $n + 1$ trapezoids within a slab
- Point location cost: $O(\log n)$ per query

Point location

- Binary search for the *slab* containing the query point q
- Binary search within the slab
  for the *trapezoid* containing the query point q
- No more than $2n + 1$ slabs
  and $n + 1$ trapezoids within a slab
- Point location cost: $O(\log n)$ per query

Point location

- Binary search for the *slab* containing the query point q
- Binary search within the slab
  for the *trapezoid* containing the query point q
- No more than $2n + 1$ slabs
  and $n + 1$ trapezoids within a slab
- Point location cost: $O(\log n)$ per query

Point location

- Binary search for the *slab* containing the query point q
- Binary search within the slab
  for the *trapezoid* containing the query point q
- No more than $2n + 1$ slabs
  and $n + 1$ trapezoids within a slab
- Point location cost: $O(\log n)$ per query

Point location

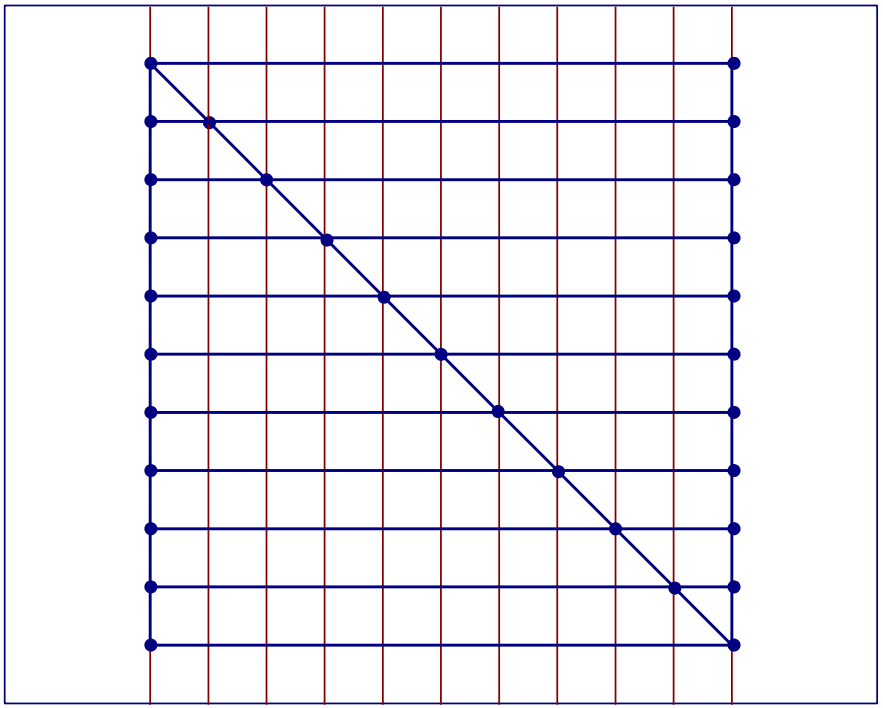
- Binary search for the *slab* containing the query point q
- Binary search within the slab
  for the *trapezoid* containing the query point q
- No more than $2n + 1$ slabs
  and $n + 1$ trapezoids within a slab
- Point location cost: $O(\log n)$ per query (good! but...)

Heavy data structure

Easy to figure out worst-case arrangements requiring $O(n^2)$ raw storage

Heavy data structure

Easy to figure out worst-case arrangements requiring $O(n^2)$ raw storage



Map features

- Provisional *general position* assumption: no two vertices of S with the same x
- Upward and downward *vertical extensions* from each vertex of S
- The extensions stop when they meet an edge of S or a wall of the bounding box B
- *Trapezoidal map* of S = subdivision induced by S, B + *upper* and *lower* vertical extensions

Map features

- Provisional *general position* assumption: no two vertices of S with the same x
- Upward and downward *vertical extensions* from each vertex of S
- The extensions stop when they meet an edge of S or a wall of the bounding box B
- *Trapezoidal map* of S = subdivision induced by S, B + *upper* and *lower* vertical extensions

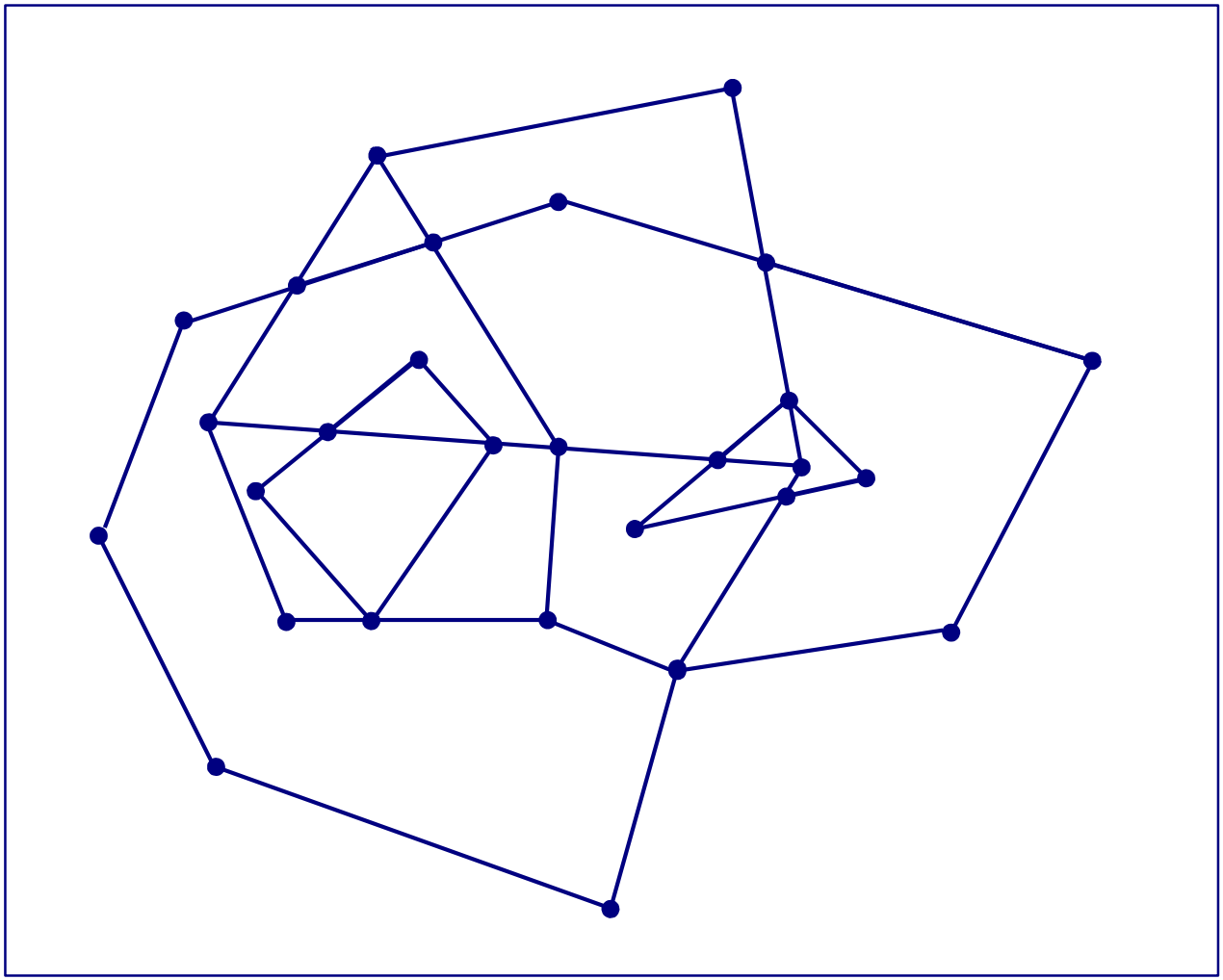
Map features

- Provisional *general position* assumption: no two vertices of S with the same x
- Upward and downward *vertical extensions* from each vertex of S
- The extensions stop when they meet an edge of S or a wall of the bounding box B
- *Trapezoidal map* of S = subdivision induced by S, B + *upper* and *lower* vertical extensions

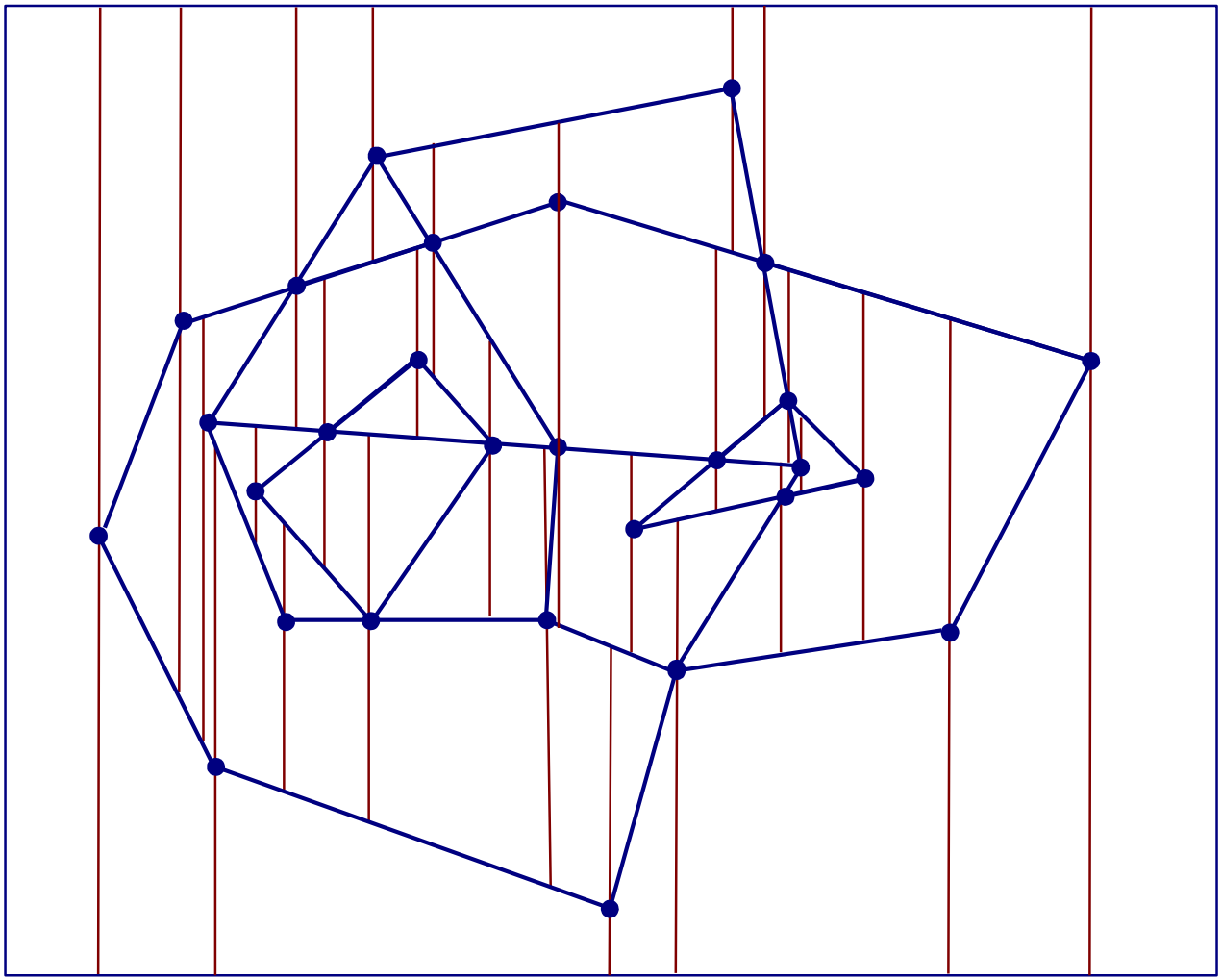
Map features

- Provisional *general position* assumption: no two vertices of S with the same x
- Upward and downward *vertical extensions* from each vertex of S
- The extensions stop when they meet an edge of S or a wall of the bounding box B
- *Trapezoidal map* of S = subdivision induced by S, B + *upper* and *lower* vertical extensions

Trapezoidal map of a planar subdivision



Trapezoidal map of a planar subdivision



Refined subdivision

- n original edges and $\Theta(n)$ original vertices
- Two new vertices added for each original vertex
- Overall $\Theta(n)$ edges and vertices
- What results is still a planar subdivision: $O(n)$ faces

Refined subdivision

- n original edges and $\Theta(n)$ original vertices
- Two new vertices added for each original vertex
- Overall $\Theta(n)$ edges and vertices
- What results is still a planar subdivision: $O(n)$ faces

Refined subdivision

- n original edges and $\Theta(n)$ original vertices
- Two new vertices added for each original vertex
- Overall $\Theta(n)$ edges and vertices
- What results is still a planar subdivision: $O(n)$ faces

Refined subdivision

- n original edges and $\Theta(n)$ original vertices
- Two new vertices added for each original vertex
- Overall $\Theta(n)$ edges and vertices
- What results is still a planar subdivision: $O(n)$ faces

Type of regions

- Regions between two original segments (above/below) and two vertical extensions (left/right)
- Possibly one degenerate vertical wall $\rightarrow$ point
- Possibly *bounding box*'s wall(s)
  instead of original segment(s) or vertical extension(s)
- *Trapezoids* and *triangles* (= degenerate trapezoids)

Type of regions

- Regions between two original segments (above/below) and two vertical extensions (left/right)
- Possibly one degenerate vertical wall $\rightarrow$ point
- Possibly *bounding box*'s wall(s) instead of original segment(s) or vertical extension(s)
- *Trapezoids* and *triangles* (= degenerate trapezoids)

Type of regions

- Regions between two original segments (above/below) and two vertical extensions (left/right)
- Possibly one degenerate vertical wall $\rightarrow$ point
- Possibly *bounding box*'s wall(s) instead of original segment(s) or vertical extension(s)
- *Trapezoids* and *triangles* (= degenerate trapezoids)

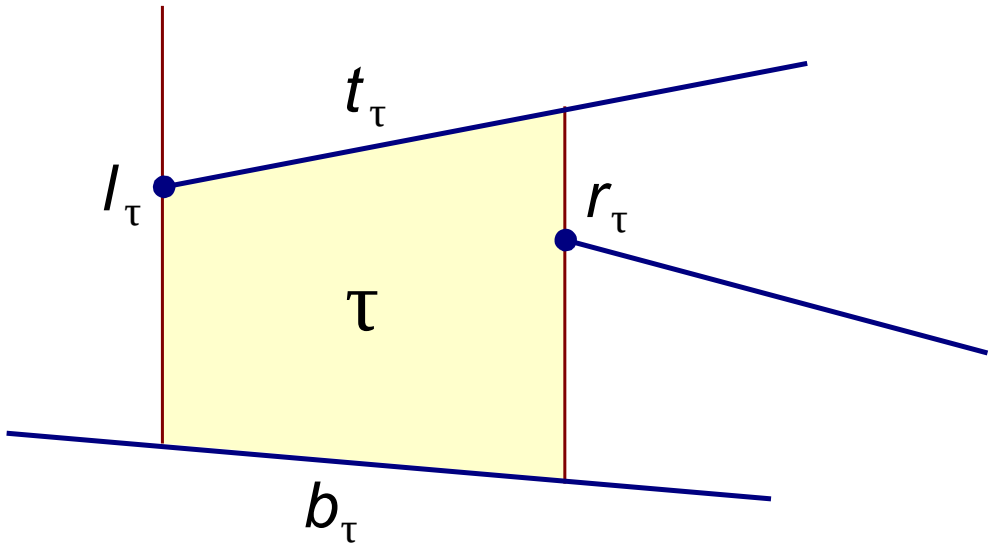
Type of regions

- Regions between two original segments (above/below) and two vertical extensions (left/right)
- Possibly one degenerate vertical wall $\rightarrow$ point
- Possibly *bounding box*'s wall(s) instead of original segment(s) or vertical extension(s)
- *Trapezoids* and *triangles* (= degenerate trapezoids)

Items defining a trapezoid

Trapezoid τ :

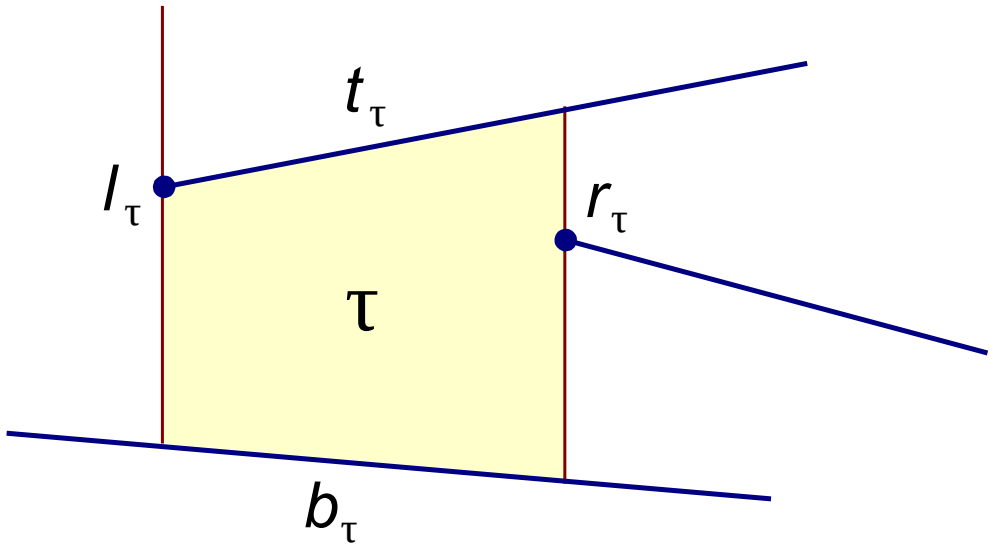
- Top edge: t_τ
- Bottom edge: b_τ
- Left vertex: l_τ
- Right vertex: r_τ
- (possibly horizontal walls / vertices of the bounding box)



Items defining a trapezoid

Trapezoid τ :

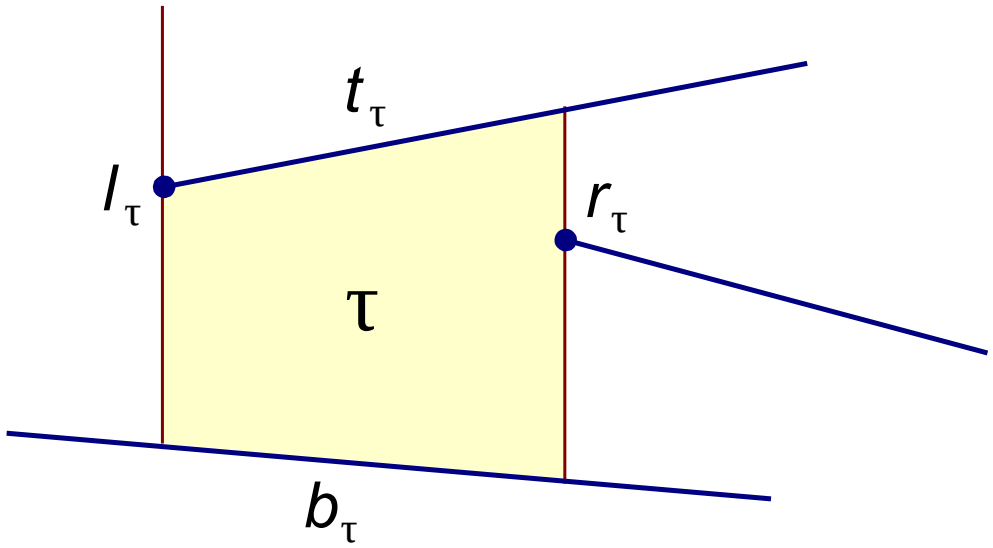
- Top edge: t_τ
- Bottom edge: b_τ
- Left vertex: l_τ
- Right vertex: r_τ
- (possibly horizontal walls / vertices of the bounding box)



Items defining a trapezoid

Trapezoid τ :

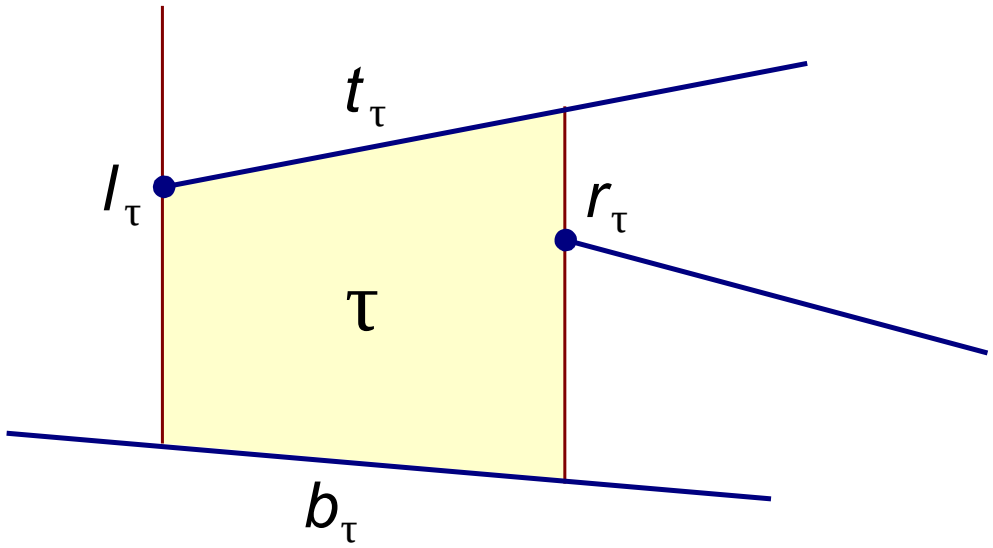
- Top edge: t_τ
- Bottom edge: b_τ
- Left vertex: l_τ
- Right vertex: r_τ
- (possibly horizontal walls / vertices of the bounding box)



Items defining a trapezoid

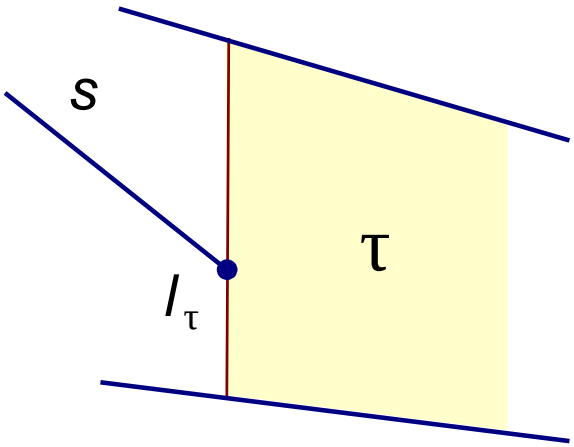
Trapezoid τ :

- Top edge: t_τ
- Bottom edge: b_τ
- Left vertex: l_τ
- Right vertex: r_τ

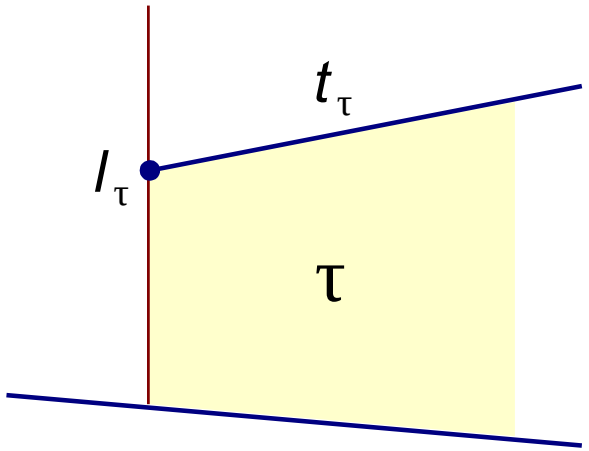


- (possibly horizontal walls / vertices of the bounding box)

Classification based on left boundary

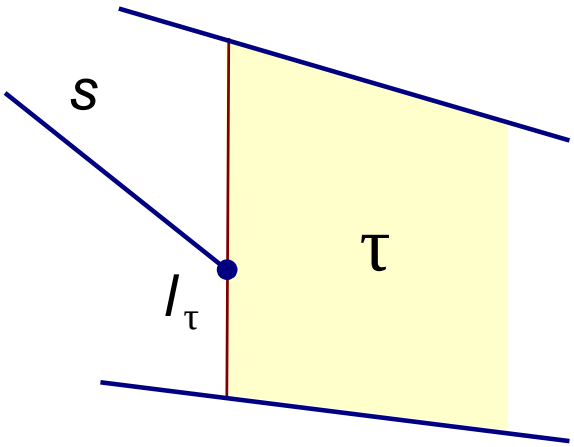


whole vertical extension
of original endpoint l_τ

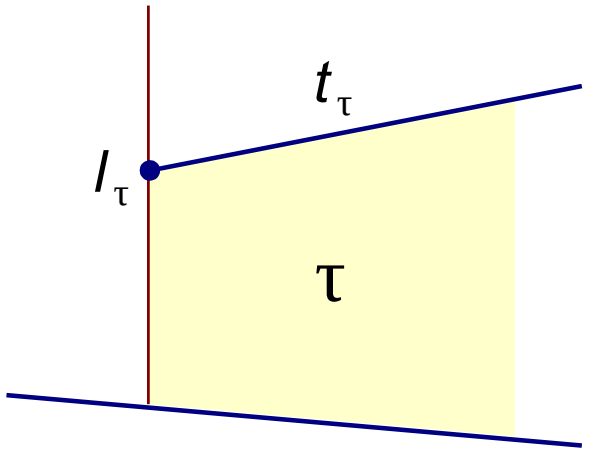


lower vertical extension
of original endpoint l_τ

Classification based on left boundary

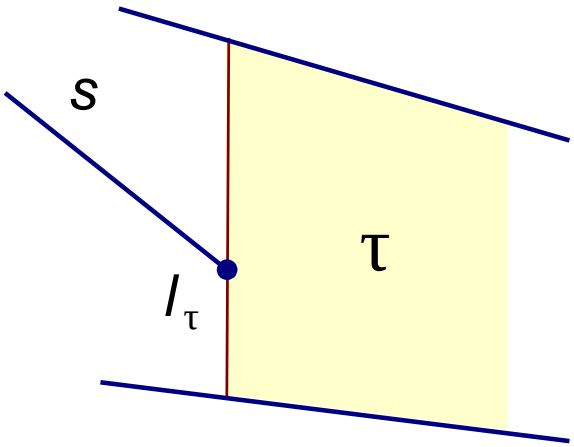


whole vertical extension
of original endpoint l_τ

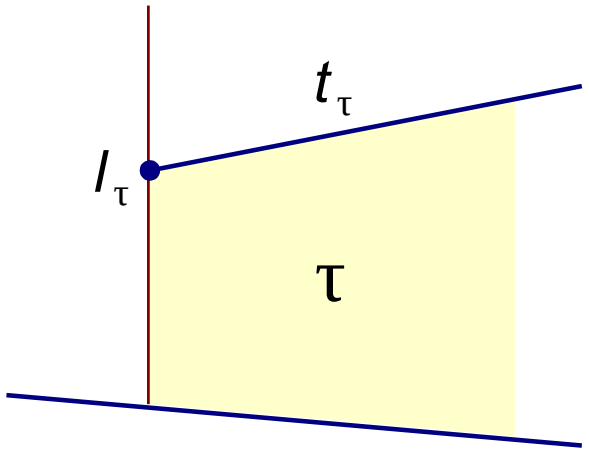


lower vertical extension
of original endpoint l_τ

Classification based on left boundary

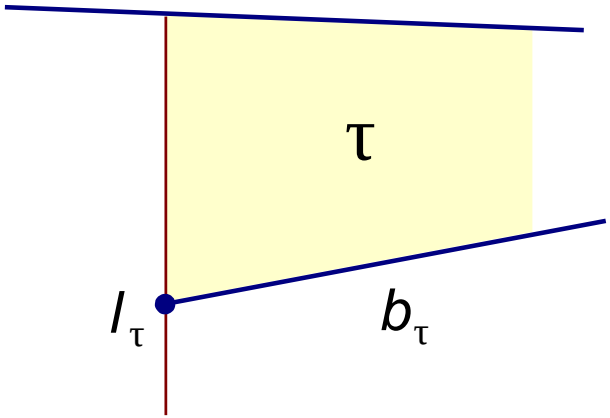


whole vertical extension
of original endpoint l_τ

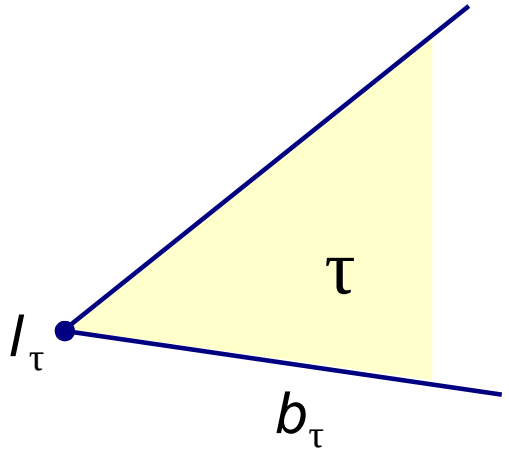


lower vertical extension
of original endpoint l_τ

Classification based on left boundary

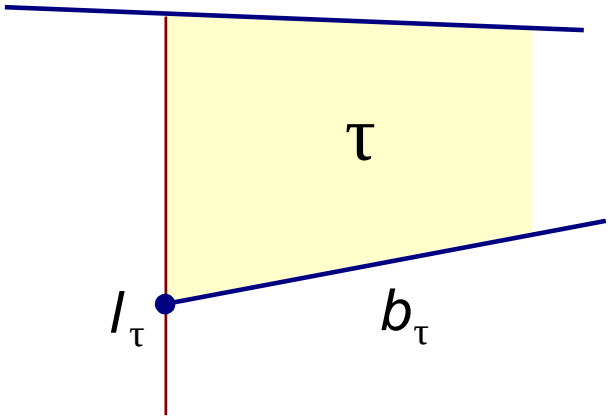


upper vertical extension
of original endpoint l_τ

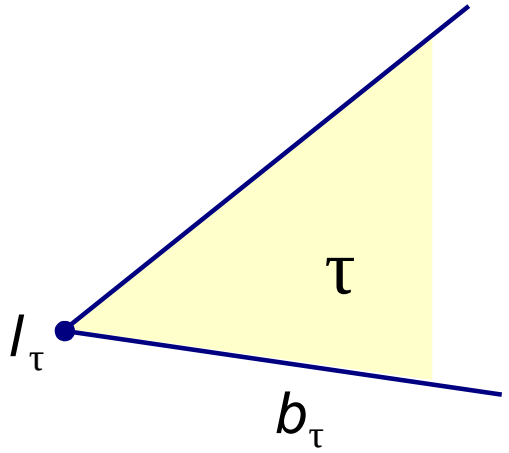


meeting point l_τ
of two original segments
degenerate case of ←

Classification based on left boundary

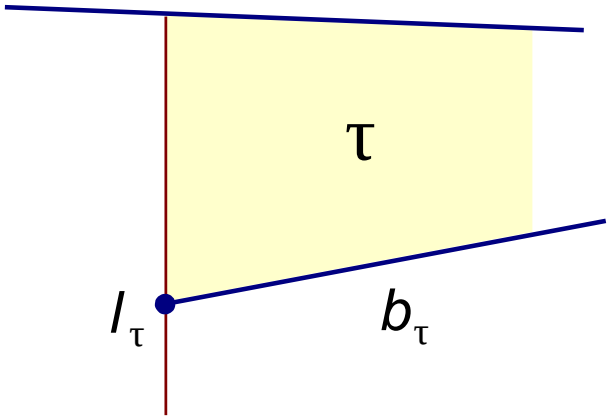


upper vertical extension
of original endpoint l_τ

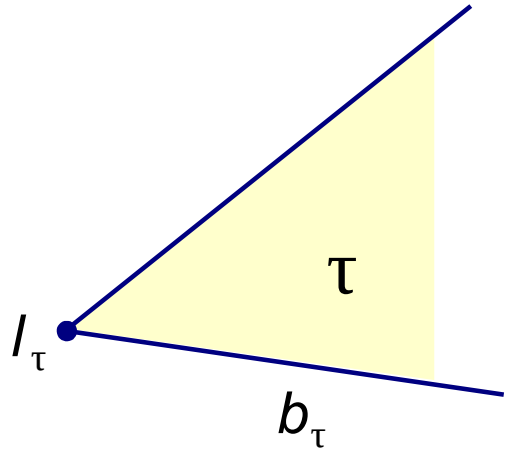


meeting point l_τ
of two original segments
degenerate case of ←

Classification based on left boundary

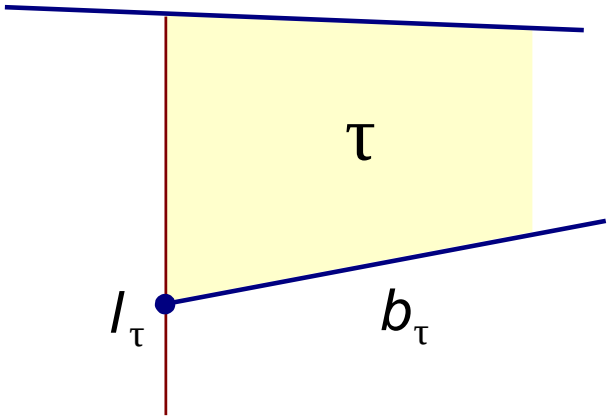


upper vertical extension
of original endpoint l_τ

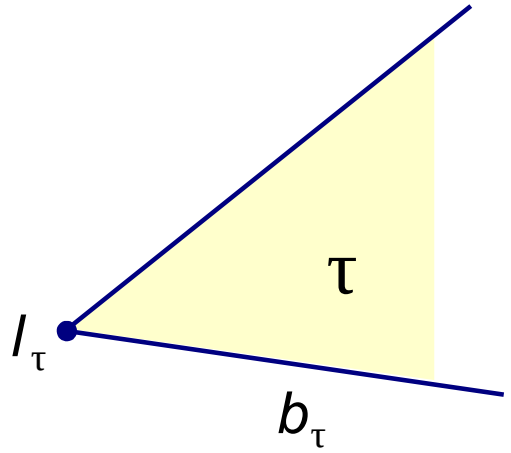


meeting point l_τ
of two original segments
degenerate case of ←

Classification based on left boundary

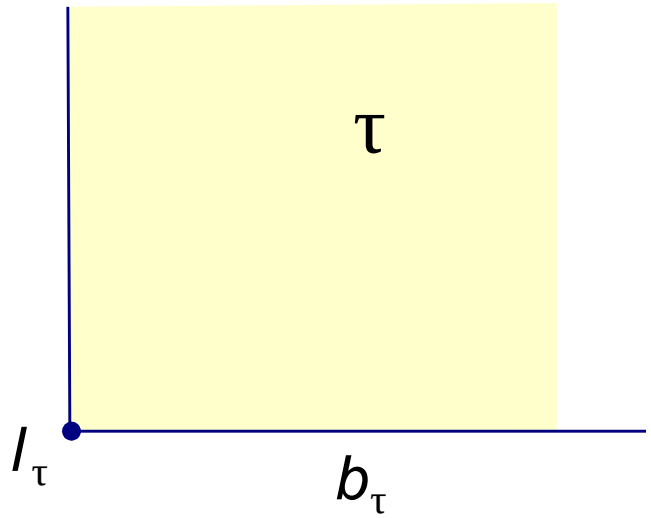


upper vertical extension
of original endpoint l_τ



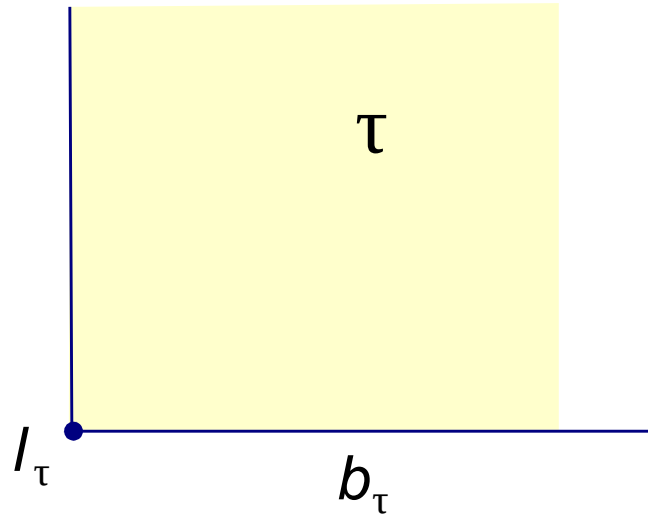
meeting point l_τ
of two original segments
degenerate case of ←

Classification based on left boundary



left wall of the bounding box
only one such trapezoid

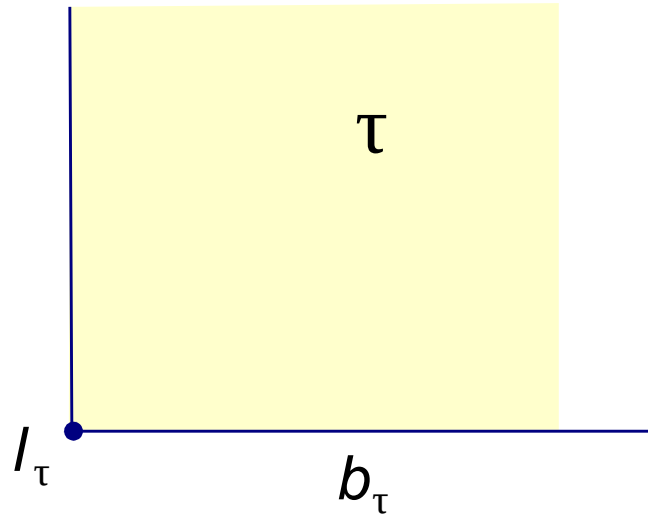
Classification based on left boundary



left wall of the bounding box

only one such trapezoid

Classification based on left boundary



left wall of the bounding box
only one such trapezoid

Refined subdivision: Finer analysis

- n original edges...
- $\rightarrow \quad \leq 2n$ original endpoints
- $\rightarrow \quad \leq 2 \times 2n$ additional vertices
- $+ \quad 4$ corners of the bounding box
- $\rightarrow \quad \leq 6n + 4$ vertices in total

Refined subdivision: Finer analysis

- n original edges...
- $\rightarrow \quad \leq 2n$ original endpoints
- $\rightarrow \quad \leq 2 \times 2n$ additional vertices
- $+ \quad 4$ corners of the bounding box
- $\rightarrow \quad \leq 6n + 4$ vertices in total

Refined subdivision: Finer analysis

- n original edges...
- $\rightarrow \quad \leq 2n$ original endpoints
- $\rightarrow \quad \leq 2 \times 2n$ additional vertices
- $+ \quad 4$ corners of the bounding box
- $\rightarrow \quad \leq 6n + 4$ vertices in total

Refined subdivision: Finer analysis

- n original edges...
- $\rightarrow \quad \leq 2n$ original endpoints
- $\rightarrow \quad \leq 2 \times 2n$ additional vertices
- $+ \quad 4$ corners of the bounding box
- $\rightarrow \quad \leq 6n + 4$ vertices in total

Refined subdivision: Finer analysis

- n original edges...
- $\rightarrow \quad \leq 2n$ original endpoints
- $\rightarrow \quad \leq 2 \times 2n$ additional vertices
- $+ \quad 4$ corners of the bounding box
- $\rightarrow \quad \leq 6n + 4$ vertices in total

Refined subdivision: Finer analysis

- n original edges...
- each original left endpoint is l_τ for at most two trapezoids
- each original right endpoint is l_τ for at most one trapezoid
- l_τ is the bottom-left corner of the bounding box for exactly one trapezoid
- l_τ is an original endpoint for all the other trapezoids
- $\rightarrow \quad \leq 3n+1$ trapezoids in total

Refined subdivision: Finer analysis

- n original edges...
- each original left endpoint is l_τ for at most two trapezoids
- each original right endpoint is l_τ for at most one trapezoid
- l_τ is the bottom-left corner of the bounding box for exactly one trapezoid
- l_τ is an original endpoint for all the other trapezoids
- $\rightarrow \quad \leq 3n+1$ trapezoids in total

Refined subdivision: Finer analysis

- n original edges...
- each original left endpoint is l_τ for at most two trapezoids
- each original right endpoint is l_τ for at most one trapezoid
- l_τ is the bottom-left corner of the bounding box for exactly one trapezoid
- l_τ is an original endpoint for all the other trapezoids
- $\rightarrow \quad \leq 3n+1$ trapezoids in total

Refined subdivision: Finer analysis

- n original edges...
- each original left endpoint is l_τ for at most two trapezoids
- each original right endpoint is l_τ for at most one trapezoid
- l_τ is the bottom-left corner of the bounding box for exactly one trapezoid
- l_τ is an original endpoint for all the other trapezoids
- $\rightarrow \quad \leq 3n+1$ trapezoids in total

Refined subdivision: Finer analysis

- n original edges...
- each original left endpoint is l_τ for at most two trapezoids
- each original right endpoint is l_τ for at most one trapezoid
- l_τ is the bottom-left corner of the bounding box for exactly one trapezoid
- l_τ is an original endpoint for all the other trapezoids
- $\rightarrow \quad \leq 3n+1$ trapezoids in total

Refined subdivision: Finer analysis

- n original edges...
- each original left endpoint is l_τ for at most two trapezoids
- each original right endpoint is l_τ for at most one trapezoid
- l_τ is the bottom-left corner of the bounding box for exactly one trapezoid
- l_τ is an original endpoint for all the other trapezoids
- $\rightarrow \quad \leq 3n+1$ trapezoids in total

Refined subdivision: Adjacencies

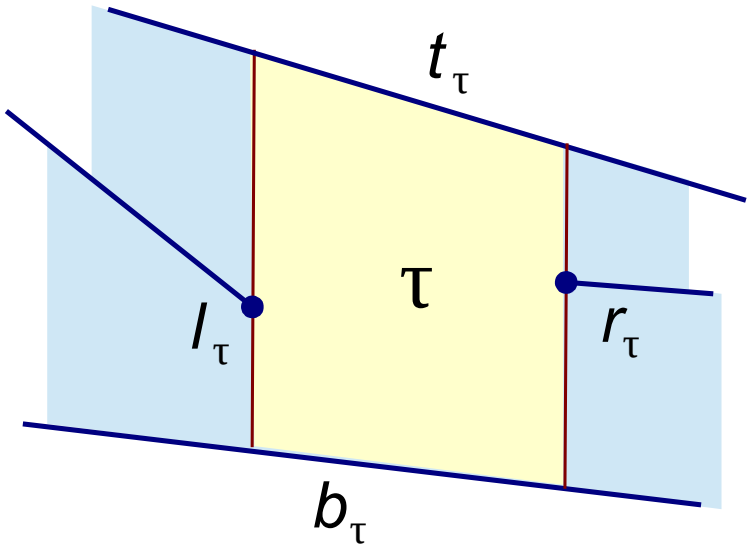
- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-left neighbor of τ : $b_{\tau'} = b_\tau$, $r_{\tau'} = l_\tau$
- τ' upper-left neighbor of τ : $t_{\tau'} = t_\tau$, $r_{\tau'} = l_\tau$
- and so on...
- This suggests to represent the map
  by a more specialized data structure than a DCEL

Refined subdivision: Adjacencies

- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-left neighbor of τ : $b_{\tau'} = b_\tau$, $r_{\tau'} = l_\tau$
- τ' upper-left neighbor of τ : $t_{\tau'} = t_\tau$, $r_{\tau'} = l_\tau$
- and so on...
- This suggests to represent the map by a more specialized data structure than a DCEL

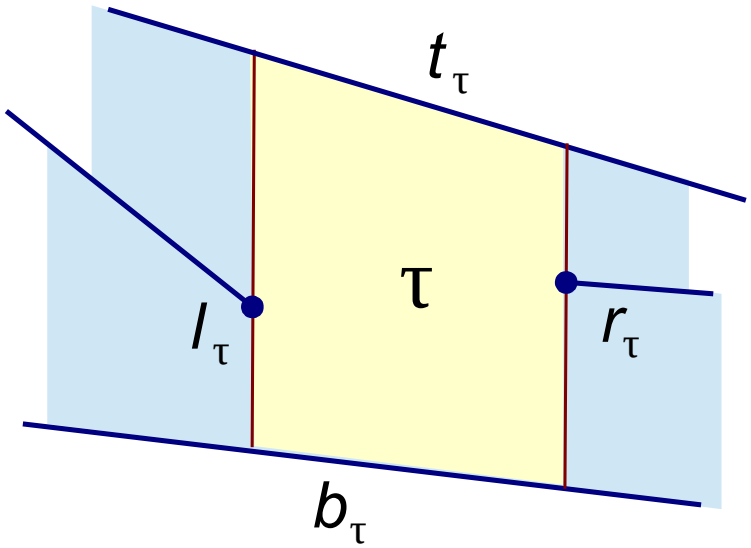
Refined subdivision: Adjacencies

- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
  - τ' lower-
  - τ' upper-
  - and so or
  - This sugg
    by a more



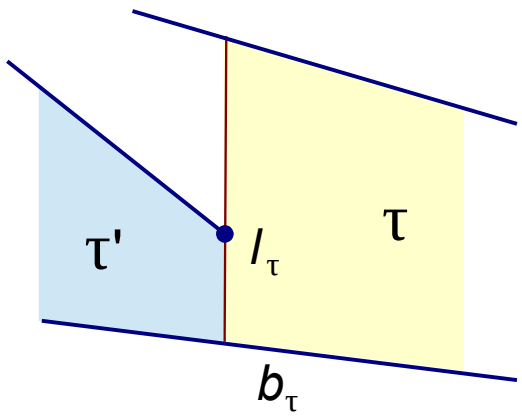
Refined subdivision: Adjacencies

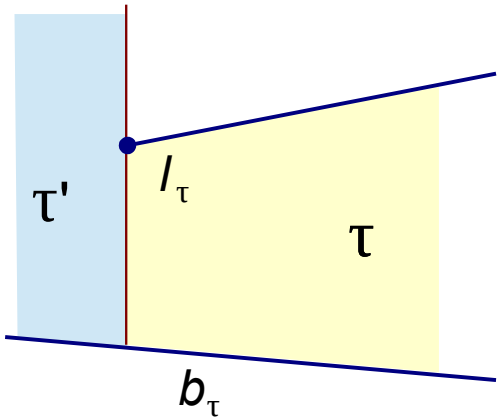
- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-
- τ' upper-
- and so or
- This sugg
  by a more



Refined subdivision: Adjacencies

- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-left neighbor of τ : $b_{\tau'} = b_\tau$, $r_{\tau'} = l_\tau$
- τ'
- an
- Th
by





Refined subdivision: Adjacencies

- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-left neighbor of τ : $b_{\tau'} = b_\tau$, $r_{\tau'} = l_\tau$
- τ' upper-left neighbor of τ : $t_{\tau'} = t_\tau$, $r_{\tau'} = l_\tau$
- and so on...
- This suggests to represent the map by a more specialized data structure than a DCEL

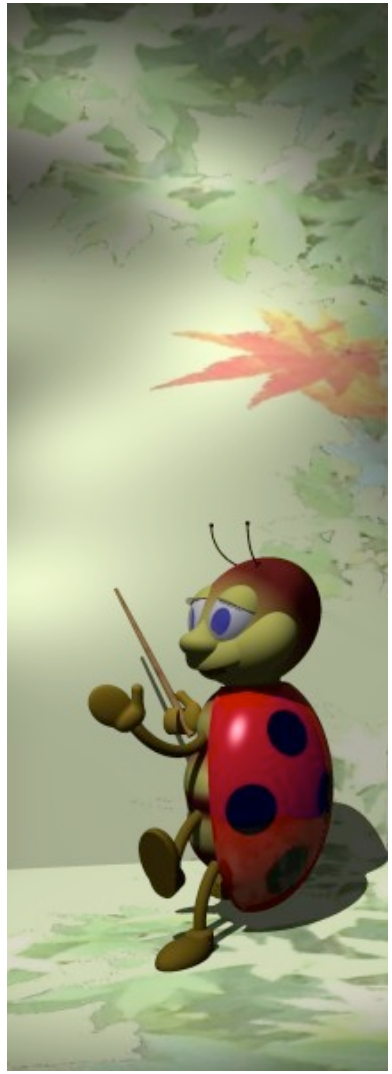
Refined subdivision: Adjacencies

- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-left neighbor of τ : $b_{\tau'} = b_\tau$, $r_{\tau'} = l_\tau$
- τ' upper-left neighbor of τ : $t_{\tau'} = t_\tau$, $r_{\tau'} = l_\tau$
- and so on...
- This suggests to represent the map by a more specialized data structure than a DCEL

Refined subdivision: Adjacencies

- τ, τ' *adjacent* if they share a vertical extension
- $\rightarrow$ same face of the original subdivision
- At most four adjacencies (*general position* assumption):
- τ' lower-left neighbor of τ : $b_{\tau'} = b_\tau$, $r_{\tau'} = l_\tau$
- τ' upper-left neighbor of τ : $t_{\tau'} = t_\tau$, $r_{\tau'} = l_\tau$
- and so on...
- This suggests to represent the map
  by a more specialized data structure than a DCEL

Outline



Plane sweep to build the map?

- Events: $O(n)$ vertices of the original subdivision S
- Sweep line: b_τ / t_τ of trapezoids being constructed
- Adjacency information: computed in $O(1)$ per trapezoid
- Efficient algorithm: $O(n \log n)$
- But what about *point location* costs?

Plane sweep to build the map?

- Events: $O(n)$ vertices of the original subdivision S
- Sweep line: b_τ / t_τ of trapezoids being constructed
- Adjacency information: computed in $O(1)$ per trapezoid
- Efficient algorithm: $O(n \log n)$
- But what about *point location* costs?

Plane sweep to build the map?

- Events: $O(n)$ vertices of the original subdivision S
- Sweep line: b_τ / t_τ of trapezoids being constructed
- Adjacency information: computed in $O(1)$ per trapezoid
- Efficient algorithm: $O(n \log n)$
- But what about *point location* costs?

Plane sweep to build the map?

- Events: $O(\ n\ )$ vertices of the original subdivision S
- Sweep line: b_τ / t_τ of trapezoids being constructed
- Adjacency information: computed in $O(\ 1\ )$ per trapezoid
- Efficient algorithm: $O(\ n \log n\ )$
- But what about *point location* costs?

Plane sweep to build the map?

- Events: $O(\ n\ )$ vertices of the original subdivision S
- Sweep line: b_τ / t_τ of trapezoids being constructed
- Adjacency information: computed in $O(\ 1\ )$ per trapezoid
- Efficient algorithm: $O(\ n \log n\ )$
- But what about *point location* costs?

Point-location structure

- Directed Acyclic Graph (DAG)
- Just one *root*
- One leaf for each trapezoid of the map
- Non-leaf nodes have *out-degree* 2 (two “children”)
- Two types of non-leaf nodes: x-nodes and y-nodes

Point-location structure

- Directed Acyclic Graph (DAG)
- Just one *root*
- One leaf for each trapezoid of the map
- Non-leaf nodes have *out-degree* 2 (two “children”)
- Two types of non-leaf nodes: x-nodes and y-nodes

Point-location structure

- Directed Acyclic Graph (DAG)
- Just one *root*
- One leaf for each trapezoid of the map
- Non-leaf nodes have *out-degree* 2 (two “children”)
- Two types of non-leaf nodes: x-nodes and y-nodes

Point-location structure

- Directed Acyclic Graph (DAG)
- Just one *root*
- One leaf for each trapezoid of the map
- Non-leaf nodes have *out-degree* 2 (two “children”)
- Two types of non-leaf nodes: x-nodes and y-nodes

Point-location structure

- Directed Acyclic Graph (DAG)
- Just one *root*
- One leaf for each trapezoid of the map
- Non-leaf nodes have *out-degree* 2 (two “children”)
- Two types of non-leaf nodes: x-nodes and y-nodes

DAG's nodes

- x-Node ν is connected with vertex $v_\nu \in S$
- y-Node ν is connected with edge $e_\nu \in S$
- Leaf node ν represents trapezoid τ_ν of the map, i.e. a final destination of the search

DAG's nodes

- x-Node ν is connected with vertex $v_\nu \in S$
- y-Node ν is connected with edge $e_\nu \in S$
- Leaf node ν represents trapezoid τ_ν of the map, i.e. a final destination of the search

DAG's nodes

- x-Node ν is connected with vertex $v_\nu \in S$
- y-Node ν is connected with edge $e_\nu \in S$
- Leaf node ν represents trapezoid τ_ν of the map, i.e. a final destination of the search

Point-location logic

- Query point: q
- Starting from the *root*...
- At x-node ν test if q is to the left/right of v_ν and move to ν's corresponding child
- At y-node ν test if q is below/above e_ν and move to ν's corresponding child
- At leaf node ν we know that q lies in τ_ν
  (for the sake of simplicity assume that q lies strictly inside a trapezium)

Point-location logic

- Query point: q
- Starting from the *root*...
- At x-node ν test if q is to the left/right of v_ν and move to ν's corresponding child
- At y-node ν test if q is below/above e_ν and move to ν's corresponding child
- At leaf node ν we know that q lies in τ_ν (for the sake of simplicity assume that q lies strictly inside a trapezium)

Point-location logic

- Query point: q
- Starting from the *root*...
- At x-node ν test if q is to the left/right of v_ν and move to ν's corresponding child
- At y-node ν test if q is below/above e_ν and move to ν's corresponding child
- At leaf node ν we know that q lies in τ_ν (for the sake of simplicity assume that q lies strictly inside a trapezium)

Point-location logic

- Query point: q
- Starting from the *root*...
- At x-node ν test if q is to the left/right of v_ν and move to ν's corresponding child
- At y-node ν test if q is below/above e_ν and move to ν's corresponding child
- At leaf node ν we know that q lies in τ_ν
  (for the sake of simplicity assume that q lies strictly inside a trapezium)

Point-location logic

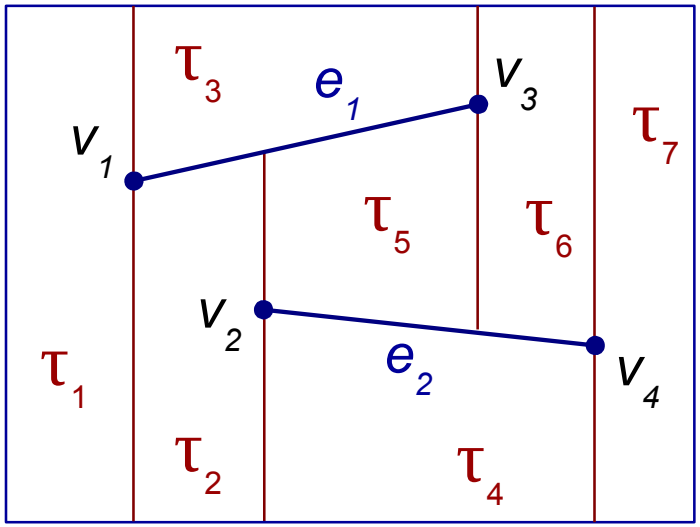
- Query point: q
- Starting from the *root*...
- At x-node ν test if q is to the left/right of v_ν and move to ν's corresponding child
- At y-node ν test if q is below/above e_ν and move to ν's corresponding child
- At leaf node ν we know that q lies in τ_ν

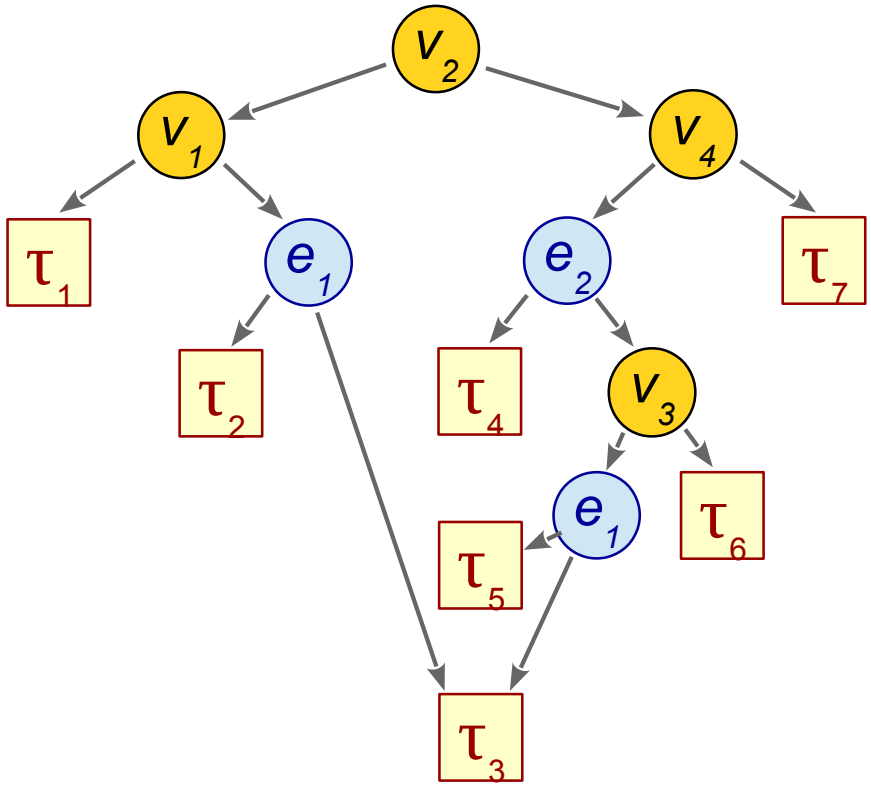
  (for the sake of simplicity assume that q lies strictly inside a trapezium)

Point-location logic

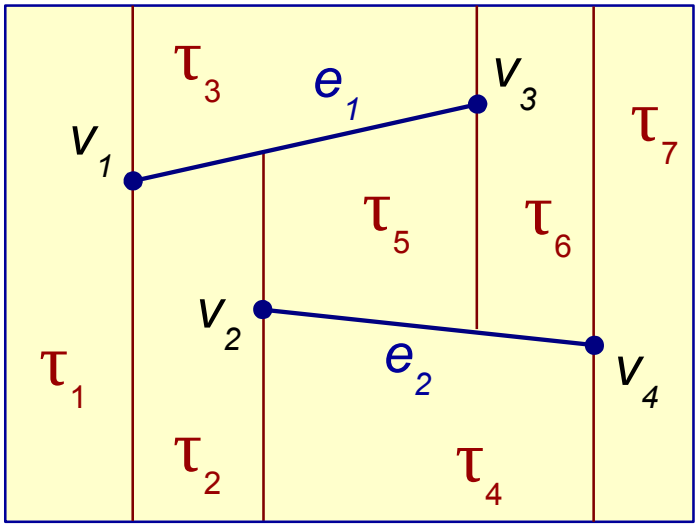
- Query point: q
- Starting from the *root*...
- At x-node ν test if q is to the left/right of v_ν and move to ν's corresponding child
- At y-node ν test if q is below/above e_ν and move to ν's corresponding child
- At leaf node ν we know that q lies in τ_ν
  (for the sake of simplicity assume that q lies strictly inside a trapezium)

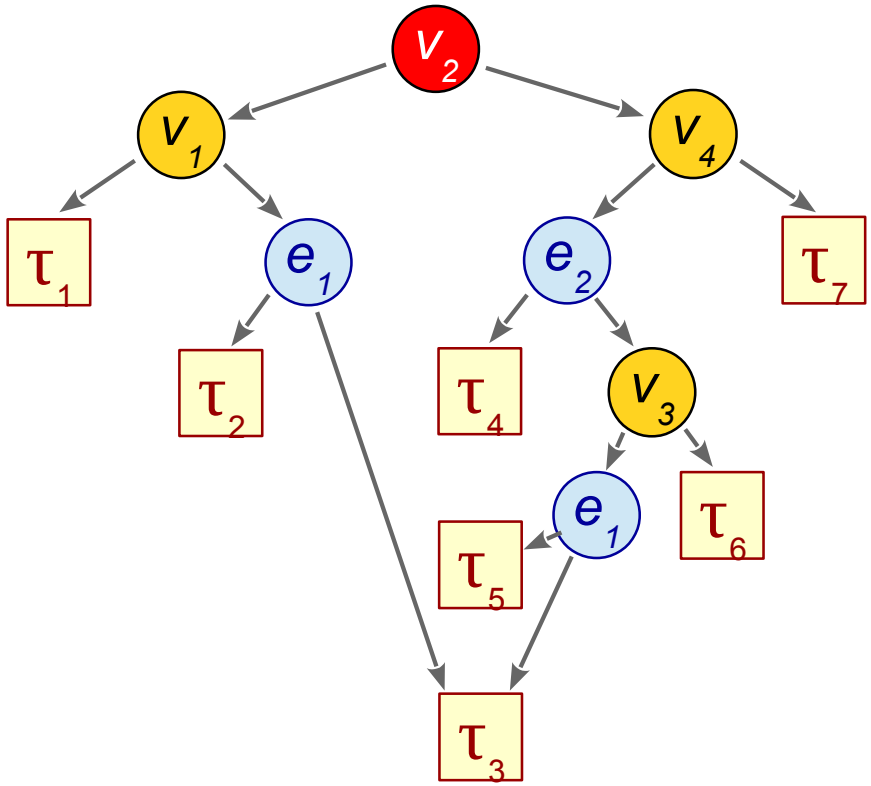
Trapezoidal map and related DAG



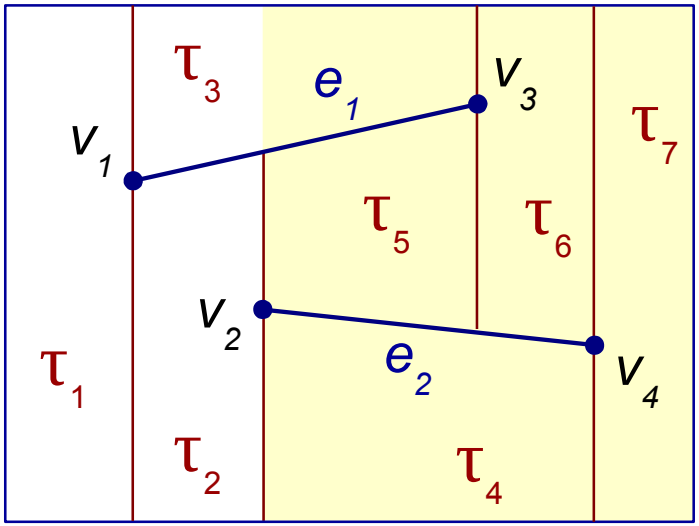


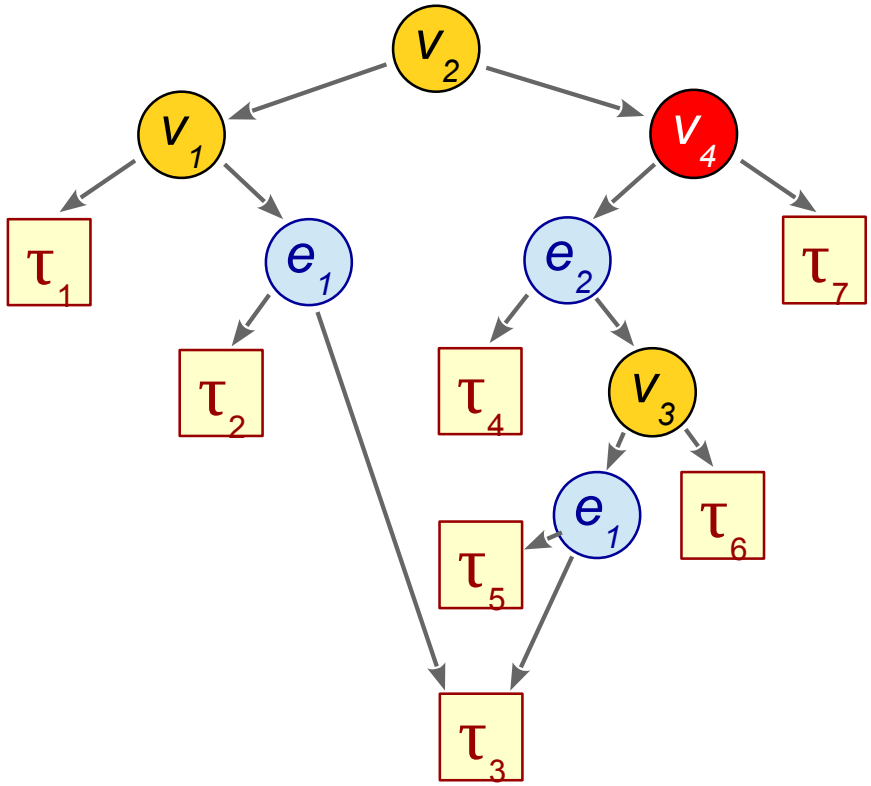
Search through the DAG



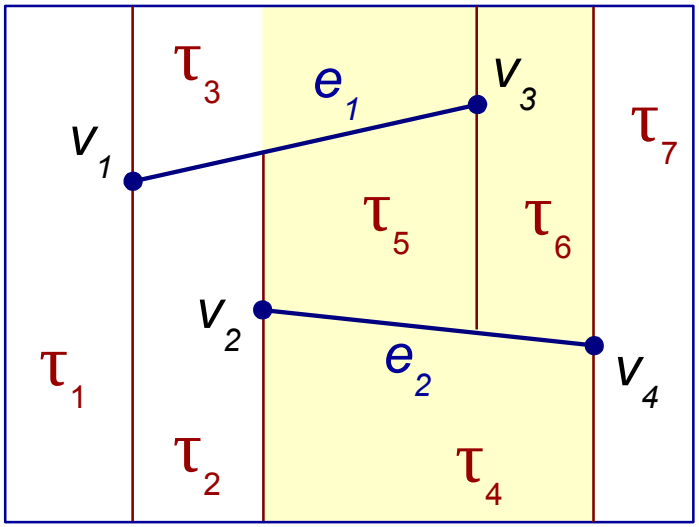


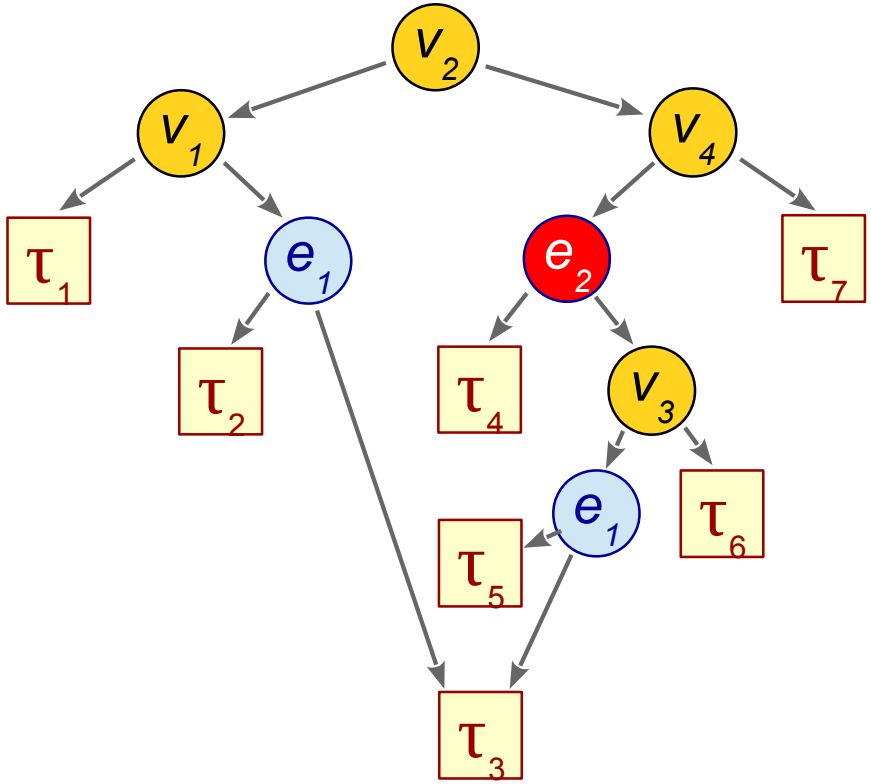
Search through the DAG



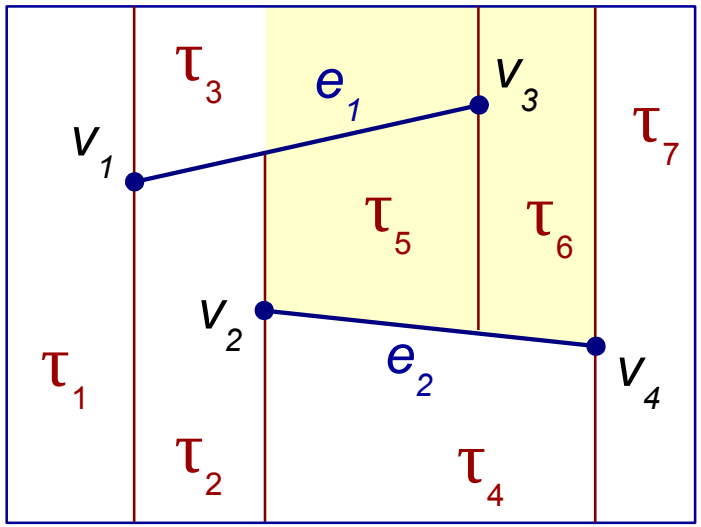


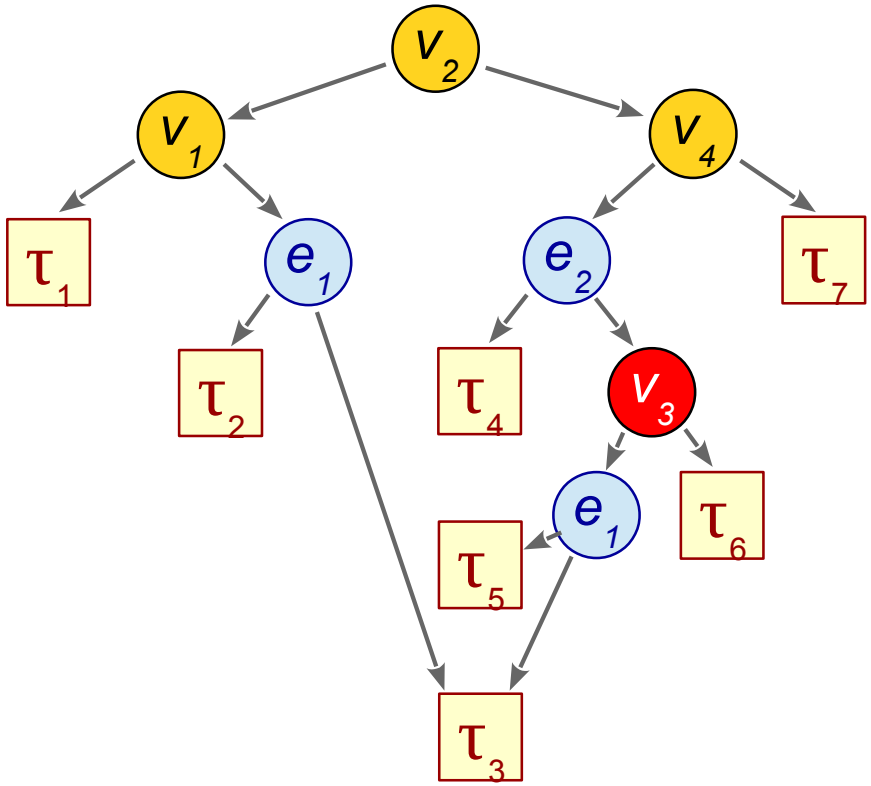
Search through the DAG



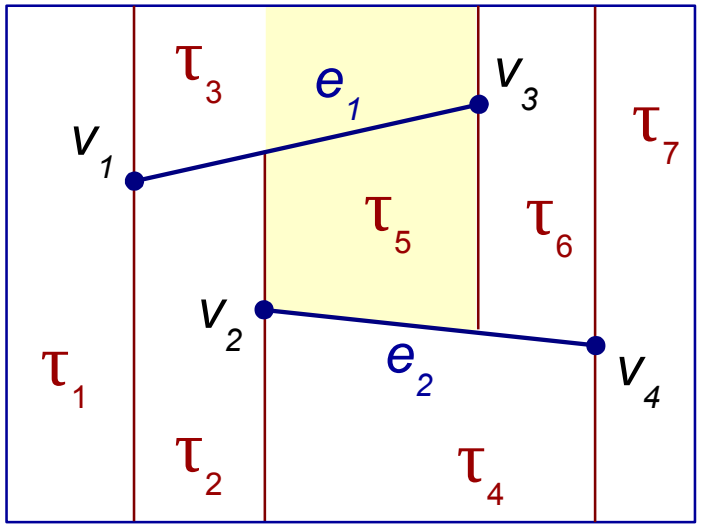


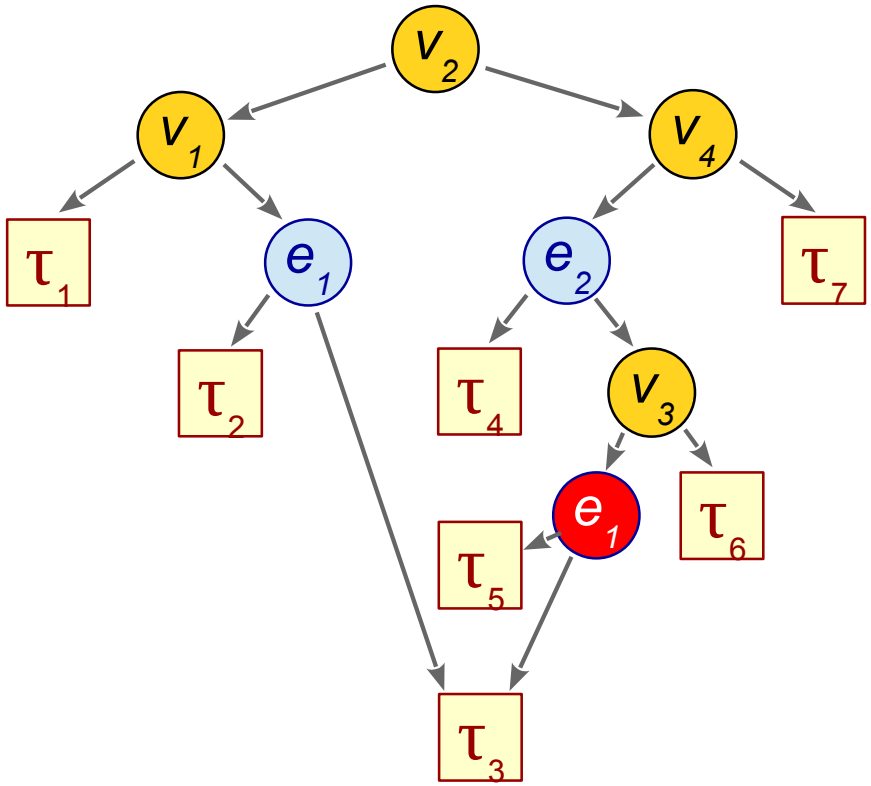
Search through the DAG



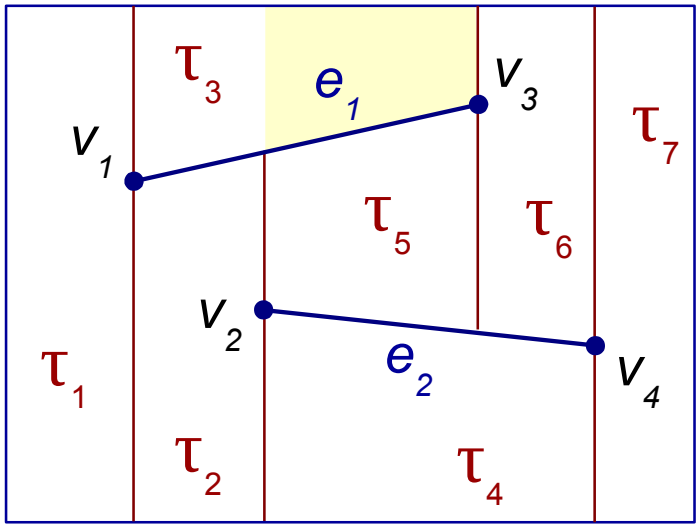


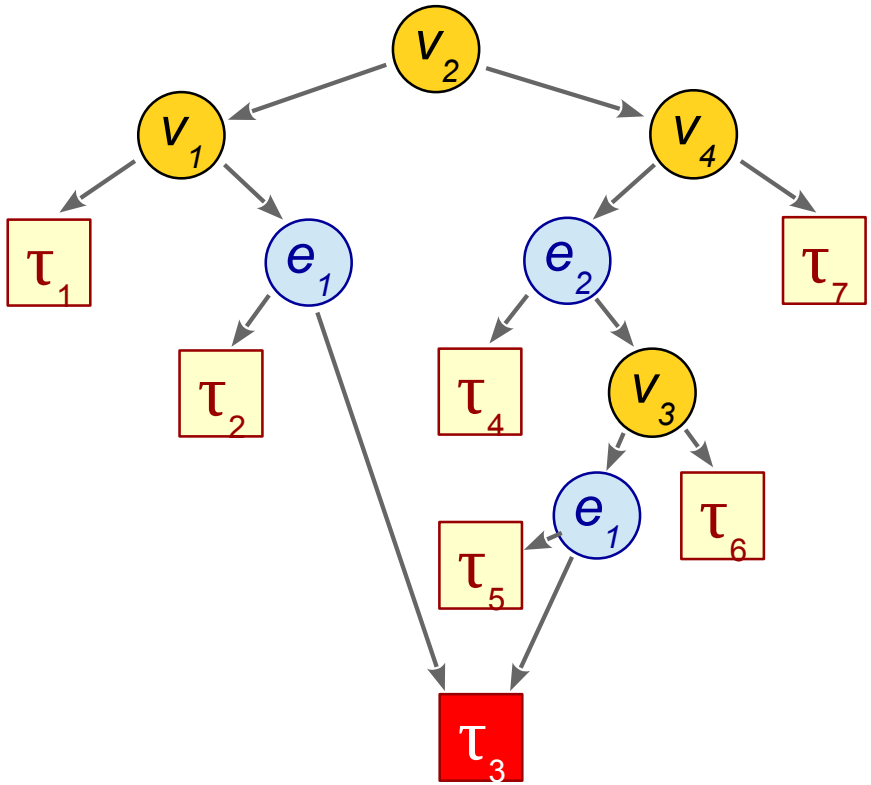
Search through the DAG





Search through the DAG





Randomized incremental algorithm

- Edges are added one at a time
- The map and the DAG are incrementally updated to represent the trapezoidal map of the added edges
- The “efficiency” of the search structure (DAG) depends on the order in which edges are added
- Randomization ensures good expected performance

Randomized incremental algorithm

- Edges are added one at a time
- The map and the DAG are incrementally updated to represent the trapezoidal map of the added edges
- The “efficiency” of the search structure (DAG) depends on the order in which edges are added
- Randomization ensures good expected performance

Randomized incremental algorithm

- Edges are added one at a time
- The map and the DAG are incrementally updated to represent the trapezoidal map of the added edges
- The “efficiency” of the search structure (DAG) depends on the order in which edges are added
- Randomization ensures good expected performance

Randomized incremental algorithm

- Edges are added one at a time
- The map and the DAG are incrementally updated to represent the trapezoidal map of the added edges
- The “efficiency” of the search structure (DAG) depends on the order in which edges are added
- Randomization ensures good expected performance

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Algorithm steps

- Initially the map contains only the *bounding box*
- $\rightarrow$ one-node DAG
- For each edge $e \in S$ in randomized order...
  - remove the trapezoids $\tau_1, \tau_2, \ldots \tau_k$ in *conflict* with e
  - replace them with the new trapezoids determined by e
  - remove the DAG's leaves linked to $\tau_1, \tau_2, \ldots \tau_k$
  - replace these leaves with x-/y-nodes as appropriate
  - create and link leaves for the new trapezoids

Finding trapezoids in conflict with a new edge

- Point location of e's left endpoint (current DAG)
- $\rightarrow$ leftmost trapezoid τ_1 in conflict with e
- Follow right-neighbor links from τ_1 to the trapezoid τ_k which contains e's right endpoint (edges do not cross)
- The correct neighbor τ_{i+1} of τ_i is identified by testing where r_{τ_i} lies relative to e

Finding trapezoids in conflict with a new edge

- Point location of e's left endpoint (current DAG)
- $\rightarrow$ leftmost trapezoid τ_1 in conflict with e
- Follow right-neighbor links from τ_1 to the trapezoid τ_k which contains e's right endpoint (edges do not cross)
- The correct neighbor τ_{i+1} of τ_i is identified by testing where r_{τ_i} lies relative to e

Finding trapezoids in conflict with a new edge

- Point location of e's left endpoint (current DAG)
- $\rightarrow$ leftmost trapezoid τ_1 in conflict with e
- Follow right-neighbor links from τ_1 to the trapezoid τ_k which contains e's right endpoint (edges do not cross)
- The correct neighbor τ_{i+1} of τ_i is identified by testing where r_{τ_i} lies relative to e

Finding trapezoids in conflict with a new edge

- Point location of e's left endpoint (current DAG)
- $\rightarrow$ leftmost trapezoid τ_1 in conflict with e
- Follow right-neighbor links from τ_1 to the trapezoid τ_k which contains e's right endpoint (edges do not cross)
- The correct neighbor τ_{i+1} of τ_i is identified by testing where r_{τ_i} lies relative to e

Updating the map

- τ_1 and τ_k are partitioned in three parts (four if $\tau_1 = \tau_k$)
- $\tau_2, \tau_3, \ldots \tau_{k-1}$ are split
- Whenever possible, the resulting trapezoids bounded by e are merged
- All operations can be done in $O(\ k\ )$ (in constant time for each involved trapezoid)

Updating the map

- τ_1 and τ_k are partitioned in three parts (four if $\tau_1 = \tau_k$)
- $\tau_2, \tau_3, \ldots \tau_{k-1}$ are split
- Whenever possible, the resulting trapezoids bounded by e are merged
- All operations can be done in $O(k)$ (in constant time for each involved trapezoid)

Updating the map

- τ_1 and τ_k are partitioned in three parts (four if $\tau_1 = \tau_k$)
- $\tau_2, \tau_3, \ldots \tau_{k-1}$ are split
- Whenever possible, the resulting trapezoids bounded by e are merged
- All operations can be done in $O(k)$ (in constant time for each involved trapezoid)

Updating the map

- τ_1 and τ_k are partitioned in three parts (four if $\tau_1 = \tau_k$)
- $\tau_2, \tau_3, \ldots \tau_{k-1}$ are split
- Whenever possible, the resulting trapezoids bounded by e are merged
- All operations can be done in $O(\ k\ )$ (in constant time for each involved trapezoid)

Updating the DAG

- Cross links between leaf nodes and trapezoids
- At most three new x-/y-nodes for each removed trapezoid
- Several nodes are linked to a new “merged” trapezoid
- All arrangements can be done in $O(\ k\ )$

Updating the DAG

- Cross links between leaf nodes and trapezoids
- At most three new x-/y-nodes for each removed trapezoid
- Several nodes are linked to a new “merged” trapezoid
- All arrangements can be done in $O(\ k\ )$

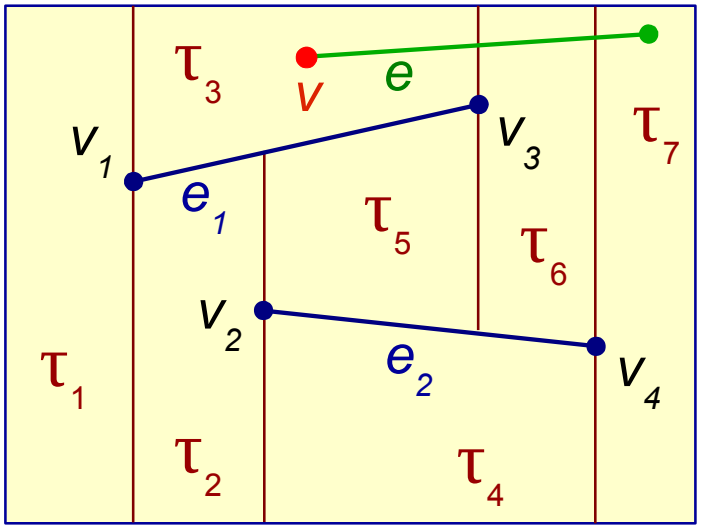
Updating the DAG

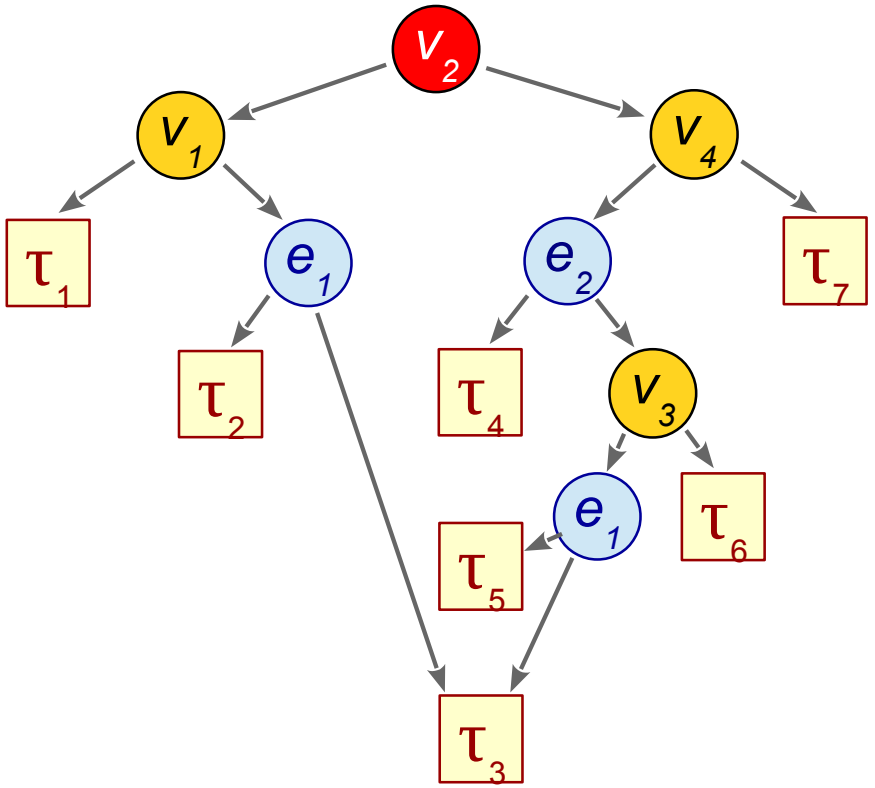
- Cross links between leaf nodes and trapezoids
- At most three new x-/y-nodes for each removed trapezoid
- Several nodes are linked to a new “merged” trapezoid
- All arrangements can be done in $O(\ k\ )$

Updating the DAG

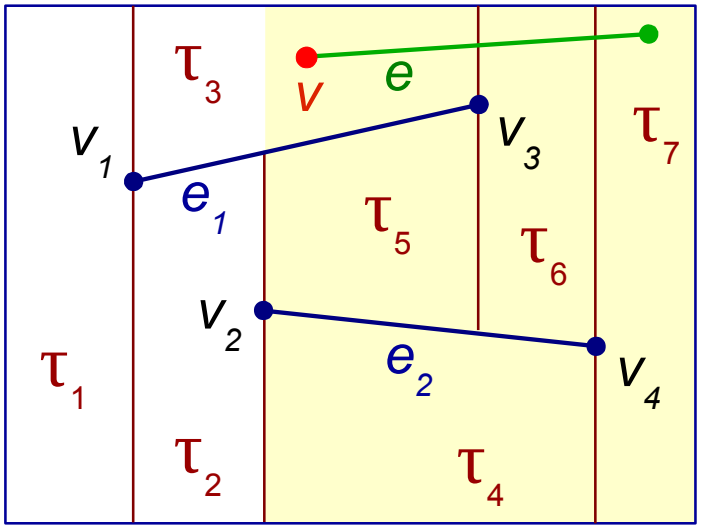
- Cross links between leaf nodes and trapezoids
- At most three new x-/y-nodes for each removed trapezoid
- Several nodes are linked to a new “merged” trapezoid
- All arrangements can be done in $O(\ k\ )$

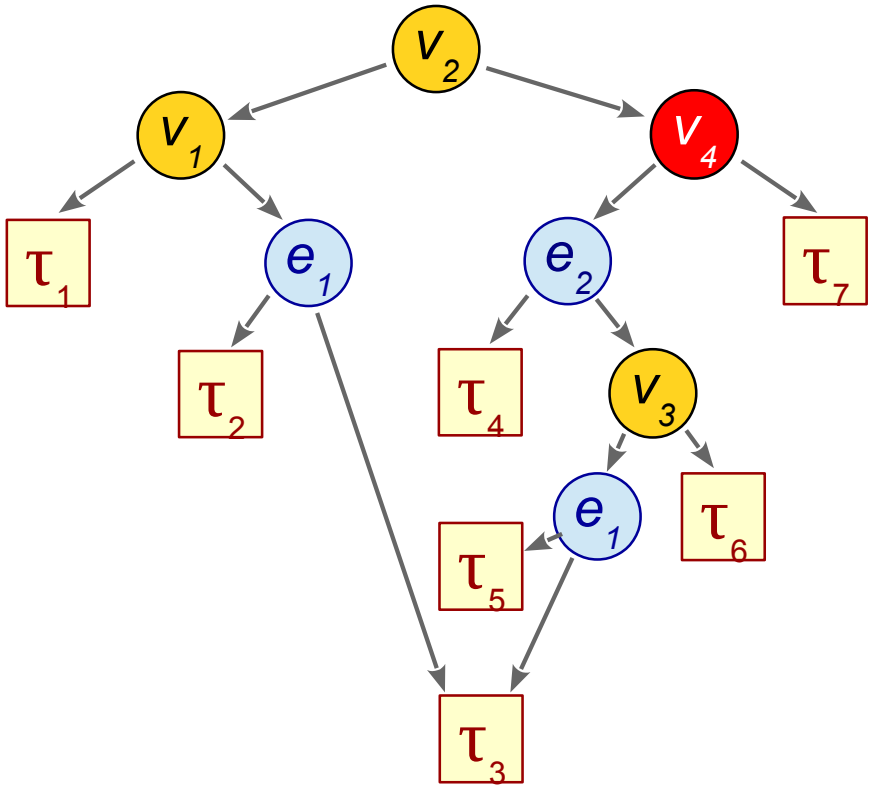
In summary: Locate leftmost endpoint of new edge...



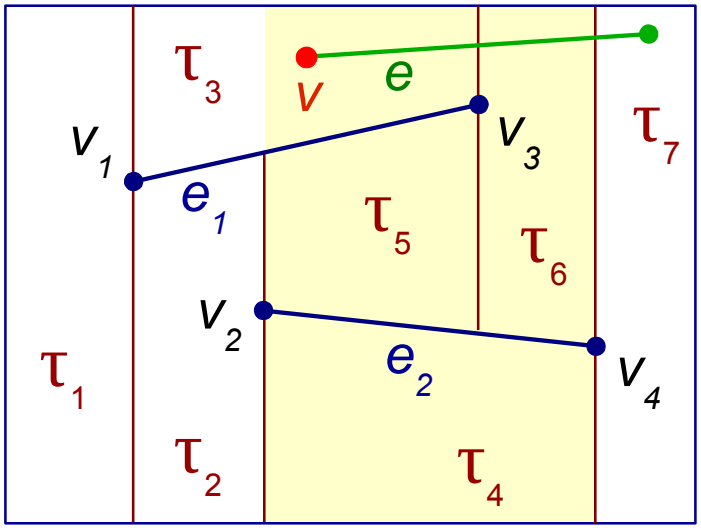


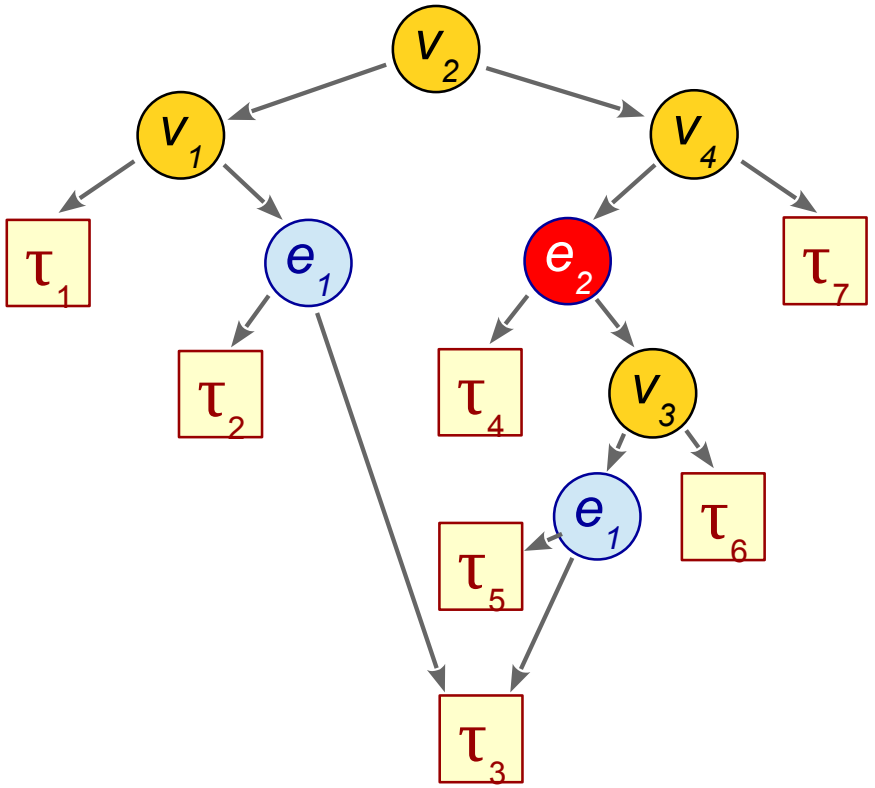
In summary: . . . by stepping down the DAG



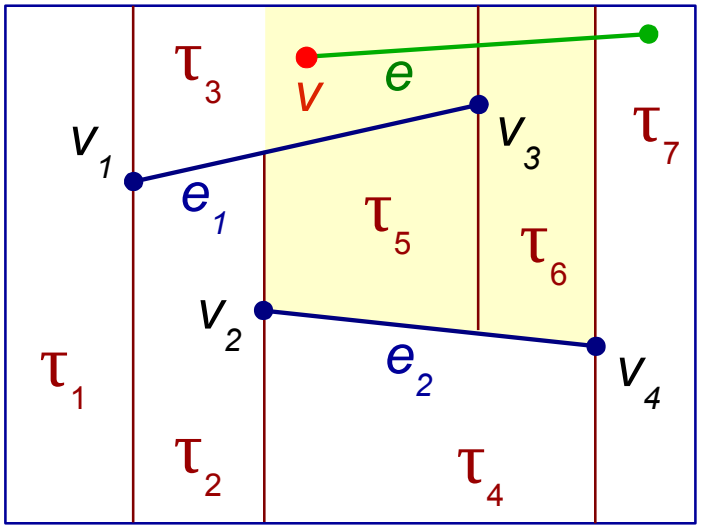


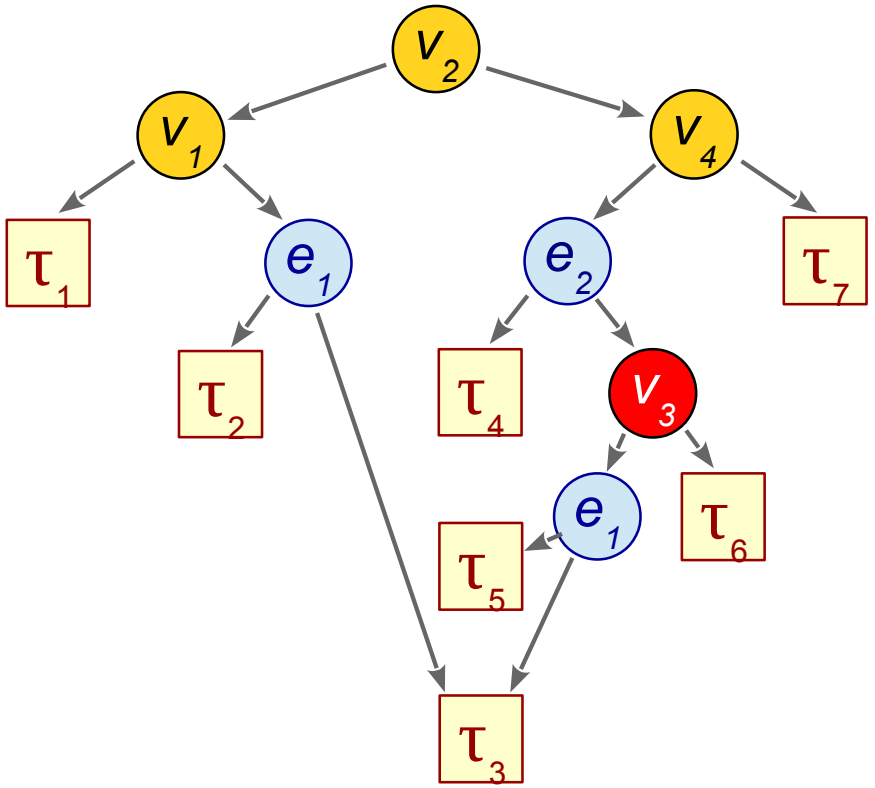
In summary: . . . by stepping down the DAG



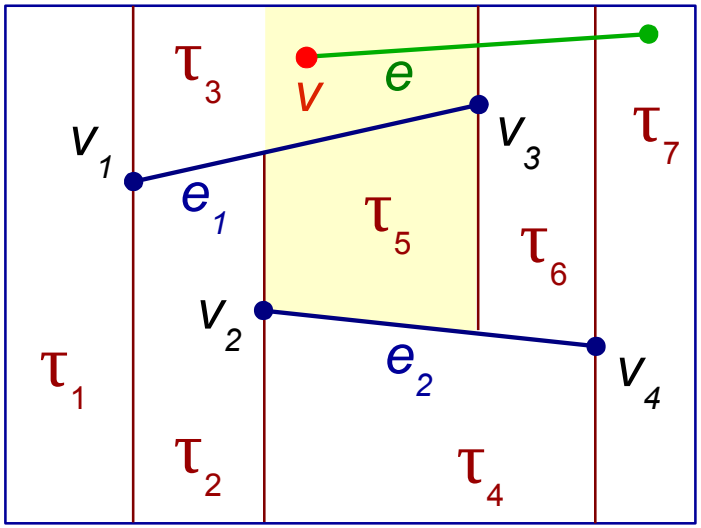


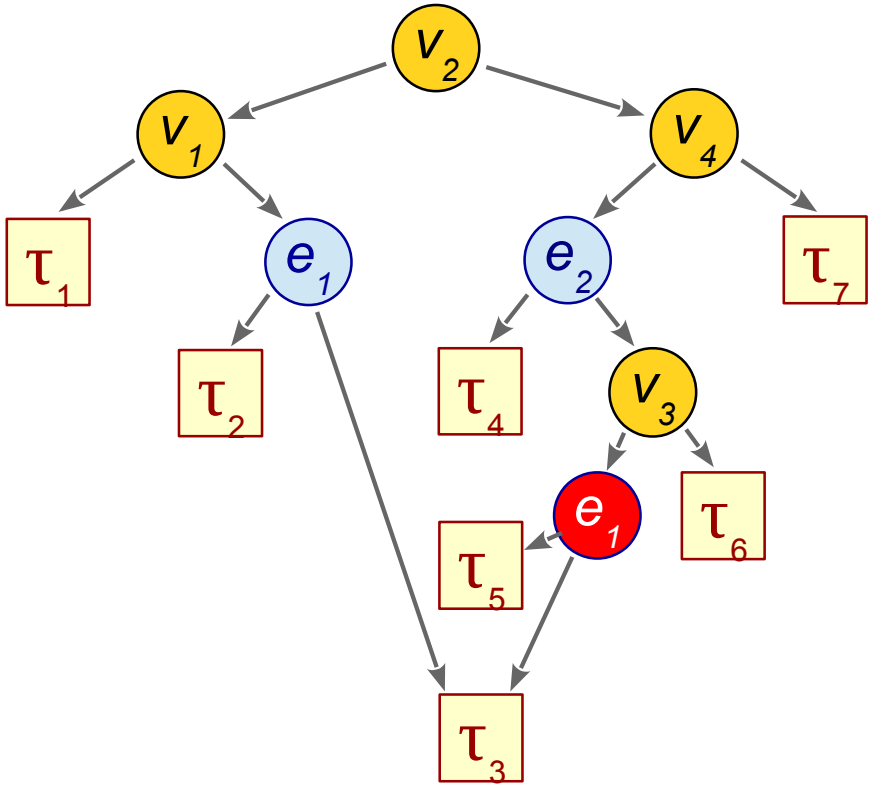
In summary: . . . by stepping down the DAG



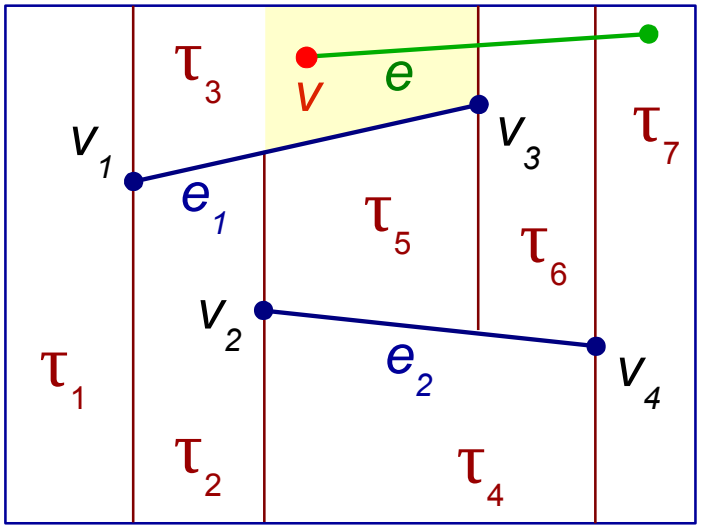


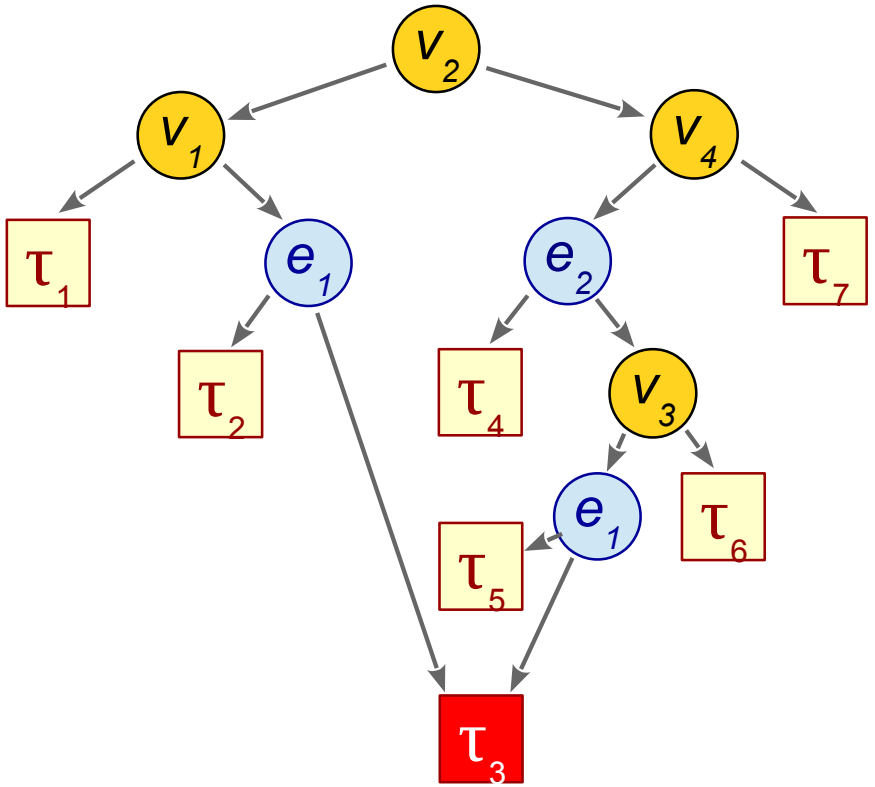
In summary: . . . by stepping down the DAG



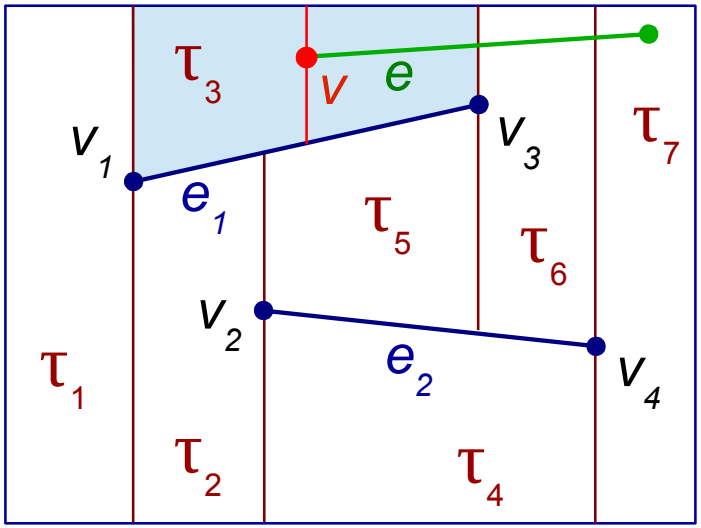


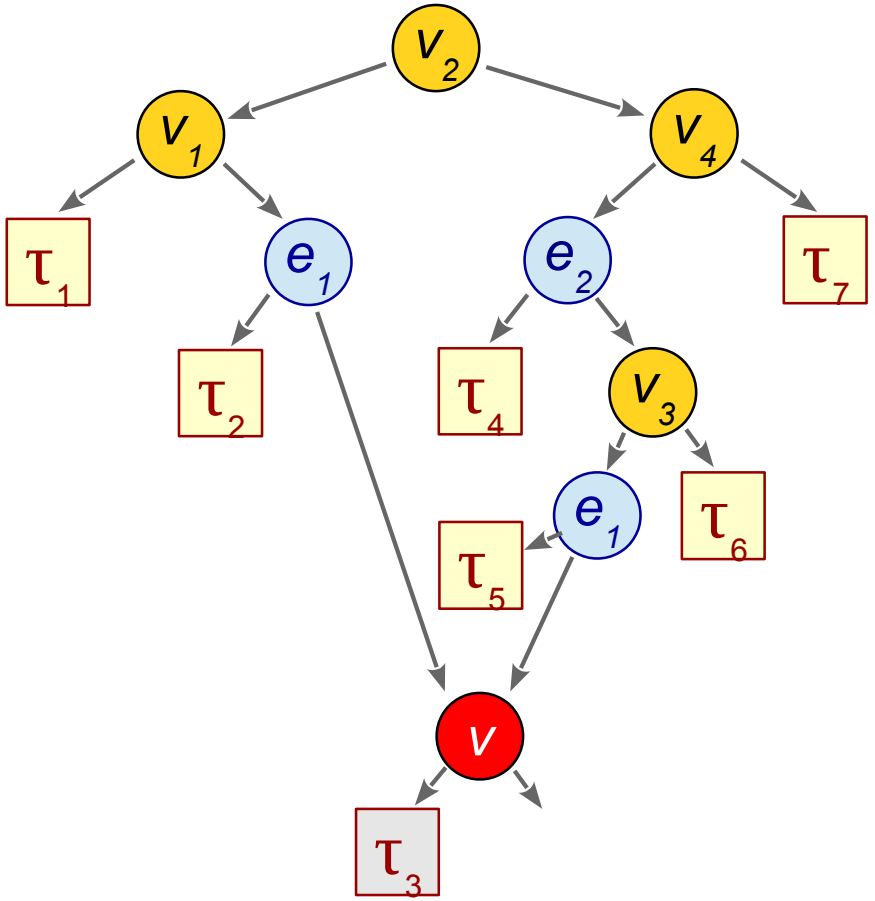
In summary: Start from leftmost trapezoid in conflict



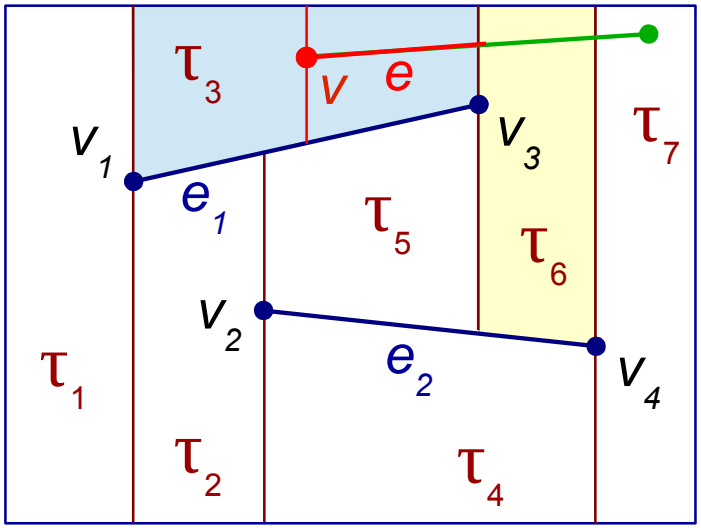


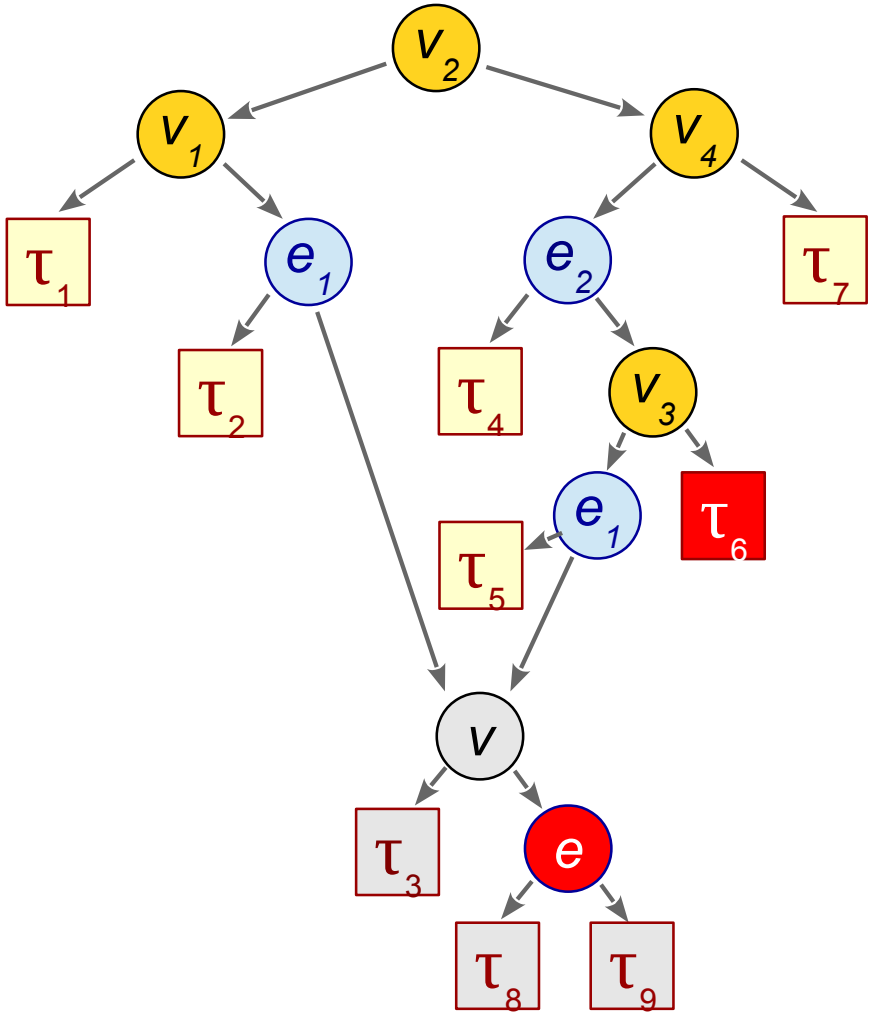
In summary: Update trapezoid...



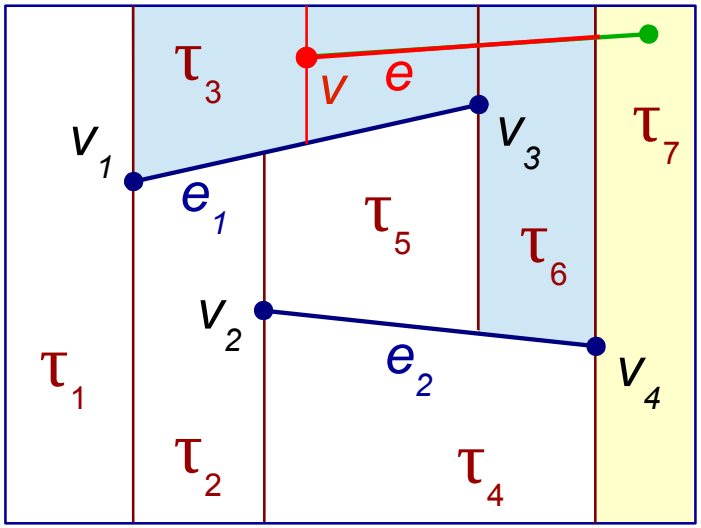


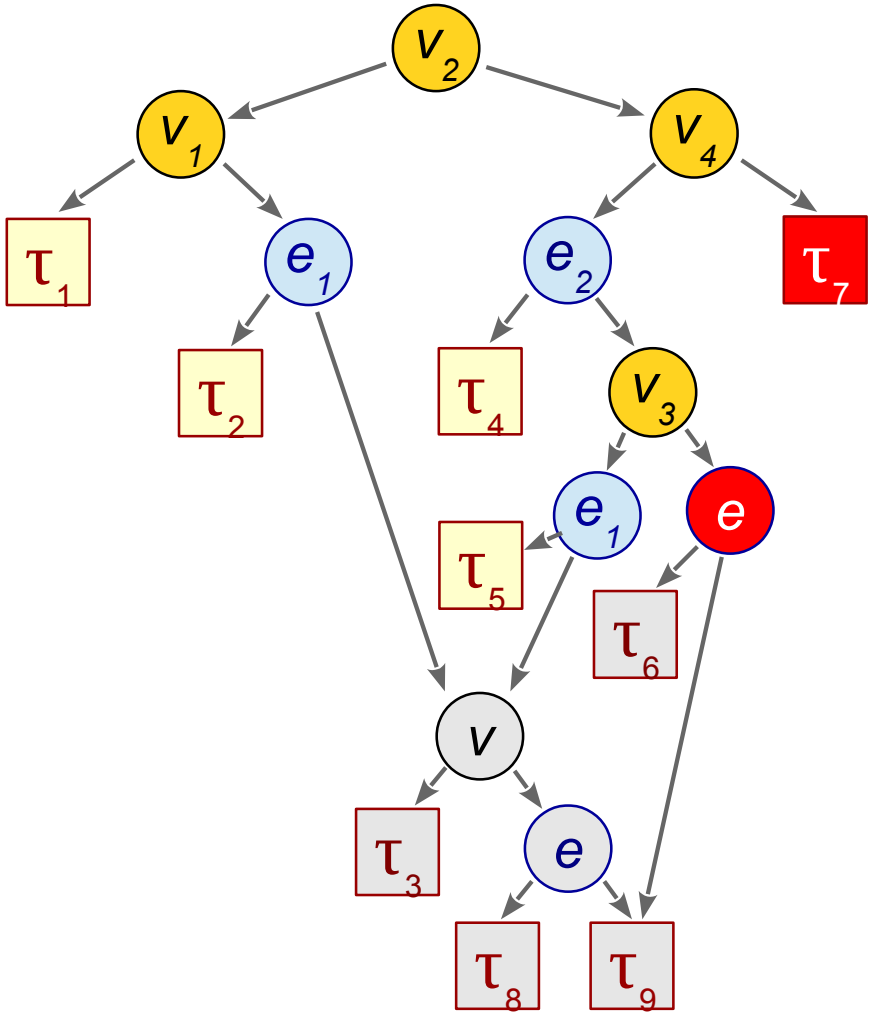
In summary: . . . and walk along edge



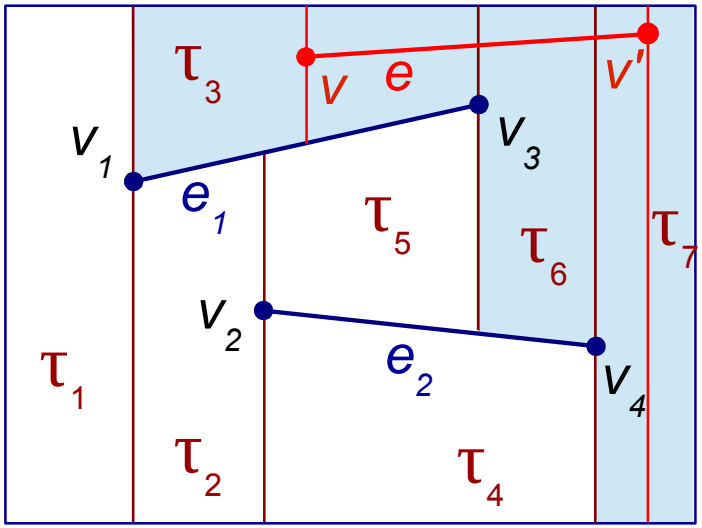


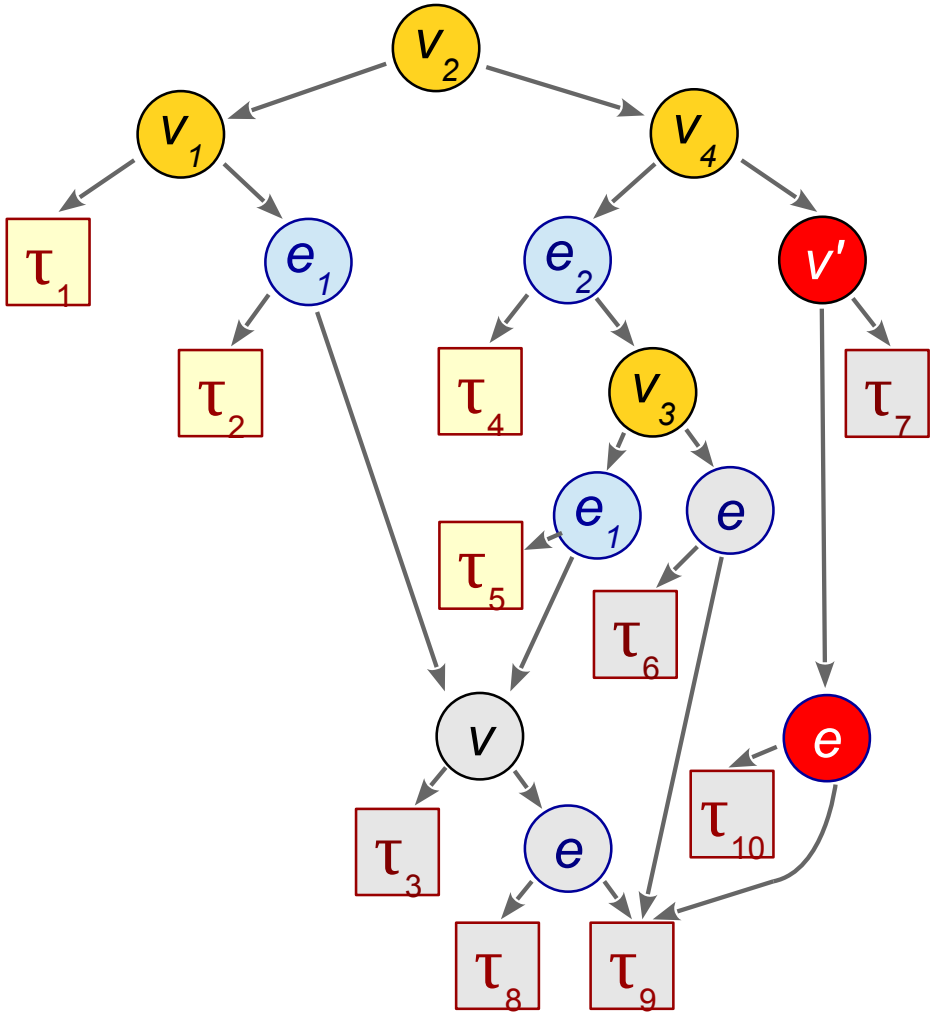
In summary: Split & merge new trapezoids...



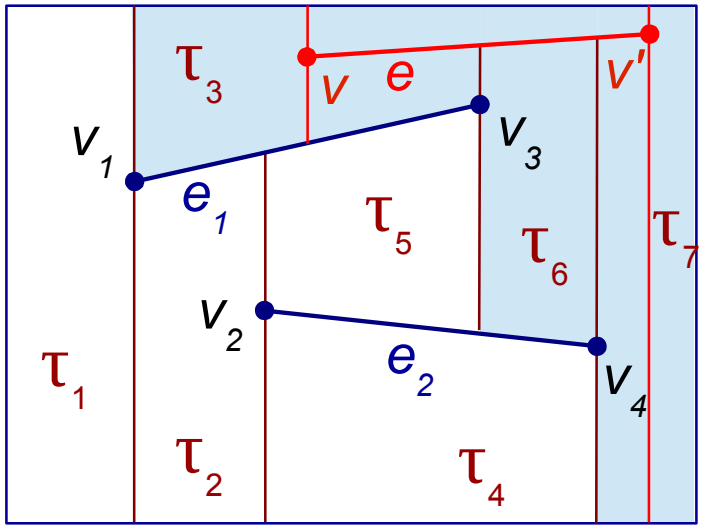


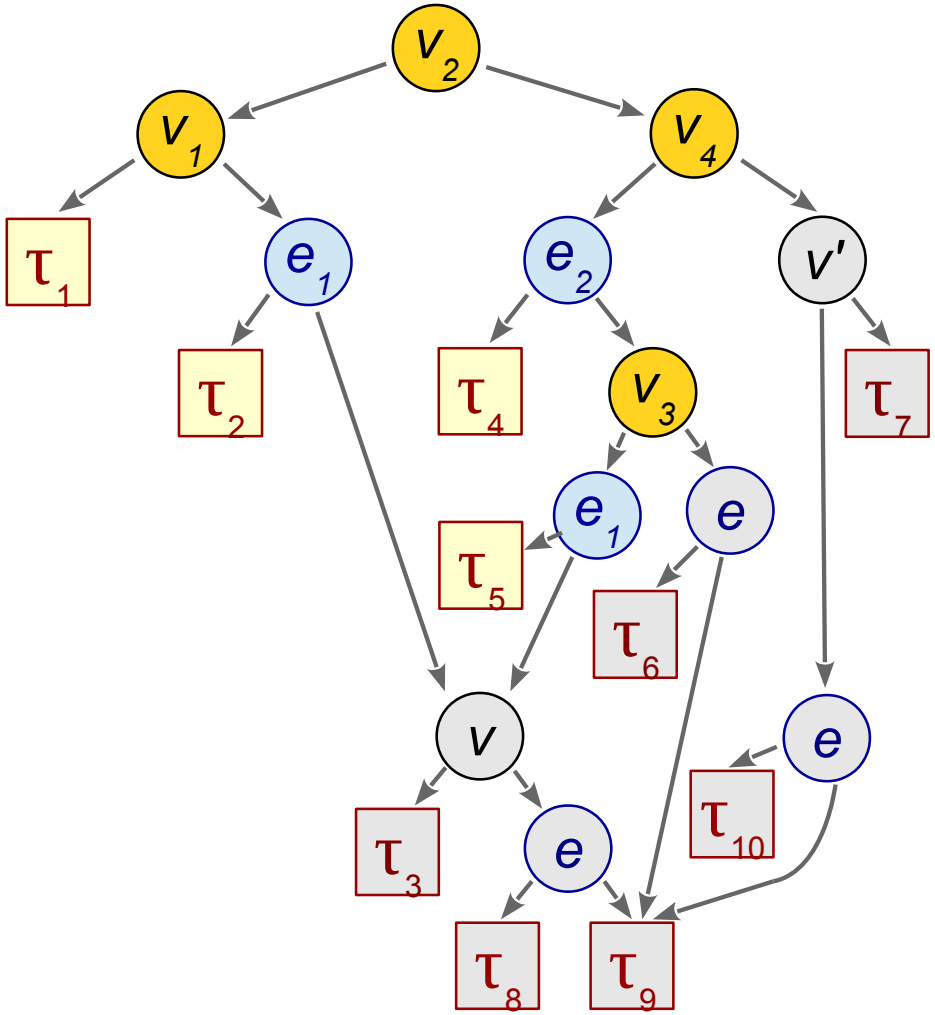
In summary: . . . up to the rightmost endpoint



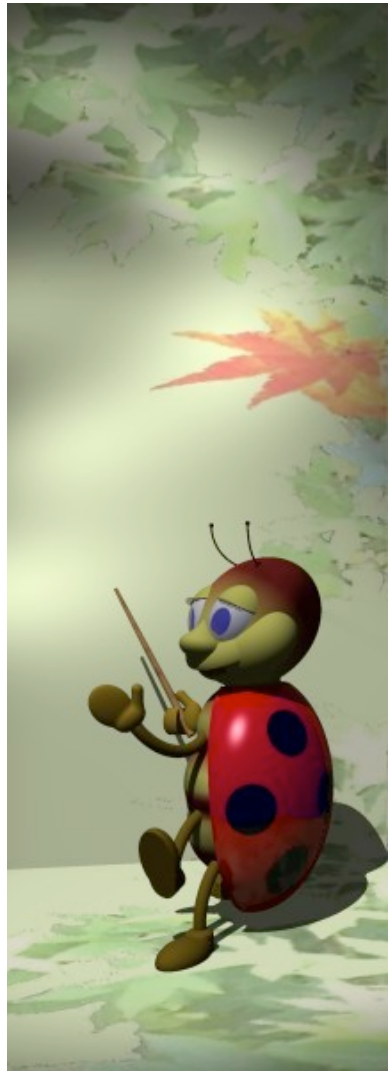


In summary: At the end Map and DAG are updated





Outline



Point location costs

- For given planar subdivision S and query point q
- Follow q's point location path π through the DAG
- Reflecting its construction steps:

$$S_0 = B,\ S_1,\ S_2,\ \ldots\ S_n = S$$

- N_i = number of nodes created on π at step $i \in [1, n]$

Point location costs

- For given planar subdivision S and query point q
- Follow q's point location path π through the DAG
- Reflecting its construction steps:

$$S_0 = B,\ S_1,\ S_2,\ \ldots\ S_n = S$$

- N_i = number of nodes created on π at step $i \in [1, n]$

Point location costs

- For given planar subdivision S and query point q
- Follow q's point location path π through the DAG
- Reflecting its construction steps:

$$S_0 = B, \; S_1, \; S_2, \; \ldots \; S_n = S$$

- N_i = number of nodes created on π at step $i \in [1, n]$

Point location costs

- For given planar subdivision S and query point q
- Follow q's point location path π through the DAG
- Reflecting its construction steps:

$$S_0 = B, \; S_1, \; S_2, \; \ldots \; S_n = S$$

- N_i = number of nodes created on π at step $i \in [1, n]$

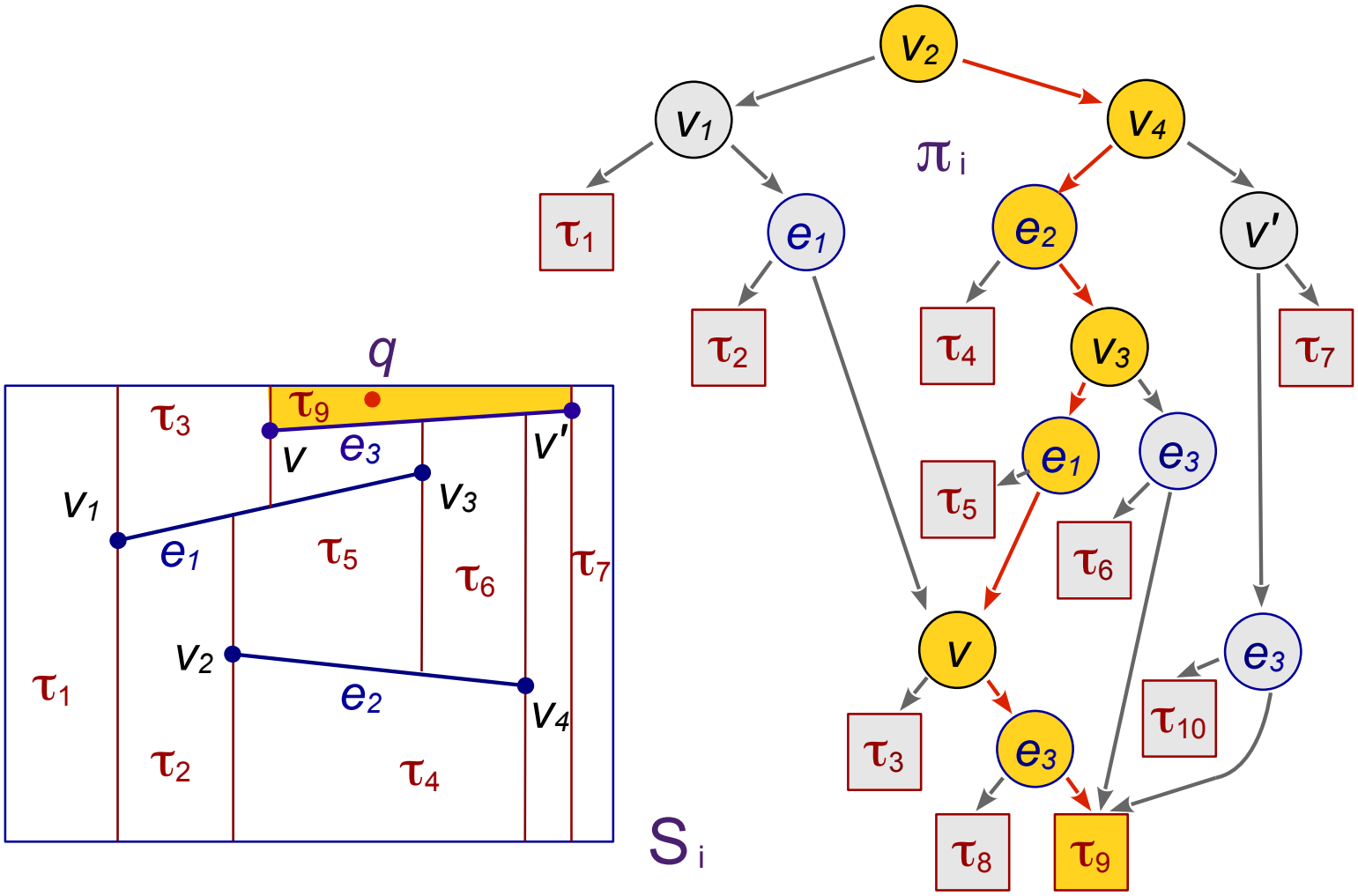
Point location costs

- For given planar subdivision S and query point q
- Follow q's point location path π through the DAG
- Reflecting its construction steps:

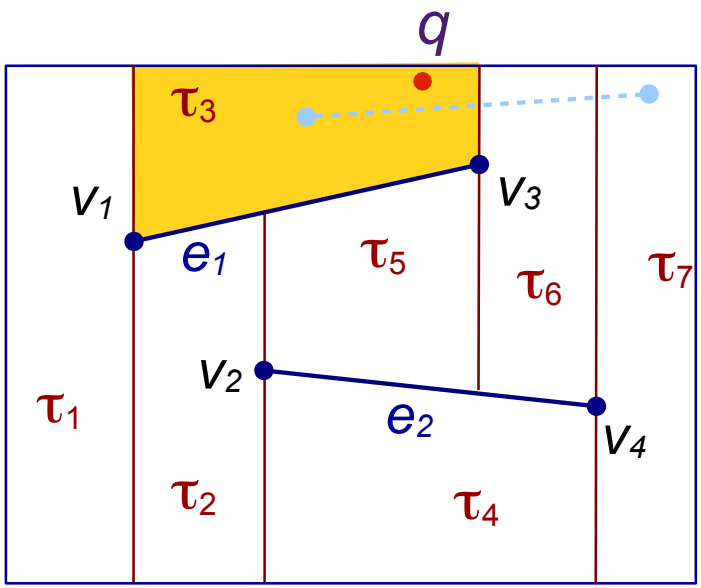
$$S_0 = B, \; S_1, \; S_2, \; \ldots \; S_n = S$$

- N_i = number of nodes created on π at step $i \in [1, n]$

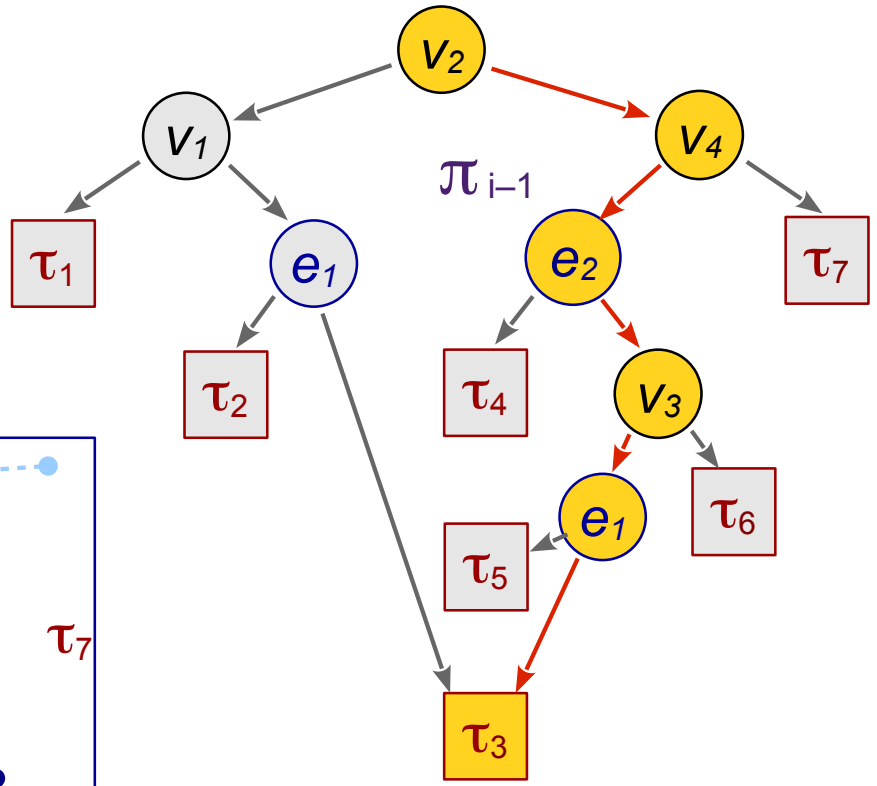
Point location costs: Path π to τ_i



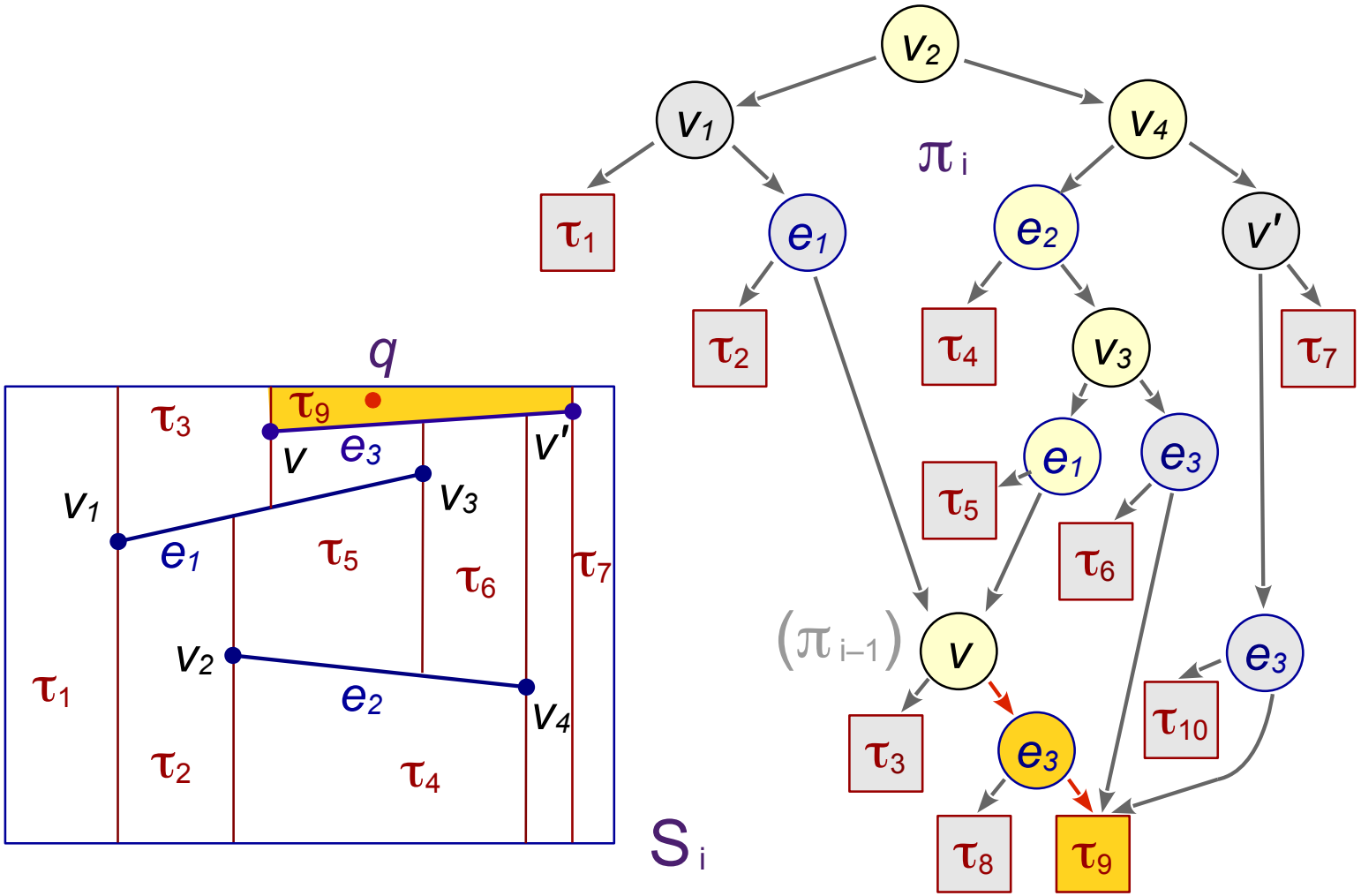
Point location costs: Path π to τ_{i-1}



S_{i-1}

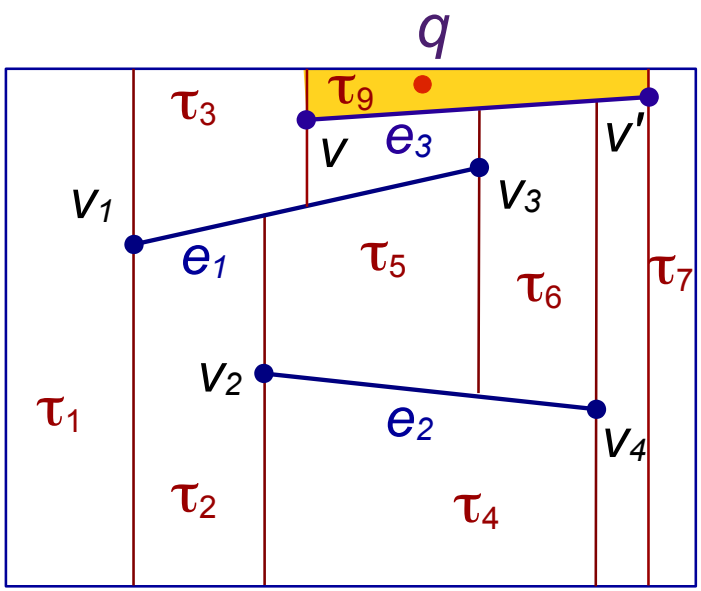


Point location costs: Trapezoid τ_i is created at step i

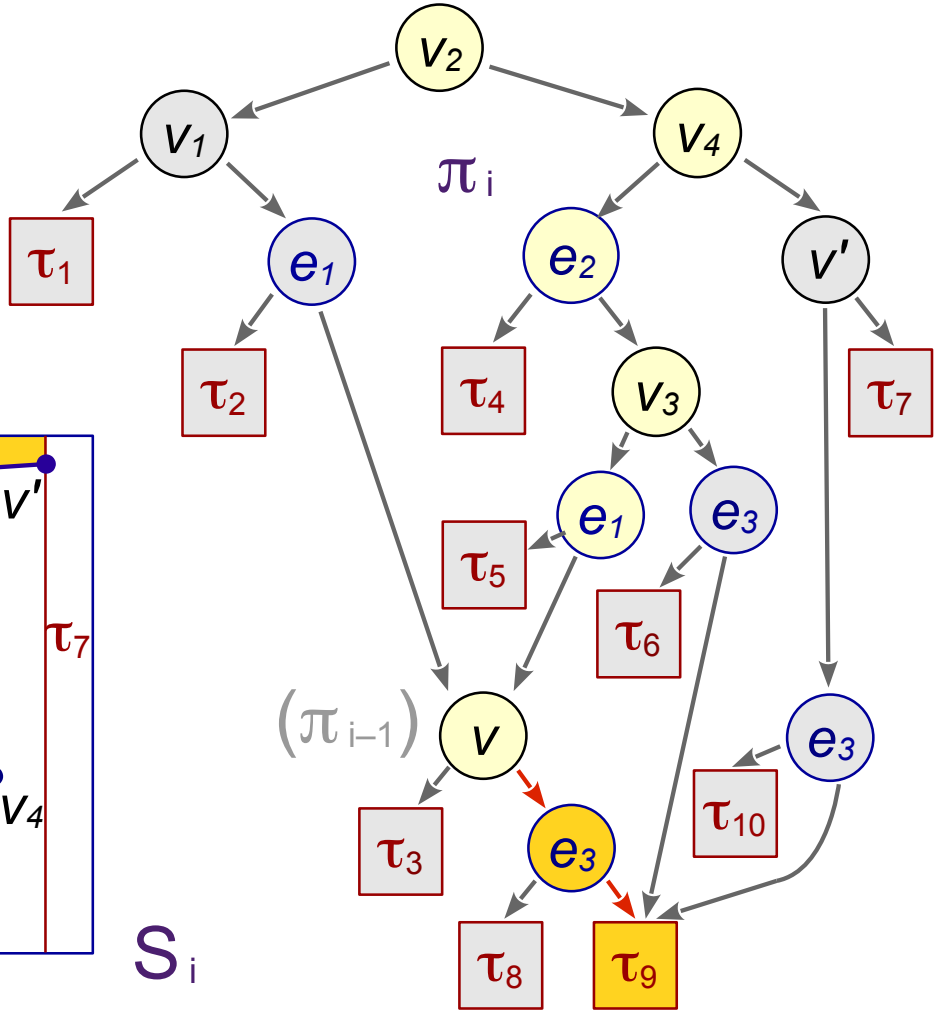


Point location costs: Trapezoid τ_i is created at step i

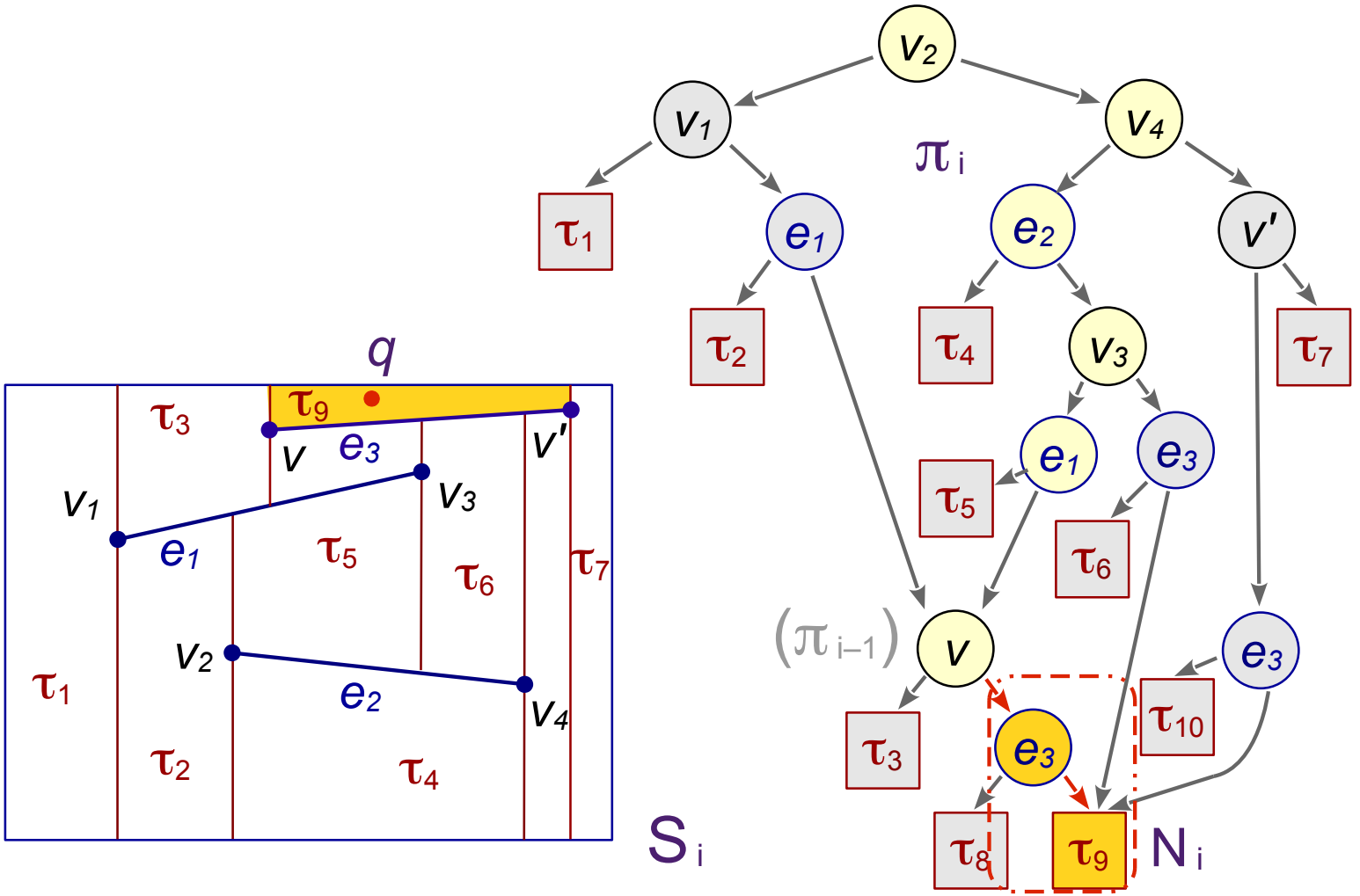
since e_i constrains τ_i !



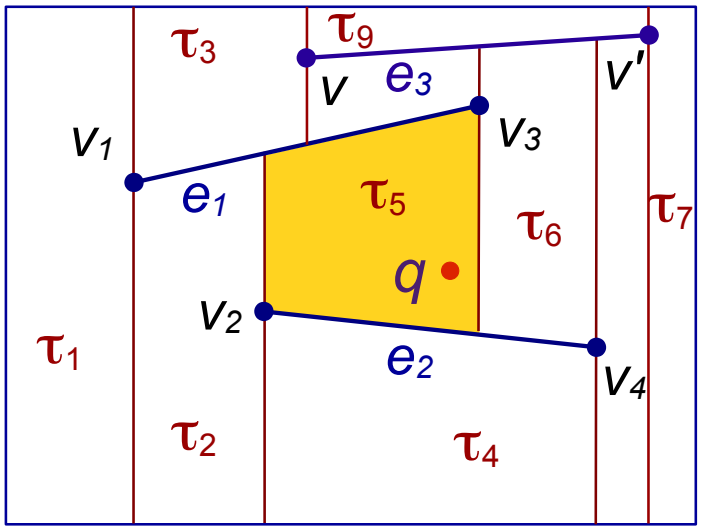
S_i



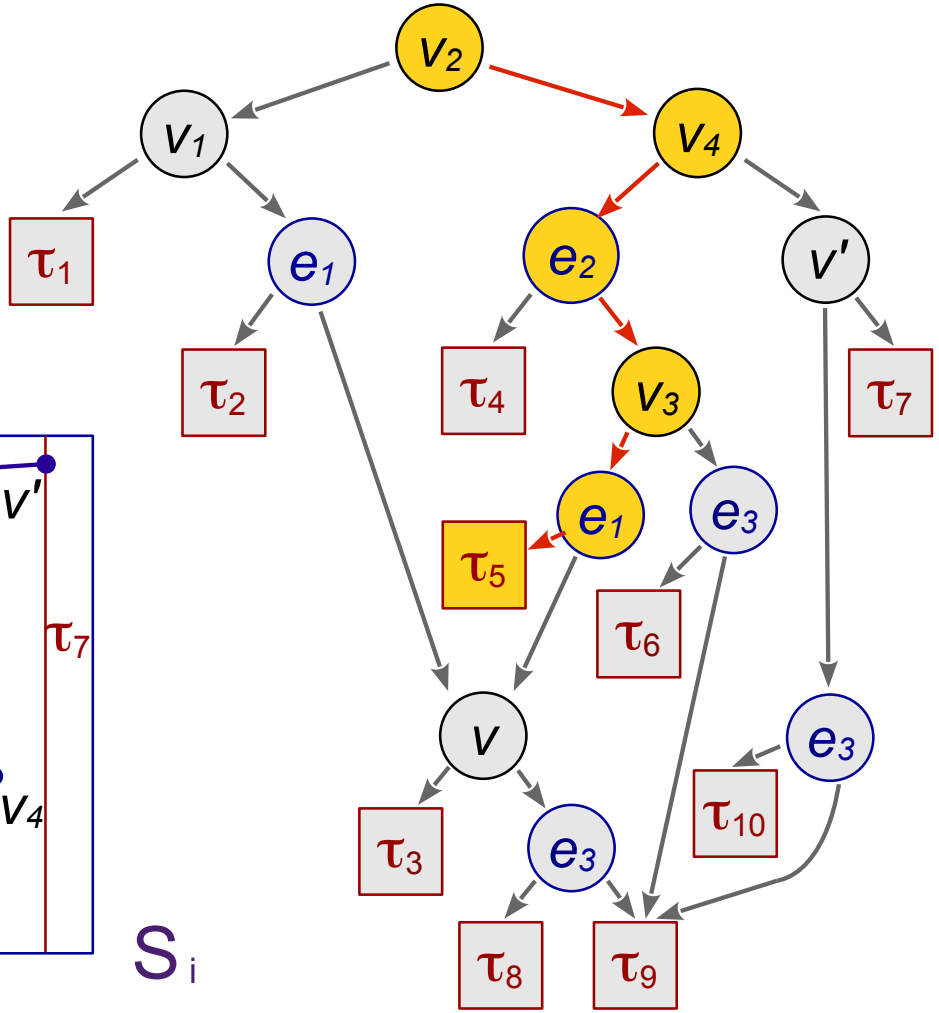
Point location costs: Step i contributes N_i nodes on π



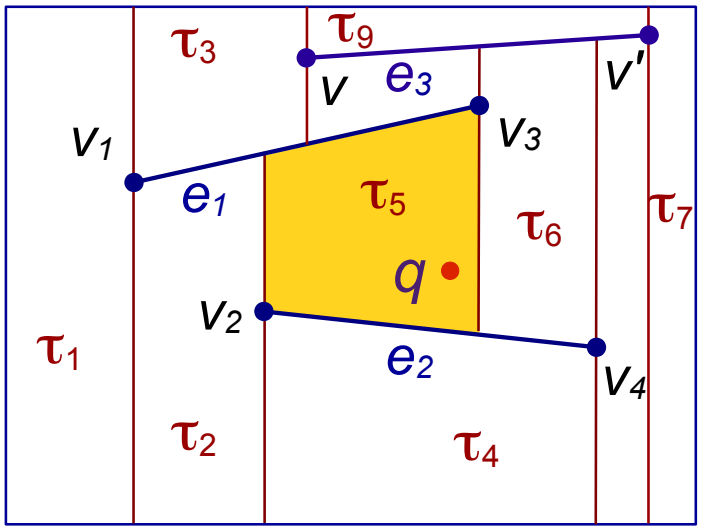
. . . But may not contribute nodes on a different path



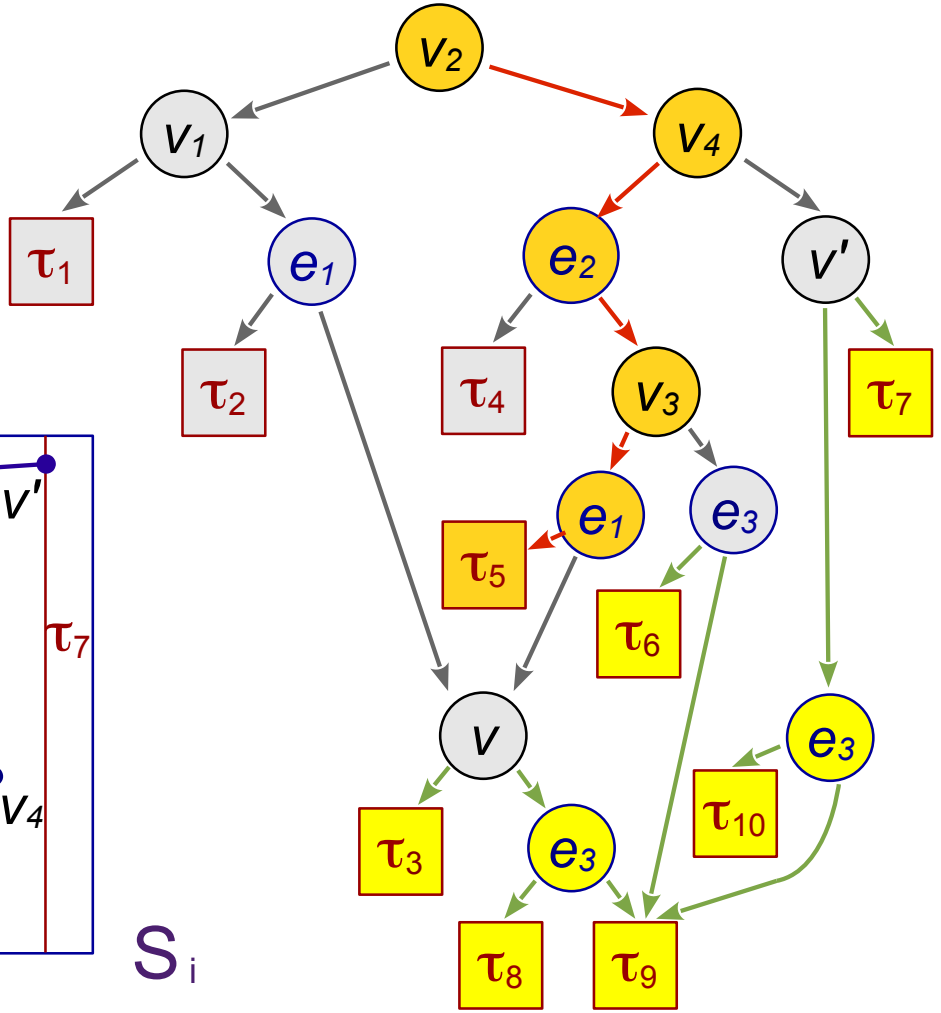
S_i



. . . But may not contribute nodes on a different path

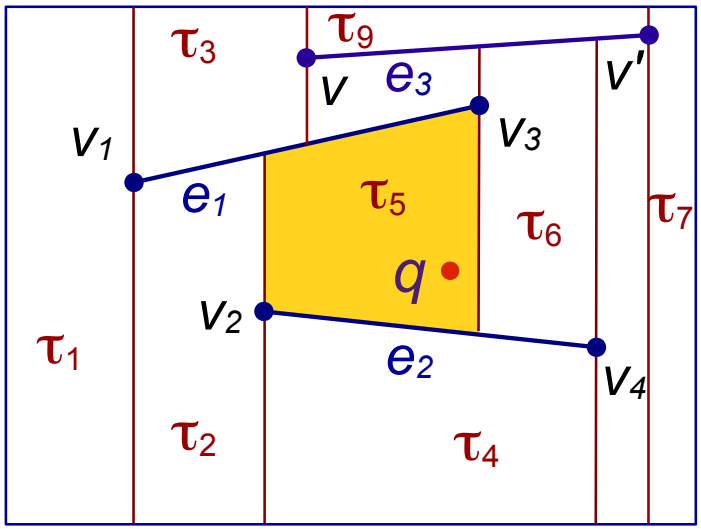


S_i

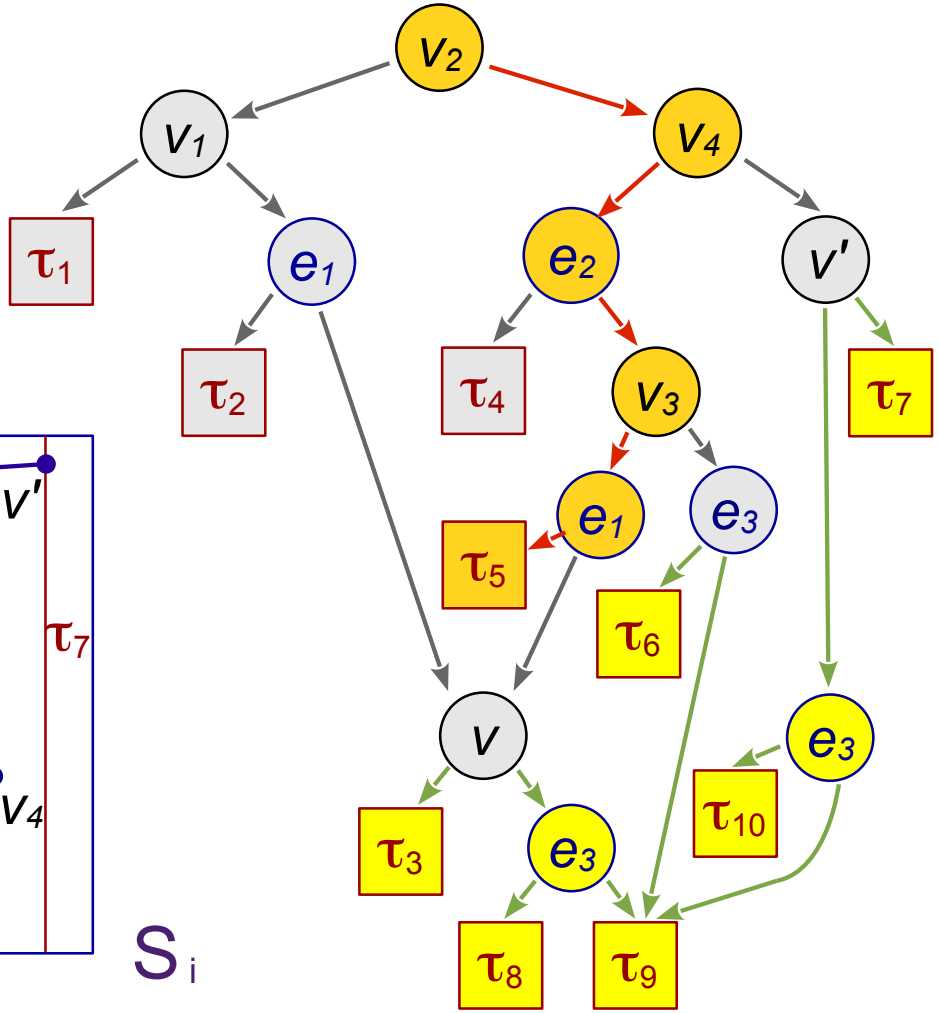


. . . But may not contribute nodes on a different path

when e_i does not constrain τ_i !



S_i



Point location costs

- Expected path length:

$$E[\sum_{i=1}^{n} N_i\ ] \quad = \quad \sum_{i=1}^{n} E[N_i]$$

- Of course $N_i \leq 3$
- For P_i = probability that nodes are added on π at step i:

$$E[N_i] \leq 3P_i$$

Point location costs

- Expected path length:

$$E[\sum_{i=1}^{n} N_i\ ] \quad = \quad \sum_{i=1}^{n} E[N_i]$$

- Of course $N_i \leq 3$
- For P_i = probability that nodes are added on π at step i:

$$E[N_i] \leq 3P_i$$

Point location costs

- Expected path length:

$$E[\sum_{i=1}^{n} N_i\ ] \quad = \quad \sum_{i=1}^{n} E[N_i]$$

- Of course $N_i \leq 3$
- For P_i = probability that nodes are added on π at step i:

$$E[N_i] \leq 3P_i$$

Point location costs

- At step i: $q \in \tau_i$ of S_i
- Step i contributes nodes to π precisely when $\tau_i \neq \tau_{i-1}$
- $\rightarrow$ τ_i was created at step i
- $\rightarrow$ τ_i is bounded by the edge e_i added at step i
- or meets one of its endpoints

Point location costs

- At step i: $q \in \tau_i$ of S_i
- Step i contributes nodes to π precisely when $\tau_i \neq \tau_{i-1}$
- $\rightarrow$ τ_i was created at step i
- $\rightarrow$ τ_i is bounded by the edge e_i added at step i
- or meets one of its endpoints

Point location costs

- At step i: $q \in \tau_i$ of S_i
- Step i contributes nodes to π precisely when $\tau_i \neq \tau_{i-1}$
- $\rightarrow$ τ_i was created at step i
- $\rightarrow$ τ_i is bounded by the edge e_i added at step i
- or meets one of its endpoints

Point location costs

- At step i: $q \in \tau_i$ of S_i
- Step i contributes nodes to π precisely when $\tau_i \neq \tau_{i-1}$
- $\rightarrow$ τ_i was created at step i
- $\rightarrow$ τ_i is bounded by the edge e_i added at step i
- or meets one of its endpoints

Point location costs

- At step i: $q \in \tau_i$ of S_i
- Step i contributes nodes to π precisely when $\tau_i \neq \tau_{i-1}$
- $\rightarrow$ τ_i was created at step i
- $\rightarrow$ τ_i is bounded by the edge e_i added at step i
- or meets one of its endpoints

Backward analysis

- Let us choose a particular set of i edges
- Also the resulting subdivision S_i is then fixed
- Which probability that τ_i disappears by removing e_i?
- $e_i = b_{\tau_i}$ or $e_i = t_{\tau_i}$ or
  l_{τ_i} endpoint of e_i or r_{τ_i} endpoint of e_i
- Each of the above cases arises with probability $1/i$
  (some technicalities should possibly be considered here,
  i.e. τ_i disappears if e_i is the *only* edge incident at l_{τ_i}/r_{τ_i})

Backward analysis

- Let us choose a particular set of i edges
- Also the resulting subdivision S_i is then fixed
- Which probability that τ_i disappears by removing e_i?
- $e_i = b_{\tau_i}$ or $e_i = t_{\tau_i}$ or
  l_{τ_i} endpoint of e_i or r_{τ_i} endpoint of e_i
- Each of the above cases arises with probability $1/i$
  (some technicalities should possibly be considered here,
  i.e. τ_i disappears if e_i is the *only* edge incident at l_{τ_i}/r_{τ_i})

Backward analysis

- Let us choose a particular set of i edges
- Also the resulting subdivision S_i is then fixed
- Which probability that τ_i disappears by removing e_i?
- $e_i = b_{\tau_i}$ or $e_i = t_{\tau_i}$ or
  l_{τ_i} endpoint of e_i or r_{τ_i} endpoint of e_i
- Each of the above cases arises with probability $1/i$
  (some technicalities should possibly be considered here,
  i.e. τ_i disappears if e_i is the *only* edge incident at l_{τ_i}/r_{τ_i})

Backward analysis

- Let us choose a particular set of i edges
- Also the resulting subdivision S_i is then fixed
- Which probability that τ_i disappears by removing e_i?
- $e_i = b_{\tau_i}$ or $e_i = t_{\tau_i}$ or
  l_{τ_i} endpoint of e_i or r_{τ_i} endpoint of e_i
- Each of the above cases arises with probability $1/i$
  (some technicalities should possibly be considered here,
  i.e. τ_i disappears if e_i is the *only* edge incident at l_{τ_i}/r_{τ_i})

Backward analysis

- Let us choose a particular set of i edges
- Also the resulting subdivision S_i is then fixed
- Which probability that τ_i disappears by removing e_i?
- $e_i = b_{\tau_i}$ or $e_i = t_{\tau_i}$ or
  l_{τ_i} endpoint of e_i or r_{τ_i} endpoint of e_i
- Each of the above cases arises with probability $1/i$
  (some technicalities should possibly be considered here,
  i.e. τ_i disappears if e_i is the *only* edge incident at l_{τ_i}/r_{τ_i})

Backward analysis

- Let us choose a particular set of i edges
- Also the resulting subdivision S_i is then fixed
- Which probability that τ_i disappears by removing e_i?
- $e_i = b_{\tau_i}$ or $e_i = t_{\tau_i}$ or
  l_{τ_i} endpoint of e_i or r_{τ_i} endpoint of e_i
- Each of the above cases arises with probability $1/i$
  (some technicalities should possibly be considered here,
  i.e. τ_i disappears if e_i is the *only* edge incident at l_{τ_i}/r_{τ_i})

Backward analysis

- To sum up:

$$E[N_i] \leq 3P_i \leq 3 \times \frac{4}{i}$$

- This bound does not depend on some specific S_i, hence it holds for the i-th step unconditionally

Backward analysis

- To sum up:

$$E[N_i] \leq 3P_i \leq 3 \times \frac{4}{i}$$

- This bound does not depend on some specific S_i, hence it holds for the i-th step unconditionally

Backward analysis

- To sum up:

$$E[N_i] \leq 3P_i \leq 3 \times \frac{4}{i}$$

- Thus:

$$E[\sum_{i=1}^{n} N_i] = \sum_{i=1}^{n} E[N_i] \leq 12 \sum_{i=1}^{n} \frac{1}{i}$$

$$= O(\log n)$$

Backward analysis

- To sum up:

$$E[N_i] \leq 3P_i \leq 3 \times \frac{4}{i}$$

- Thus:

$$E[\sum_{i=1}^{n} N_i] = \sum_{i=1}^{n} E[N_i] \leq 12 \sum_{i=1}^{n} \frac{1}{i}$$

$$= O(\log n)$$

Storage costs

- Size of trapezoidal map $=$ $O(\ n\ )$
- $\rightarrow$ Number of DAG's leaves $=$ $O(\ n\ )$
- Then size of DAG

$$= O(\ n\ ) + \sum_{i=1}^{n} |\ \{\textit{inner nodes created at step } i\}\ |$$

Storage costs

- Size of trapezoidal map $=$ $O(\ n\ )$
- $\rightarrow$ Number of DAG's leaves $=$ $O(\ n\ )$
- Then size of DAG

$$= O(\ n\ ) + \sum_{i=1}^{n} |\ \{\textit{inner nodes created at step } i\}\ |$$

Storage costs

- Size of trapezoidal map $=$ $O(\ n\ )$
- $\rightarrow$ Number of DAG's leaves $=$ $O(\ n\ )$
- Then size of DAG

$$= O(\ n\ ) + \sum_{i=1}^{n} |\ \{\textit{inner nodes created at step } i\}\ |$$

Storage costs

- In the *worst case*

$$|\{\textit{inner nodes created at step } i\}| = O(i)$$

- And size of DAG

$$= O(n) + \sum_{i=1}^{n} O(i) = O(n^2)$$

Storage costs

- In the *worst case*

$$|\{\textit{inner nodes created at step } i\}| = O(i)$$

- And size of DAG

$$= O(n) + \sum_{i=1}^{n} O(i) = O(n^2)$$

Storage costs

- However

$$|\{\textit{inner nodes created at step } i\}| < T_i$$

- where T_i = number of trapezoids created at i-th step

Storage costs

- However

$$|\{\textit{inner nodes created at step } i\}| < T_i$$

- where T_i = number of trapezoids created at i-th step

Backward analysis (again)

- Let us choose a particular set X_i of i edges
- Again, the resulting subdivision S_i is fixed
- $\tau \in S_i$ is created at step i if it is "constrained" by the last added edge e from X_i
- Each edge in S_i may play this role with probability $1/i$
- Hence

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} |\{\tau \in S_i : e \text{ constrains } \tau\}|$$

Backward analysis (again)

- Let us choose a particular set X_i of i edges
- Again, the resulting subdivision S_i is fixed
- $\tau \in S_i$ is created at step i if it is "constrained" by the last added edge e from X_i
- Each edge in S_i may play this role with probability $1/i$
- Hence

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} |\{\tau \in S_i : e \text{ constrains } \tau\}|$$

Backward analysis (again)

- Let us choose a particular set X_i of i edges
- Again, the resulting subdivision S_i is fixed
- $\tau \in S_i$ is created at step i if it is "constrained" by the last added edge e from X_i
- Each edge in S_i may play this role with probability $1/i$
- Hence

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} |\{\tau \in S_i : e \text{ constrains } \tau\}|$$

Backward analysis (again)

- Let us choose a particular set X_i of i edges
- Again, the resulting subdivision S_i is fixed
- $\tau \in S_i$ is created at step i if it is "constrained" by the last added edge e from X_i
- Each edge in S_i may play this role with probability $1/i$
- Hence

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} |\{\tau \in S_i : e \text{ constrains } \tau\}|$$

Backward analysis (again)

- Let us choose a particular set X_i of i edges
- Again, the resulting subdivision S_i is fixed
- $\tau \in S_i$ is created at step i if it is "constrained" by the last added edge e from X_i
- Each edge in S_i may play this role with probability $1/i$
- Hence

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} |\{\tau \in S_i : e \text{ constrains } \tau\}|$$

Backward analysis (again)

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} | \{\tau \in S_i \ : \ e \ constrsains \ \tau\} |$$

$$= \frac{1}{i} \sum_{e \in X_i} \sum_{\tau \in S_i} \delta_\tau^e = \frac{1}{i} \sum_{\tau \in S_i} \sum_{e \in X_i} \delta_\tau^e$$

$$= \frac{1}{i} \sum_{\tau \in S_i} | \{e \in X_i \ : \ e \ constrsains \ \tau\} |$$

- where $\delta_\tau^e = 1$ if e constrsains τ; otherwise $\delta_\tau^e = 0$

Backward analysis (again)

$$E[T_i] = \frac{1}{i} \sum_{e \in X_i} |\{\tau \in S_i : e \text{ constrsains } \tau\}|$$

$$= \frac{1}{i} \sum_{e \in X_i} \sum_{\tau \in S_i} \delta_\tau^e = \frac{1}{i} \sum_{\tau \in S_i} \sum_{e \in X_i} \delta_\tau^e$$

$$= \frac{1}{i} \sum_{\tau \in S_i} |\{e \in X_i : e \text{ constrsains } \tau\}|$$

- where $\delta_\tau^e = 1$ if e constrsains τ; otherwise $\delta_\tau^e = 0$

Backward analysis (again)

- We already know that

$$| \{ e \in X_i \ : \ e \ constrsains \ \tau \} | \ \leq \ 4$$

- Then, independent of the specific S_i:

$$E[T_i] \ = \ \frac{1}{i} \sum_{\tau \in S_i} | \{ e \in X_i \ : \ e \ constrsains \ \tau \} |$$

$$\leq \ \frac{4}{i} |S_i| \ = \ \frac{4}{i} O(\ i \ ) \ = \ O(\ 1 \ )$$

Backward analysis (again)

- We already know that

$$| \{ e \in X_i \ : \ e \ constrsains \ \tau \} | \ \leq \ 4$$

- Then, independent of the specific S_i:

$$E[T_i] \ = \ \frac{1}{i} \sum_{\tau \in S_i} | \{ e \in X_i \ : \ e \ constrsains \ \tau \} |$$

$$\leq \ \frac{4}{i} |S_i| \ = \ \frac{4}{i} O(\ i \ ) \ = \ O(\ 1 \ )$$

Backward analysis (again)

- We already know that

$$| \{ e \in X_i \ : \ e \ constrsains \ \tau \} | \ \leq \ 4$$

- Then, independent of the specific S_i:

$$E[T_i] \ = \ \frac{1}{i} \sum_{\tau \in S_i} | \{ e \in X_i \ : \ e \ constrsains \ \tau \} |$$

$$\leq \ \frac{4}{i} |S_i| \ = \ \frac{4}{i} O(\ i \ ) \ = \ O(\ 1 \ )$$

Expected size of DAG

- As a consequence:

$$E[\,|\,\{\textit{inner nodes created at step } i\}\,|\,] \;=\; O(\,1\,)$$

- And $E[\,\textit{DAG's size}\,]$

$$= O(\,n\,) + E[\sum_{i=1}^{n} |\,\{\textit{inner nodes created at step } i\}\,|\,]$$

$$= O(\,n\,) + \sum_{i=1}^{n} E[\,|\,\{\textit{inner nodes created at step } i\}\,|\,]$$

$$= O(\,n\,) + \sum_{i=1}^{n} O(\,1\,) \;=\; O(\,n\,)$$

Expected size of DAG

- As a consequence:

$$E[\,|\,\{\textit{inner nodes created at step } i\}\,|\,] \;=\; O(\,1\,)$$

- And $E[\,\textit{DAG's size}\,]$

$$= O(\,n\,) + E[\sum_{i=1}^{n} |\,\{\textit{inner nodes created at step } i\}\,|\,]$$

$$= O(\,n\,) + \sum_{i=1}^{n} E[\,|\,\{\textit{inner nodes created at step } i\}\,|\,]$$

$$= O(\,n\,) + \sum_{i=1}^{n} O(\,1\,) \;=\; O(\,n\,)$$

Expected preprocessing costs

- At i-th step...
- Point location (e's leftmost endpoint): $O(\log i)$
- New trapezoids + updating DAG: $O(E[T_i]) = O(1)$
- Overall:

$$\sum_{i=1}^{n} [O(\log i) + O(1)]$$

$$= O(n \log n)$$

Expected preprocessing costs

- At i-th step...
- Point location (e's leftmost endpoint): $O(\log i)$
- New trapezoids + updating DAG: $O(E[T_i]) = O(1)$
- Overall:

$$\sum_{i=1}^{n} [O(\log i) + O(1)]$$

$$= O(n \log n)$$

Expected preprocessing costs

- At i-th step...
- Point location (e's leftmost endpoint): $O(\log i)$
- New trapezoids + updating DAG: $O(E[T_i]) = O(1)$
- Overall:

$$\sum_{i=1}^{n} [O(\log i) + O(1)]$$

$$= O(n \log n)$$

Expected preprocessing costs

- At i-th step...
- Point location (e's leftmost endpoint): $O(\log i)$
- New trapezoids + updating DAG: $O(E[T_i]) = O(1)$
- Overall:

$$\sum_{i=1}^{n} [\, O(\log i) + O(1) \,]$$

$$= O(n \log n)$$

Summing up...

- Preprocessing: $O(\, n \log n \,)$
- Storage: $O(\, n \,)$
- Point location: $O(\, \log n \,)$
- Expected costs!

Summing up...

- Preprocessing: $O(\, n \log n \,)$
- Storage: $O(\, n \,)$
- Point location: $O(\, \log n \,)$
- Expected costs!

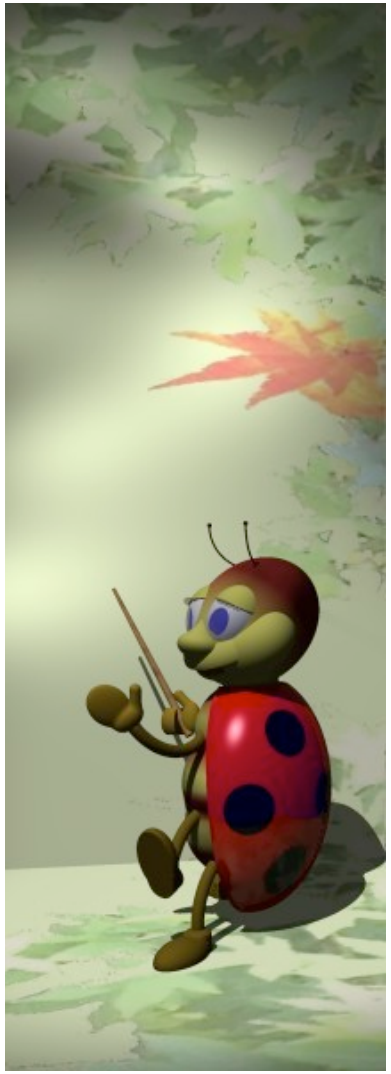
Summing up...

- Preprocessing: $O(\ n \log n\ )$
- Storage: $O(\ n\ )$
- Point location: $O(\ \log n\ )$
- Expected costs!

Summing up...

- Preprocessing: $O(\ n \log n\ )$
- Storage: $O(\ n\ )$
- Point location: $O(\ \log n\ )$
- Expected costs!

Outline



Provisional assumptions

- vertices in *general position*
- i.e. vertices not vertically aligned w.r.t. each other
- query points not vertically aligned with vertices

Provisional assumptions

- vertices in *general position*
- i.e. vertices not vertically aligned w.r.t. each other
- query points not vertically aligned with vertices

Provisional assumptions

- vertices in *general position*
- i.e. vertices not vertically aligned w.r.t. each other
- query points not vertically aligned with vertices

Treatment of “degeneracies”

- Very small rotation/affine transformation ϕ
- $\phi(x, y) = (x + \epsilon y, y)$
- Actually, just *symbolic perturbation*

Treatment of “degeneracies”

- Very small rotation/affine transformation ϕ
- $\phi(x, y) = (x + \epsilon y, y)$
- Actually, just *symbolic perturbation*

Treatment of “degeneracies”

- Very small rotation/affine transformation ϕ
- $\phi(x, y) = (x + \epsilon y, y)$
- Actually, just *symbolic perturbation*

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Original vs. transformed items

- The algorithm does not compute new geometric items: Only two simple operations...
- Left-to-right order
  - If $x' \neq x$ same order (ϵ small)
  - If $x' = x \rightarrow$ *lexicographic order*!
- Above/on/below edge e
  - In essence, ϕ preserves such relations
  - Just some specific treatment for vertical (original) edges

Left-to-right order

- Original items: $(x, y),\ (x', y')$
- Transformed items: $(x + \epsilon y, y),\ (x' + \epsilon y', y')$
- $(x' + \epsilon y') - (x + \epsilon y) = (x' - x) + \epsilon(y' - y)$
- If $x' \neq x$ assume ϵ small enough: $\epsilon|y' - y| < |x' - x|$
- If $x' = x$ just consider $y' - y$

Left-to-right order

- Original items: (x, y), (x', y')
- Transformed items: $(x + \epsilon y, y)$, $(x' + \epsilon y', y')$
- $(x' + \epsilon y') - (x + \epsilon y) = (x' - x) + \epsilon(y' - y)$
- If $x' \neq x$ assume ϵ small enough: $\epsilon|y' - y| < |x' - x|$
- If $x' = x$ just consider $y' - y$

Left-to-right order

- Original items: $(x, y),\ (x', y')$
- Transformed items: $(x + \epsilon y, y),\ (x' + \epsilon y', y')$
- $(x' + \epsilon y') - (x + \epsilon y) \ = \ (x' - x) + \epsilon(y' - y)$
- If $x' \neq x$ assume ϵ small enough: $\epsilon|y' - y| < |x' - x|$
- If $x' = x$ just consider $y' - y$

Left-to-right order

- Original items: $(x, y),\ (x', y')$
- Transformed items: $(x + \epsilon y, y),\ (x' + \epsilon y', y')$
- $(x' + \epsilon y') - (x + \epsilon y) = (x' - x) + \epsilon(y' - y)$
- If $x' \neq x$ assume ϵ small enough: $\epsilon|y' - y| < |x' - x|$
- If $x' = x$ just consider $y' - y$

Left-to-right order

- Original items: (x, y), (x', y')
- Transformed items: $(x + \epsilon y, y)$, $(x' + \epsilon y', y')$
- $(x' + \epsilon y') - (x + \epsilon y) = (x' - x) + \epsilon(y' - y)$
- If $x' \neq x$ assume ϵ small enough: $\epsilon|y' - y| < |x' - x|$
- If $x' = x$ just consider $y' - y$

Above/on/below edge

- Point $q : (x, y) \rightarrow (x + \epsilon y, y)$
- Edge $e : [(x', y')\ (x'', y'')] \rightarrow [(x' + \epsilon y', y')\ (x'' + \epsilon y'', y'')]$
- Suppose without loss of generality that $x' \leq x''$
- The algorithm tests ϕq against ϕe only if

$$x' + \epsilon y' \leq x + \epsilon y \leq x'' + \epsilon y''$$

$$\Rightarrow x' \leq x \leq x'' \qquad (\epsilon \textit{ small})$$

Above/on/below edge

- Point $q : (x, y) \rightarrow (x + \epsilon y, y)$
- Edge $e : [(x', y') (x'', y'')] \rightarrow [(x' + \epsilon y', y') (x'' + \epsilon y'', y'')]$
- Suppose without loss of generality that $x' \leq x''$
- The algorithm tests ϕq against ϕe only if

$$x' + \epsilon y' \leq x + \epsilon y \leq x'' + \epsilon y''$$

$$\Rightarrow x' \leq x \leq x'' \qquad (\epsilon \textit{ small})$$

Above/on/below edge

- Point $q : (x, y) \rightarrow (x + \epsilon y, y)$
- Edge $e : [(x', y') (x'', y'')] \rightarrow [(x' + \epsilon y', y') (x'' + \epsilon y'', y'')]$
- Suppose without loss of generality that $x' \leq x''$
- The algorithm tests ϕq against ϕe only if

$$x' + \epsilon y' \leq x + \epsilon y \leq x'' + \epsilon y''$$

$$\Rightarrow x' \leq x \leq x'' \qquad (\epsilon \textit{ small})$$

Above/on/below edge

- Point $q : (x, y) \rightarrow (x + \epsilon y, y)$
- Edge $e : [(x', y') (x'', y'')] \rightarrow [(x' + \epsilon y', y') (x'' + \epsilon y'', y'')]$
- Suppose without loss of generality that $x' \leq x''$
- The algorithm tests ϕq against ϕe only if

$$x' + \epsilon y' \leq x + \epsilon y \leq x'' + \epsilon y''$$

$$\Rightarrow x' \leq x \leq x'' \qquad (\epsilon \textit{ small})$$

Above/on/below edge

- If $x' = x''$ then $x' = x = x''$ and $y' \leq y \leq y''$
- This means that $q \in e$ and ϕ preserves incidence:

$$(x + \epsilon y, y) \quad \in \quad [(x + \epsilon y', y') \ (x + \epsilon y'', y'')]$$

- Otherwise y is to be tested against

$$y^* = y' + \frac{(x + \epsilon y) - (x' + \epsilon y')}{(x'' + \epsilon y'') - (x' + \epsilon y')} (y'' - y')$$

$$= y' + \frac{(x - x') + \epsilon(y - y')}{(x'' - x') + \epsilon(y'' - y')} (y'' - y')$$

Above/on/below edge

- If $x' = x''$ then $x' = x = x''$ and $y' \leq y \leq y''$
- This means that $q \in e$ and ϕ preserves incidence:

$$(x + \epsilon y, y) \quad \in \quad [(x + \epsilon y', y') \ (x + \epsilon y'', y'')]$$

- Otherwise y is to be tested against

$$y^* = y' + \frac{(x + \epsilon y) - (x' + \epsilon y')}{(x'' + \epsilon y'') - (x' + \epsilon y')} (y'' - y')$$

$$= y' + \frac{(x - x') + \epsilon(y - y')}{(x'' - x') + \epsilon(y'' - y')} (y'' - y')$$

Above/on/below edge

- If $x' = x''$ then $x' = x = x''$ and $y' \leq y \leq y''$
- This means that $q \in e$ and ϕ preserves incidence:

$$(x + \epsilon y, y) \quad \in \quad [(x + \epsilon y', y') \ \ (x + \epsilon y'', y'')]$$

- Otherwise y is to be tested against

$$y^* = y' + \frac{(x + \epsilon y) - (x' + \epsilon y')}{(x'' + \epsilon y'') - (x' + \epsilon y')} (y'' - y')$$

$$= y' + \frac{(x - x') + \epsilon(y - y')}{(x'' - x') + \epsilon(y'' - y')} (y'' - y')$$

Above/on/below edge

- By making ϵ smaller and smaller, y^* gets as close as we like to

$$y' + \frac{x - x'}{x'' - x'}(y'' - y')$$

- i.e. the corresponding expression for the original items

Above/on/below edge

- By making ϵ smaller and smaller, y^* gets as close as we like to

$$y' + \frac{x - x'}{x'' - x'}(y'' - y')$$

- i.e. the corresponding expression for the original items

Original vs. transformed items

- Moreover, *incidences* are invariant by linear transformation
- . . . and we know if points are the same or if a point lies on some edge
- To sum up: we *can* compute everything *without* carrying out any transformation
- But of course we build a trapezoidal map for the *transformed* edges! (e.g. "very thin" trapezoids)

Original vs. transformed items

- Moreover, *incidences* are invariant by linear transformation
- . . . and we know if points are the same or if a point lies on some edge
- To sum up: we *can* compute everything *without* carrying out any transformation
- But of course we build a trapezoidal map for the *transformed* edges! (e.g. "very thin" trapezoids)

Original vs. transformed items

- Moreover, *incidences* are invariant by linear transformation
- . . . and we know if points are the same or if a point lies on some edge
- To sum up: we *can* compute everything *without* carrying out any transformation
- But of course we build a trapezoidal map for the *transformed* edges! (e.g. "very thin" trapezoids)

Original vs. transformed items

- Moreover, *incidences* are invariant by linear transformation
- . . . and we know if points are the same or if a point lies on some edge
- To sum up: we *can* compute everything *without* carrying out any transformation
- But of course we build a trapezoidal map for the *transformed* edges! (e.g. “very thin” trapezoids)

Original vs. transformed items

- Moreover, *incidences* are invariant by linear transformation
- . . . and we know if points are the same or if a point lies on some edge
- To sum up: we *can* compute everything *without* carrying out any transformation
- But of course we build a trapezoidal map for the *transformed* edges! (e.g. “very thin” trapezoids)

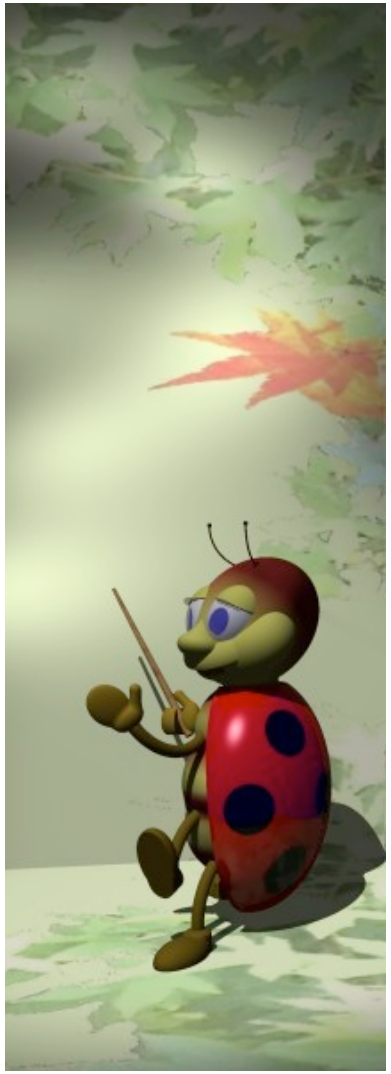
What about query points?

- Since we don't actually compute anything related to ϵ...
- We can think of a sufficiently small ϵ to accommodate for every query point q

What about query points?

- Since we don't actually compute anything related to ϵ...
- We can think of a sufficiently small ϵ
  to accommodate for every query point q

Outline



References


K. Mulmuley (1990)
A fast planar partition algorithm – I
Journal of Symbolic Computation, 10(3)


R. Seidel (1991)
A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons
Computational Geometry: Theory & Applications, 1(1)