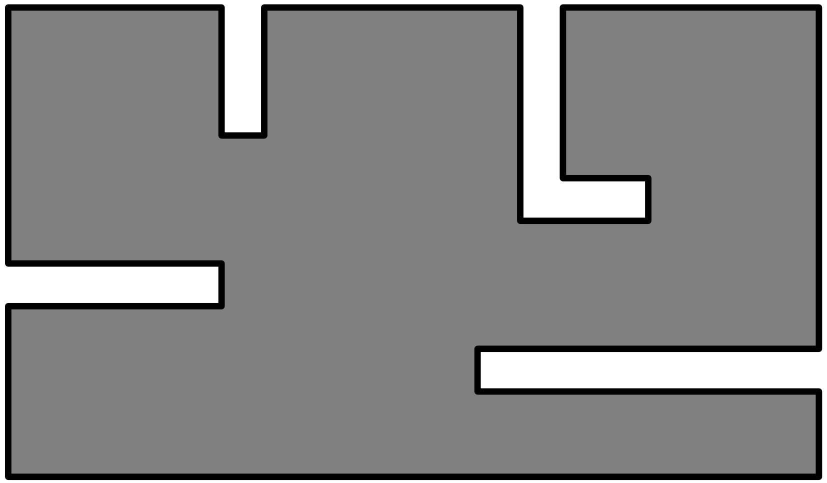
Computational Geometry – Exercise
Triangulation of Polygons & Linear Programming

LEHRSTUHL FÜR ALGORITHMIK I · INSTITUT FÜR THEORETISCHE INFORMATIK · FAKULTÄT FÜR INFORMATIK

Guido Brückner
23.05.2018

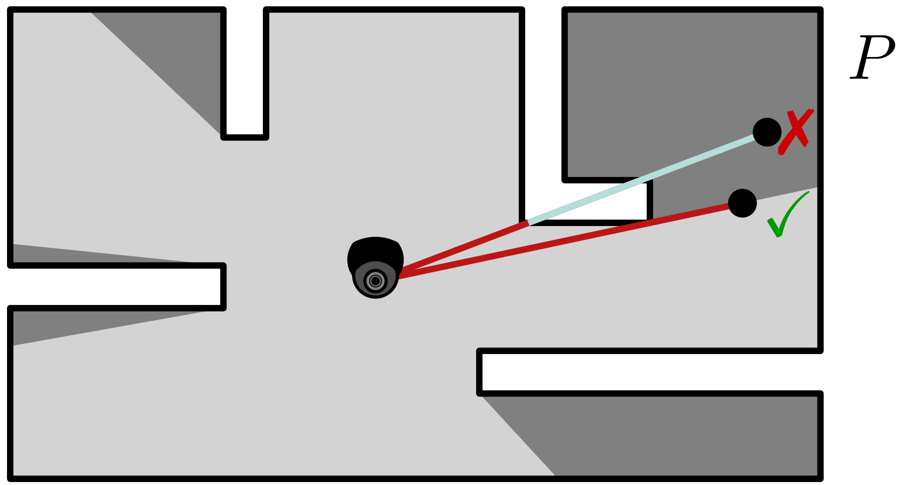
The Art-Gallery-Problem

Task: Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.



The Art-Gallery-Problem

Task: Install a number of cameras in an art gallery so that every part of the galery is visible to at least one of them.



Assumption: Art gallery is a *simple* polygon P with n corners (no self-intersections, no holes)

Observation: each camera observes a star-shaped region

Definition: Point $p \in P$ is *visible* from $c \in P$ if $\overline{cp} \in P$

Goal: Use as few cameras as possible!

NP-hard!

$\rightarrow$ The number depends on the number of corners n and on the shape of P

Observation of the Border

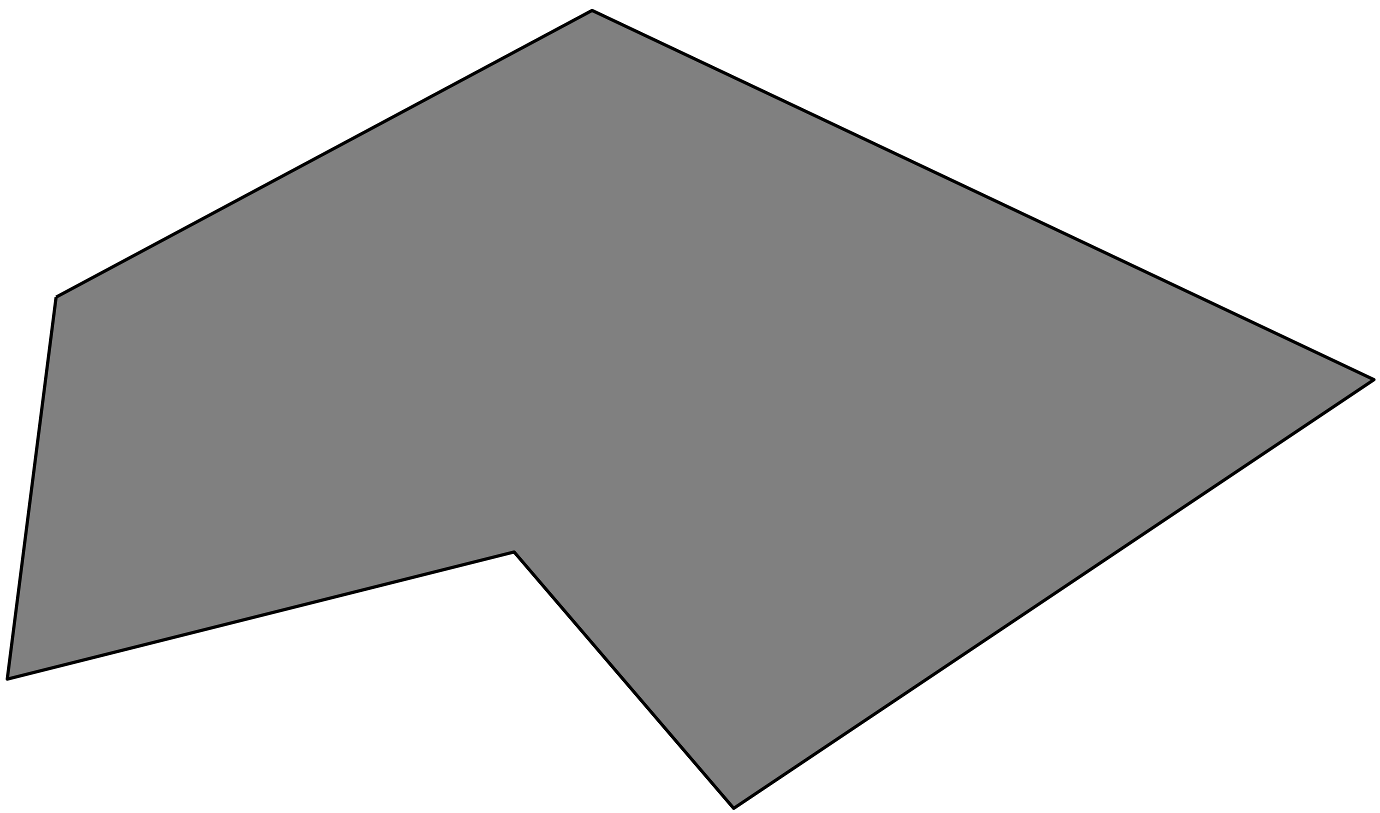
Prove or falsify the following statement.

> *Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.*

Observation of the Border

Prove or falsify the following statement.

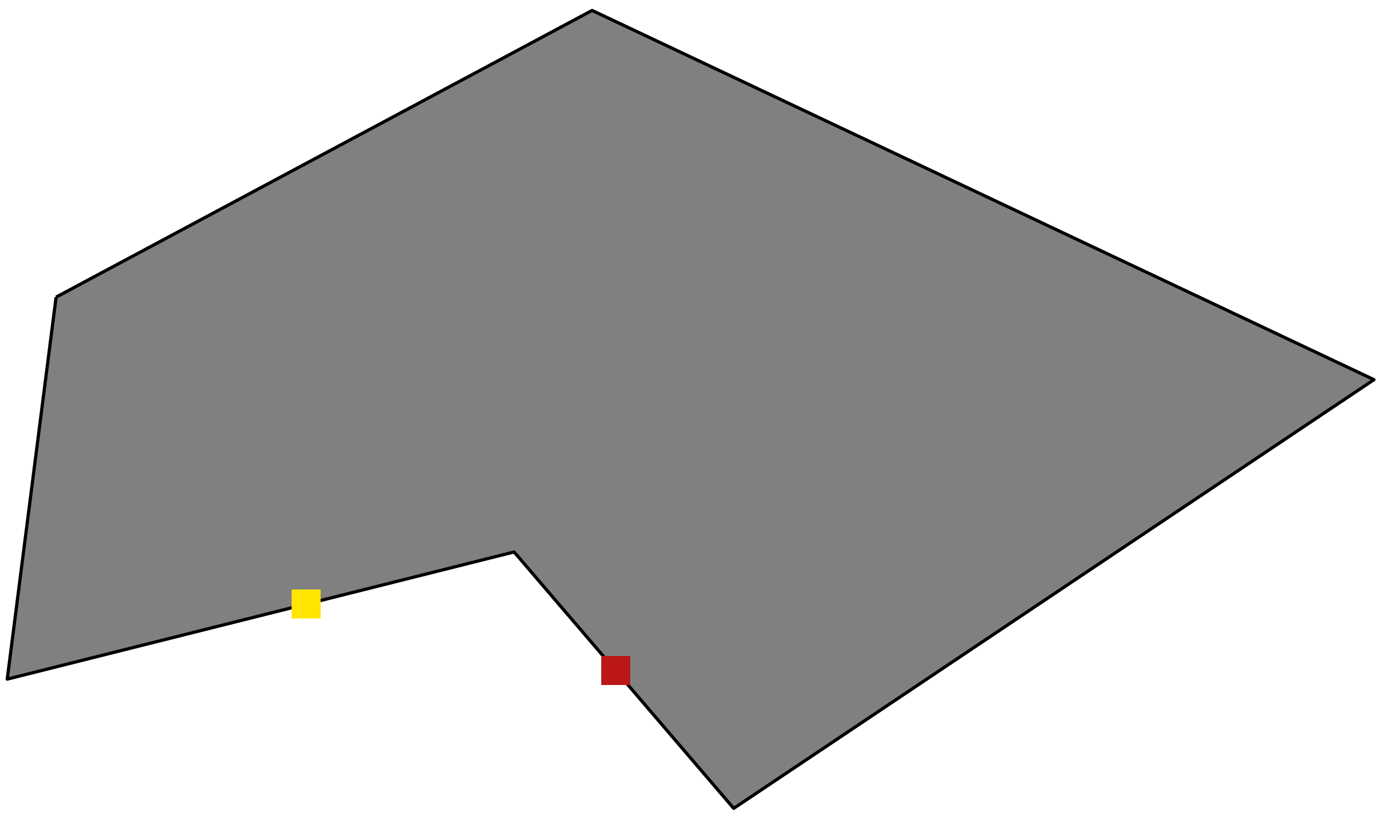
Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.



Observation of the Border

Prove or falsify the following statement.

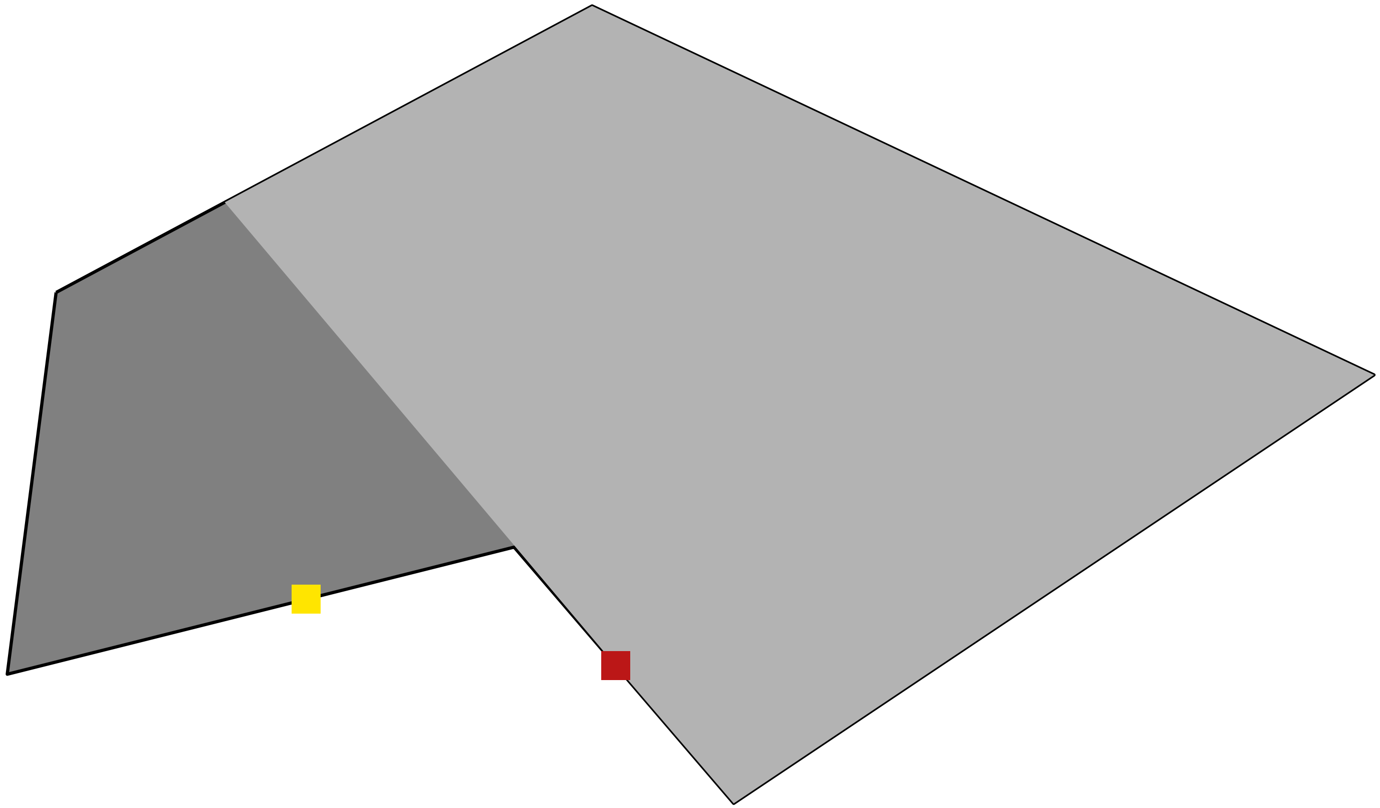
Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.



Observation of the Border

Prove or falsify the following statement.

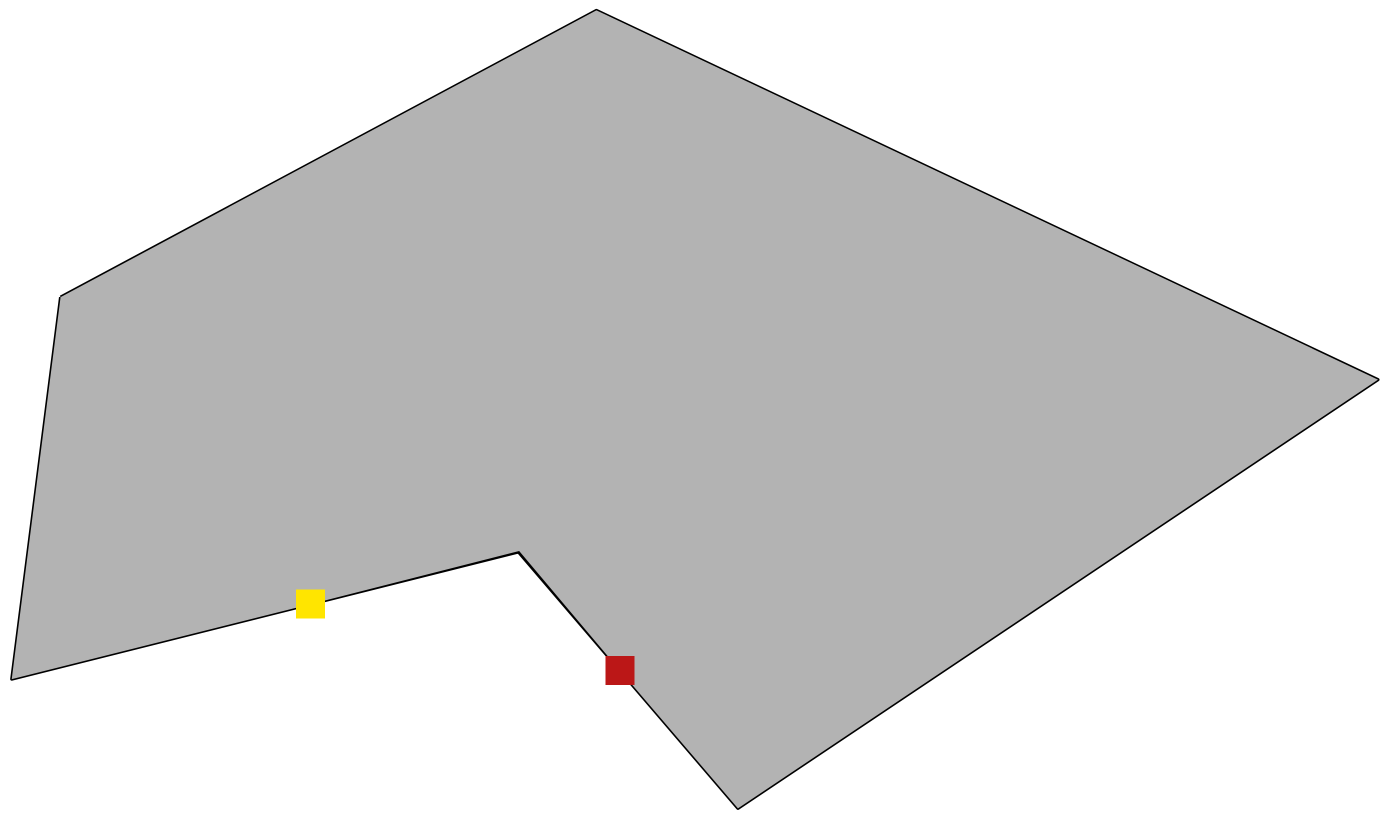
> *Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.*



Observation of the Border

Prove or falsify the following statement.

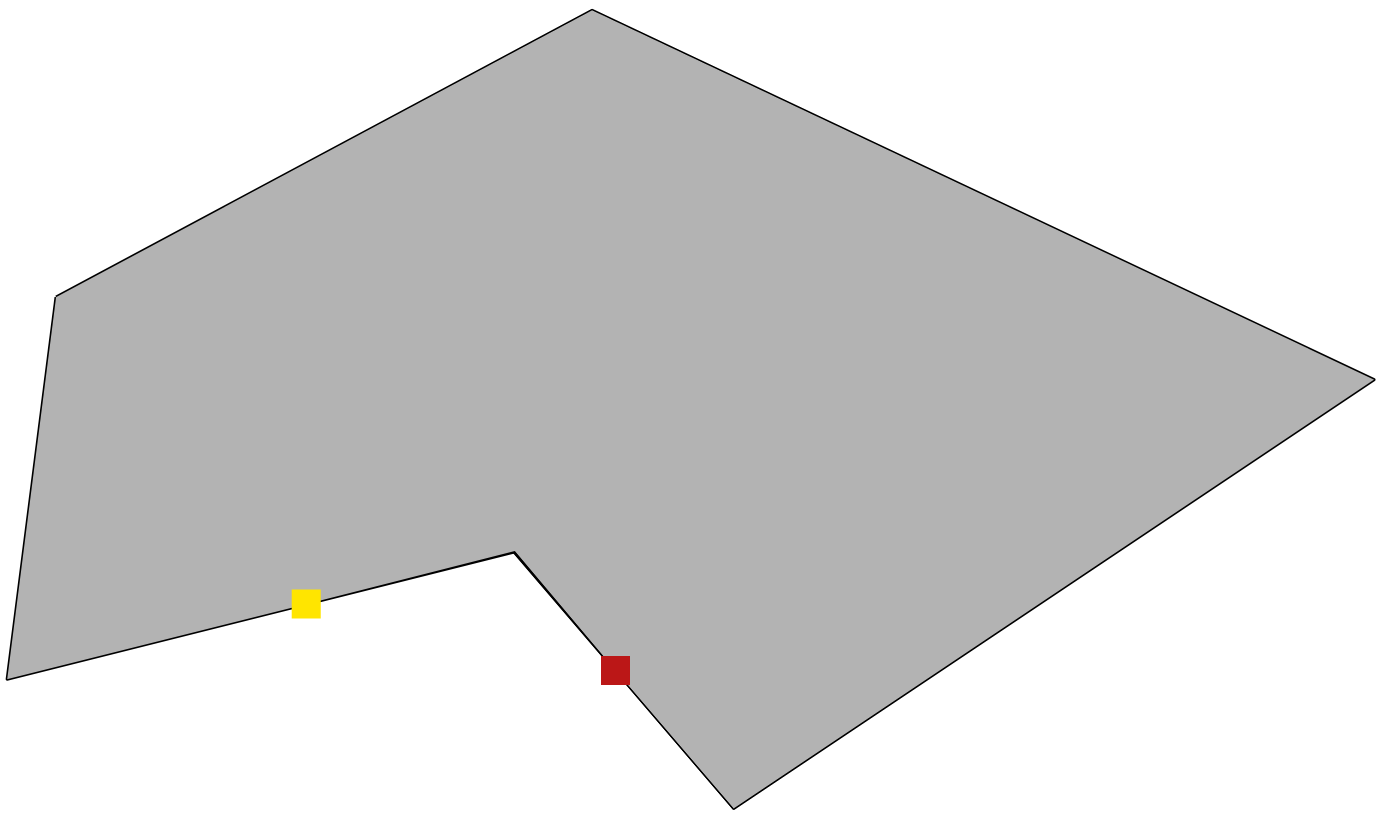
Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.



Observation of the Border

Prove or falsify the following statement.

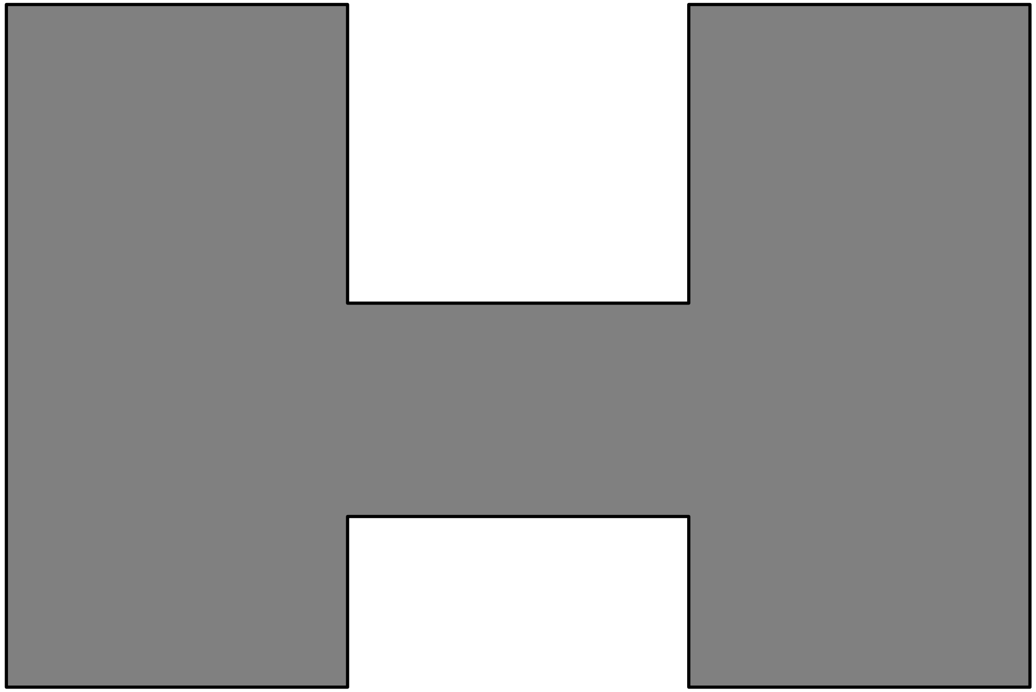
Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.



Observation of the Border

Prove or falsify the following statement.

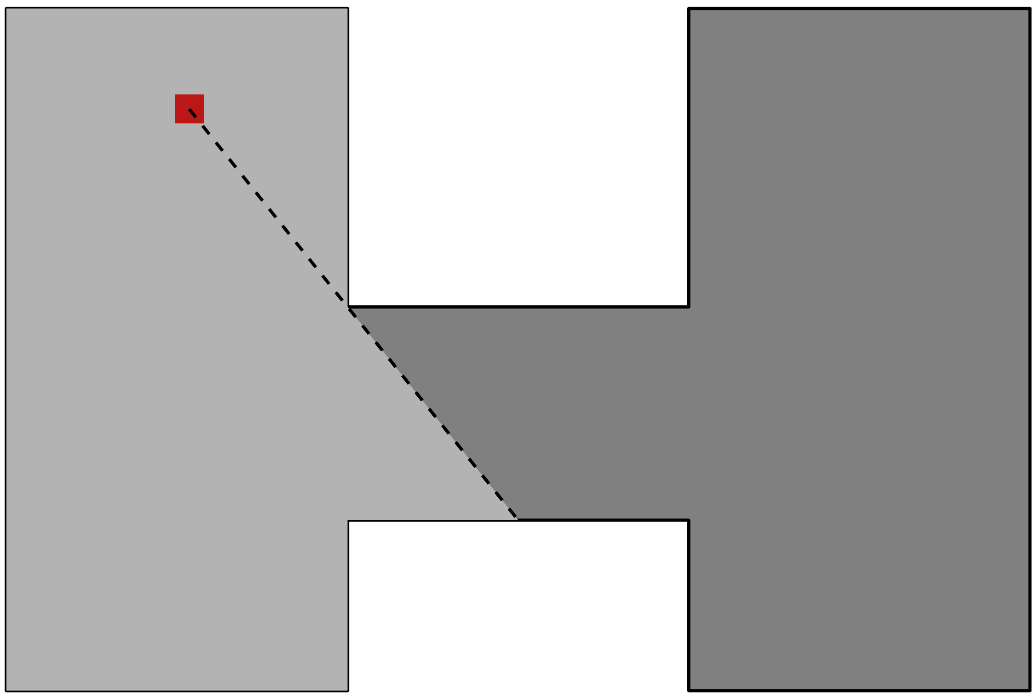
Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.



Observation of the Border

Prove or falsify the following statement.

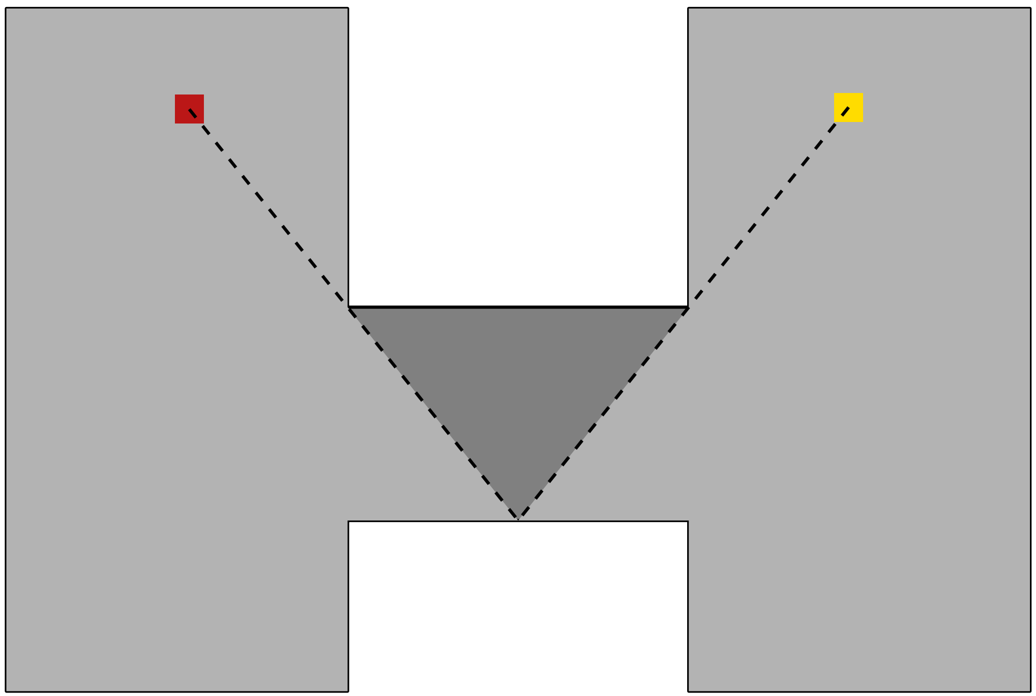
> *Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.*



Observation of the Border

Prove or falsify the following statement.

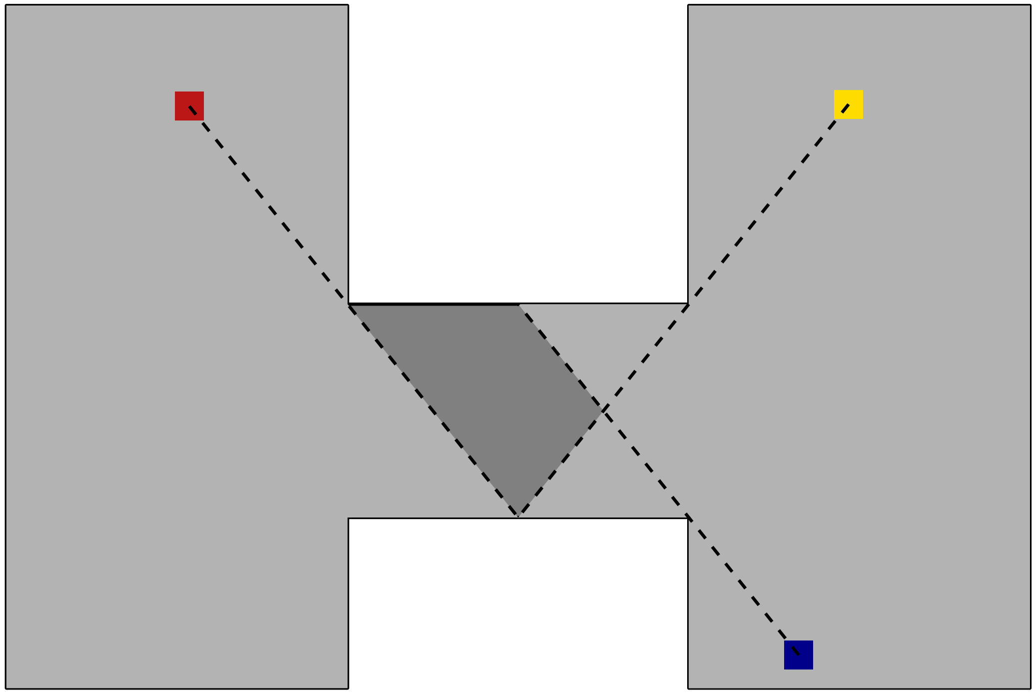
> *Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.*



Observation of the Border

Prove or falsify the following statement.

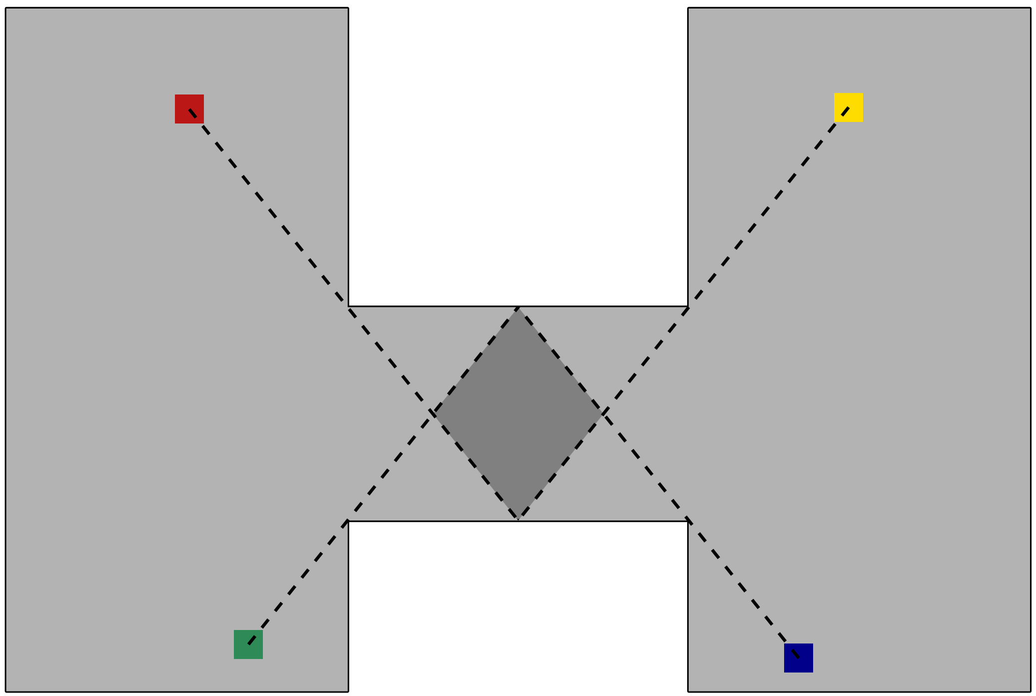
> *Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.*



Observation of the Border

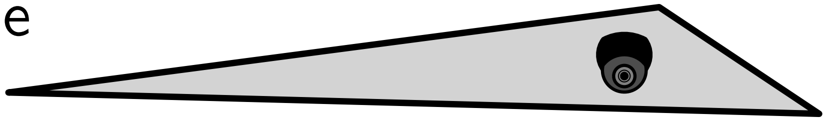
Prove or falsify the following statement.

Let $\mathcal{P}$ be a simple polygon and consider a set of cameras that together observe the complete border of P, then they also observe the complete interior of P.



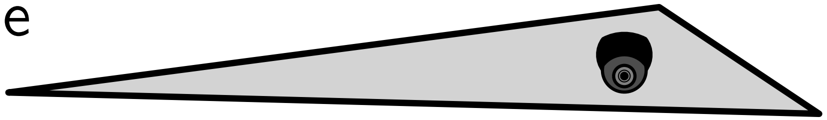
Problem Simplification

Observation: It is easy to guard a triangle

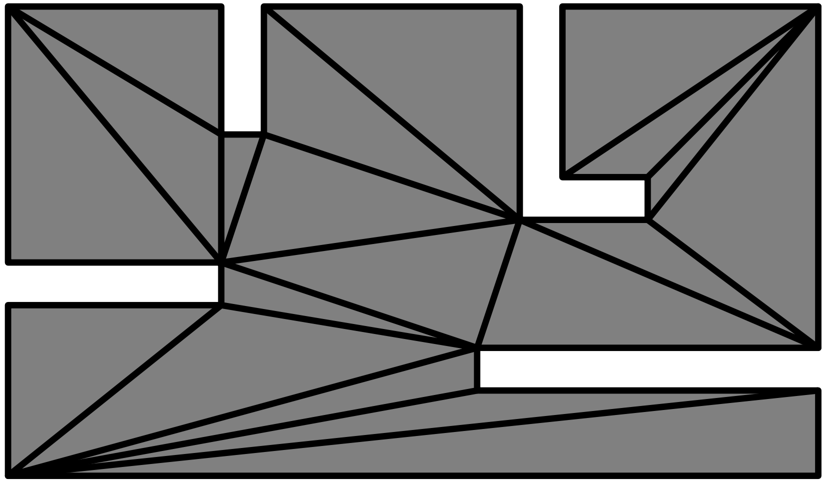


Problem Simplification

Observation: It is easy to guard a triangle

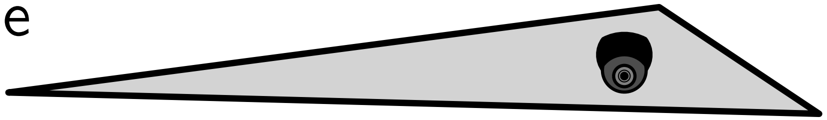


Idea: Decompose P into triangles and guard each of them

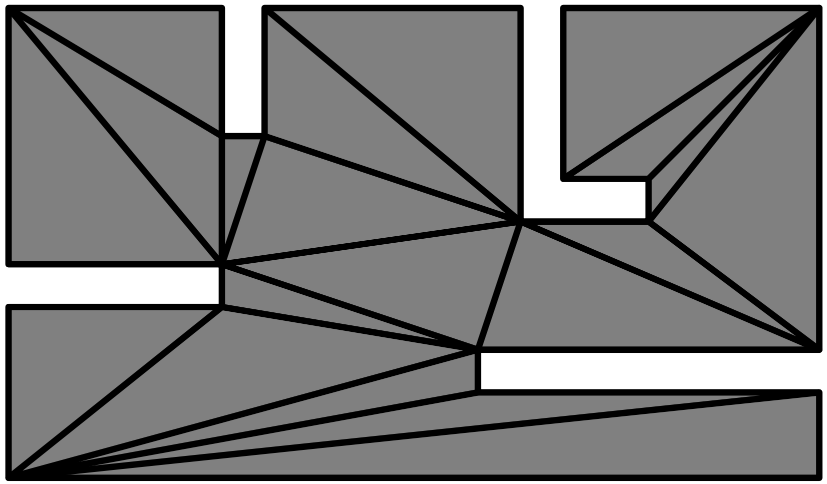


Problem Simplification

Observation: It is easy to guard a triangle



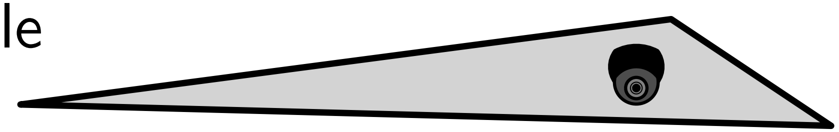
Idea: Decompose P into triangles and guard each of them



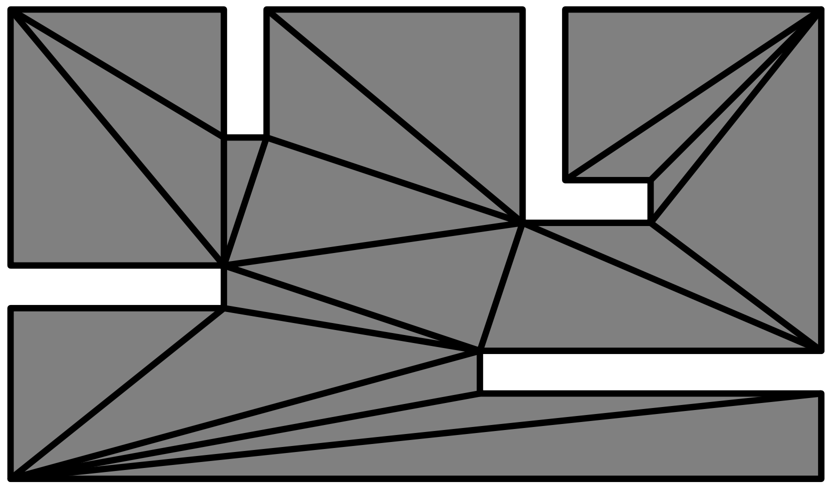
Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them

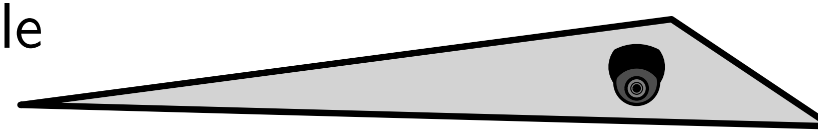


Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

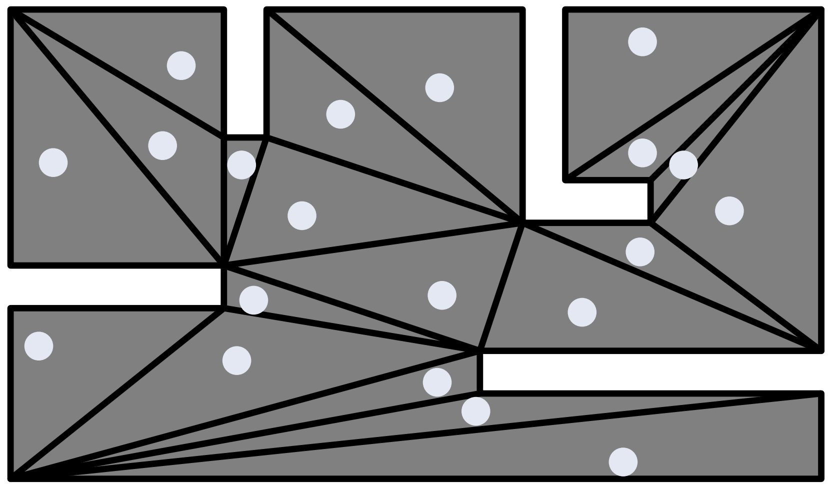
The proof implies a recursive $O(n^2)$-Algorithm!

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them

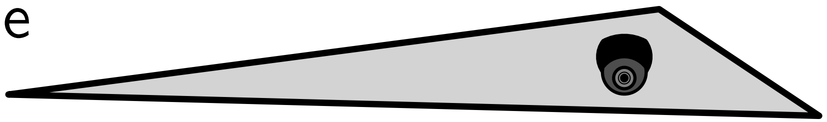


Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

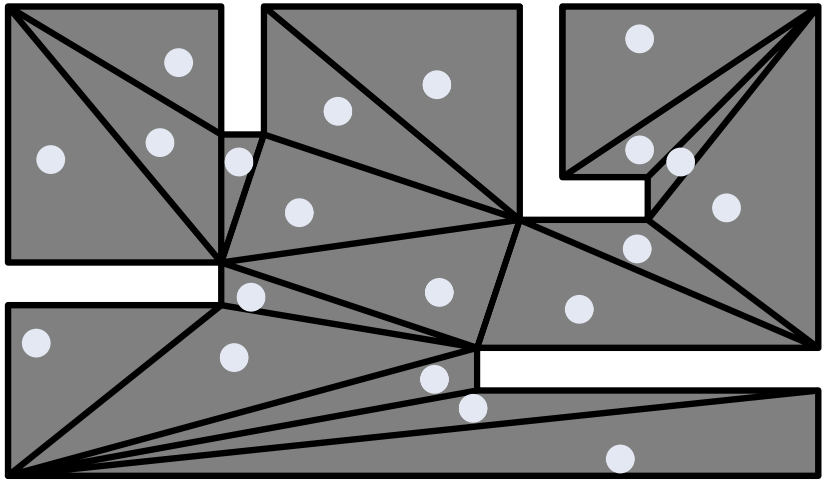
- P could be guarded by $n-2$ cameras placed in the triangles

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them



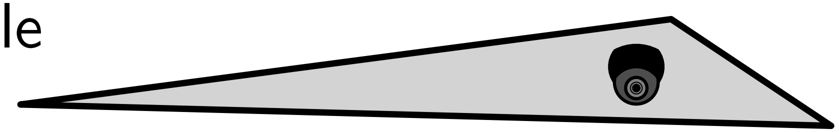
Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

- P could be guarded by $n-2$ cameras placed in the triangles

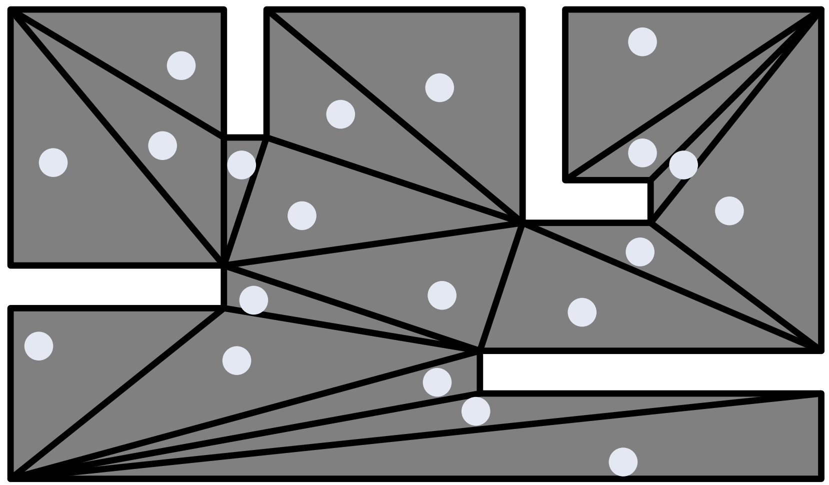
Can we do better?

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them

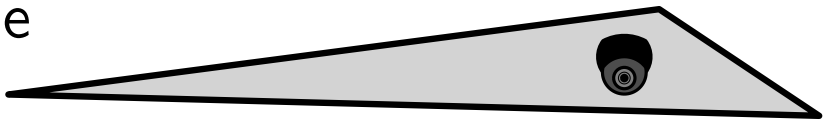


Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

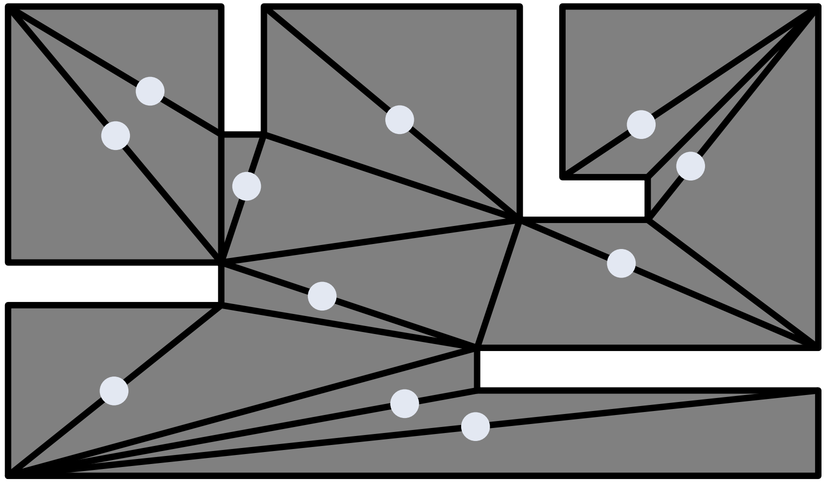
- P could be guarded by $n-2$ cameras placed in the triangles
- P can be guarded by $\approx n/2$ cameras placed on the diagonals

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them

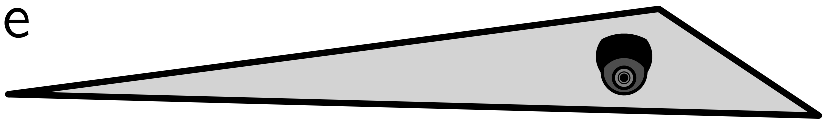


Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

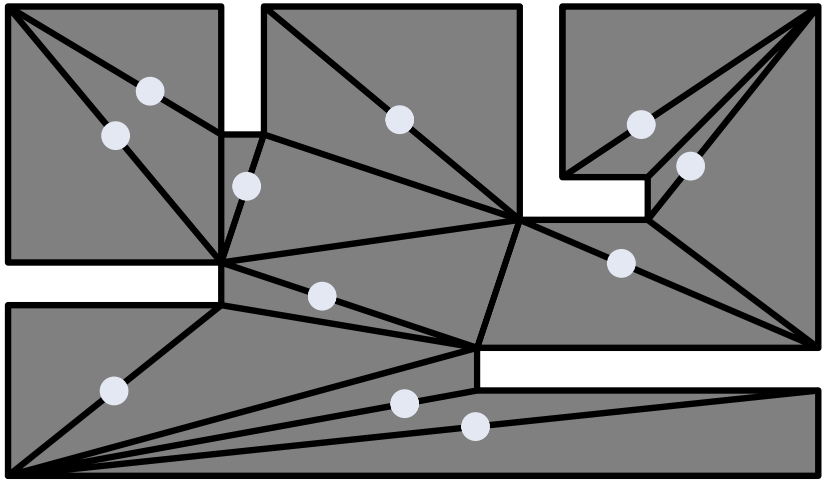
- P could be guarded by $n-2$ cameras placed in the triangles
- P can be guarded by $\approx n/2$ cameras placed on the diagonals

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them

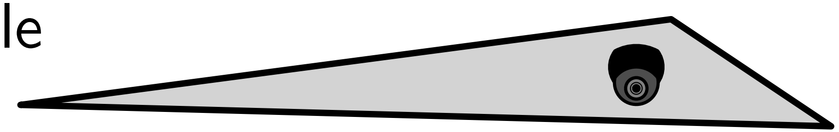


Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

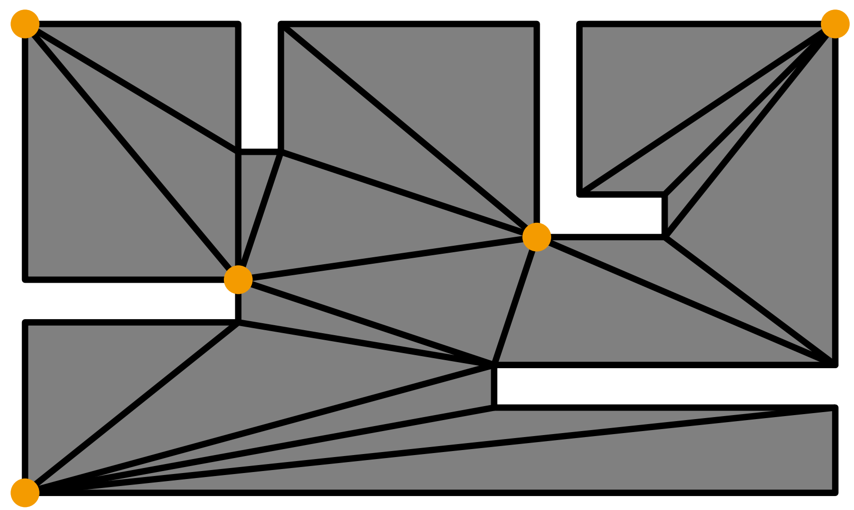
- P could be guarded by $n-2$ cameras placed in the triangles
- P can be guarded by $\approx n/2$ cameras placed on the diagonals
- P can be observed by even smaller number of cameras placed on the corners

Problem Simplification

Observation: It is easy to guard a triangle



Idea: Decompose P into triangles and guard each of them



Theorem 1: Each simple polygon with n corners admits a triangulation; any such triangulation contains exactly $n-2$ triangles.

- P could be guarded by $n-2$ cameras placed in the triangles
- P can be guarded by $\approx n/2$ cameras placed on the diagonals
- P can be observed by even smaller number of cameras placed on the corners

The Art-Gallery-Theorem [Chvátal '75]

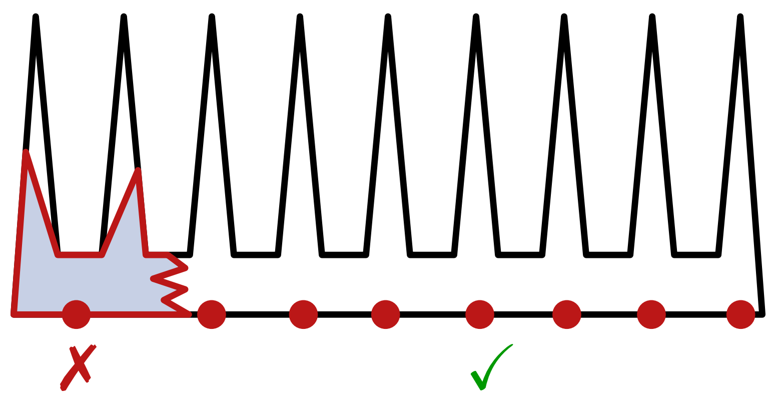
Theorem 2: For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are sometimes necessary and always sufficient to guard it.

The Art-Gallery-Theorem [Chvátal '75]

Theorem 2: For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are sometimes necessary and always sufficient to guard it.

Proof:

- Find a simple polygon with n corners that requires $\approx n/3$ cameras!



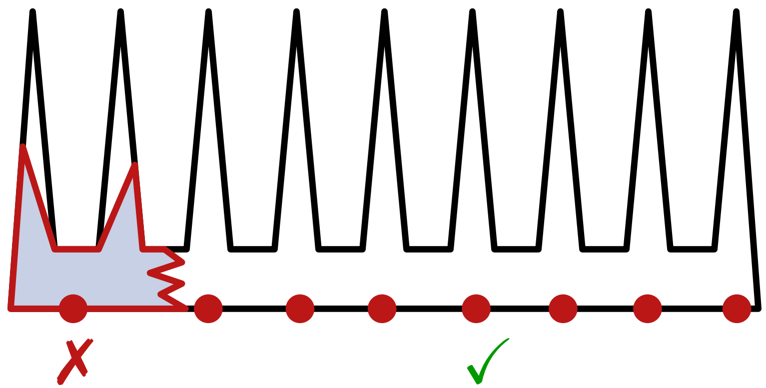
- Sufficiency on the board

The Art-Gallery-Theorem [Chvátal '75]

Theorem 2: For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are sometimes necessary and always sufficient to guard it.

Proof:

- Find a simple polygon with n corners that requires $\approx n/3$ cameras!



- Sufficiency on the board

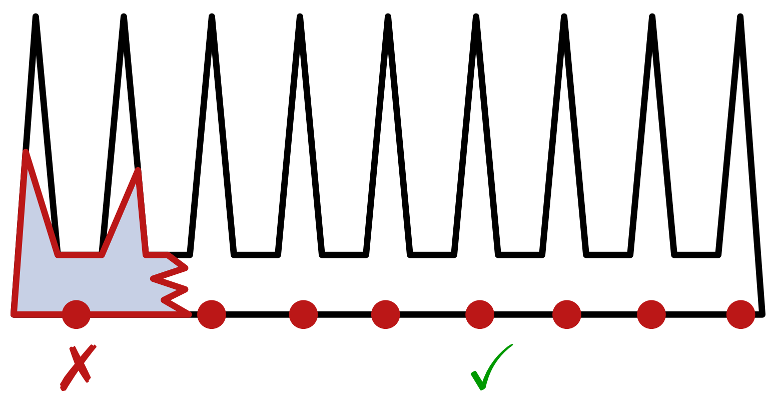
Conclusion: Given a triangulation, the $\lfloor n/3 \rfloor$ cameras that guard the polygon can be placed in $O(n)$ time.

The Art-Gallery-Theorem [Chvátal '75]

Theorem 2: For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are sometimes necessary and always sufficient to guard it.

Proof:

- Find a simple polygon with n corners that requires $\approx n/3$ cameras!



- Sufficiency on the board

Conclusion: Given a triangulation, the $\lfloor n/3 \rfloor$ cameras that guard the polygon can be placed in $O(n)$ time.

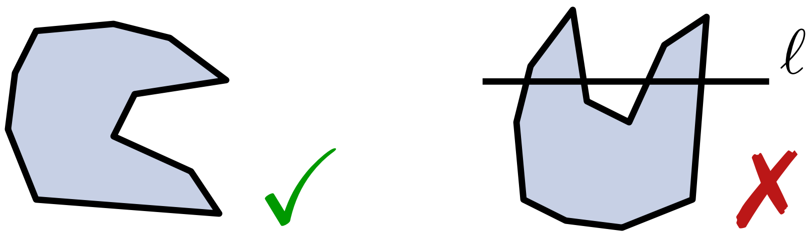
Can we do better than $O(n^2)$ described before?

Triangulation of Polygons

2-step process:

- Step 1: Decompose P into y-monotone polygons

Definition: A polygon is y-monotone, if for any horizontal line ℓ, the interection $\ell \cap P$ is connected.



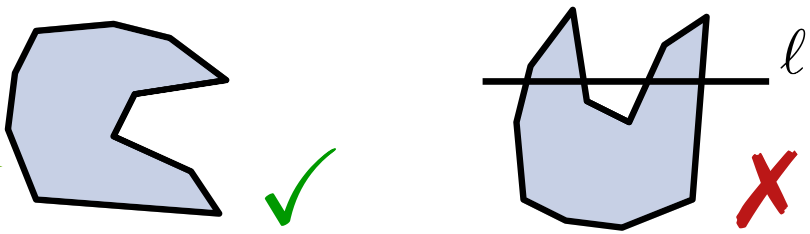
Triangulation of Polygons

2-step process:

- Step 1: Decompose P into y-monotone polygons

Definition: A polygon is y-monotone, if for any horizontal line ℓ, the interection $\ell \cap P$ is connected.

The two paths from the topmost to the bottomost point bounding the polygon, never go upward



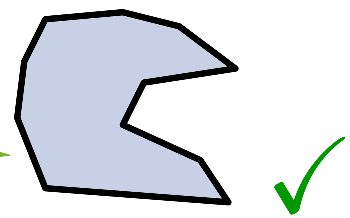
Triangulation of Polygons

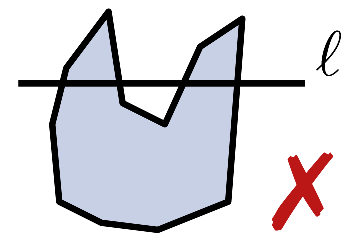
2-step process:

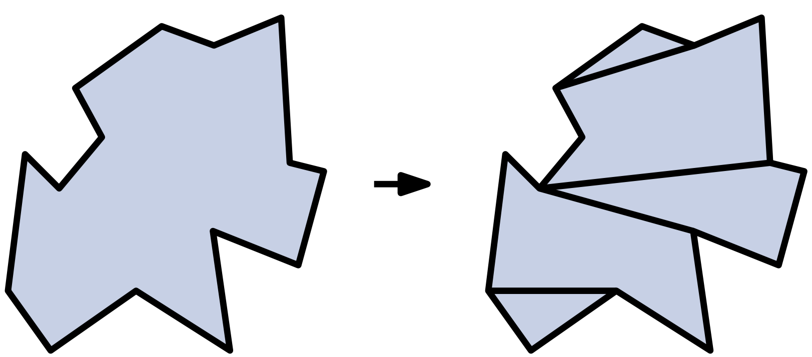
- Step 1: Decompose P into y-monotone polygons

Definition: A polygon is y-monotone, if for any horizontal line ℓ, the interection $\ell \cap P$ is connected.

The two paths from the topmost to the bottomost point bounding the polygon, never go upward





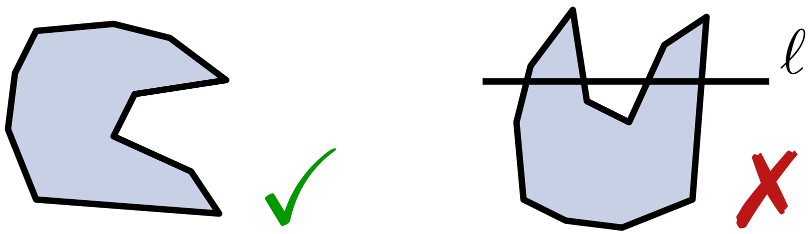


Triangulation of Polygons

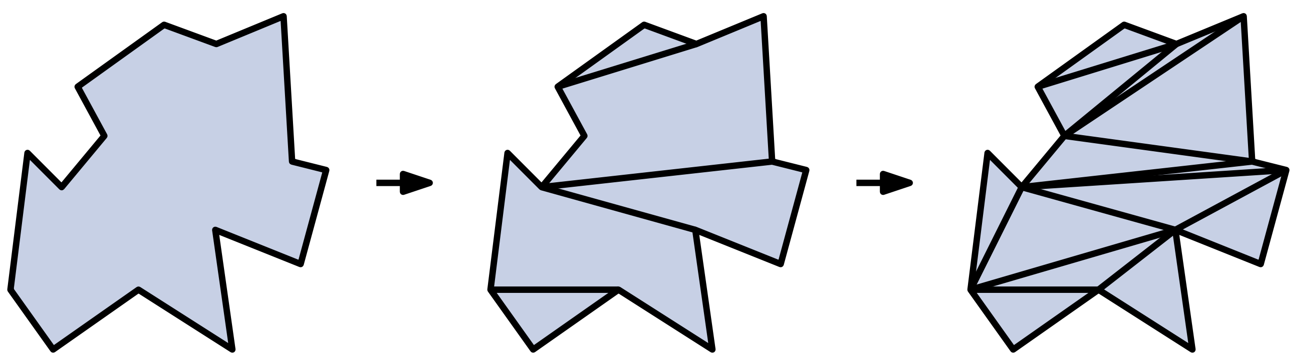
2-step process:

- Step 1: Decompose P into y-monotone polygons

Definition: A polygon is y-monotone, if for any horizontal line ℓ, the interection $\ell \cap P$ is connected.

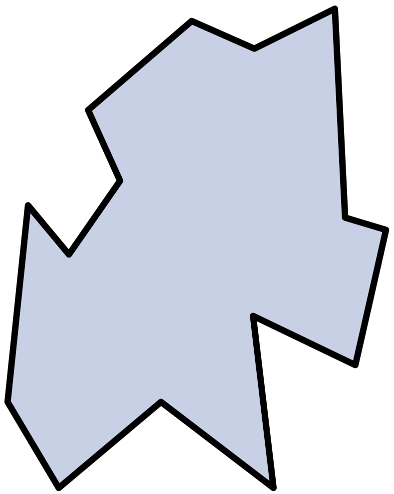


- Step 2: Triangulate the resulting y-monotone polygons



Partition into y-monotone Polygons

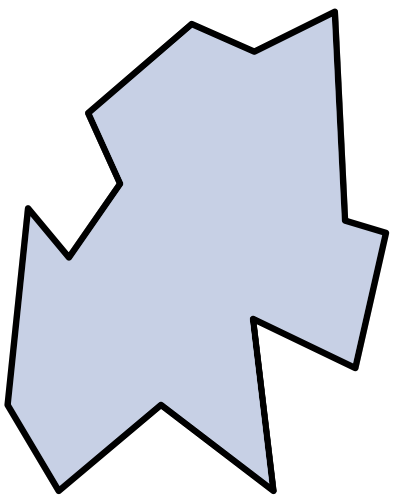
Idea: Five different types of vertices



Partition into y-monotone Polygons

Idea: Five different types of vertices

– *Turn vertices*:

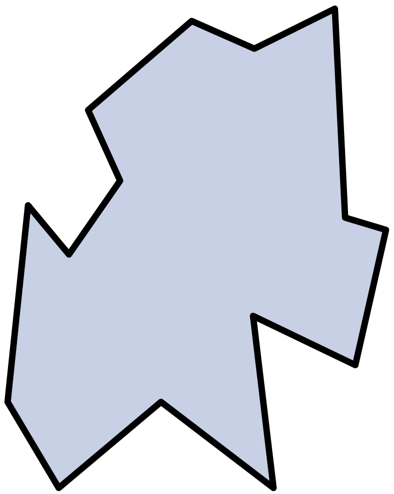


– *regular* vertices

Partition into y-monotone Polygons

Idea: Five different types of vertices

– *Turn vertices*:
vertical change in direction



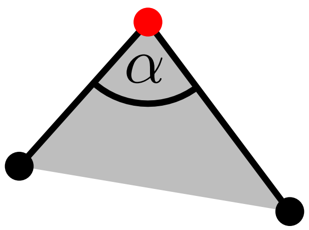
– *regular* vertices

Partition into y-monotone Polygons

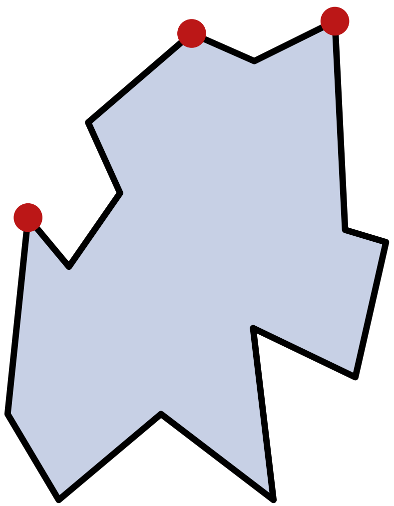
Idea: Five different types of vertices

– *Turn vertices*:
vertical change in direction

- *start* vertices



if $\alpha < 180°$



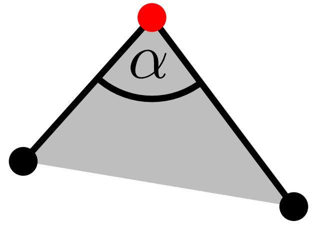
– *regular* vertices

Partition into y-monotone Polygons

Idea: Five different types of vertices

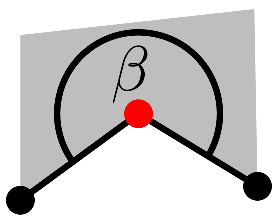
– *Turn vertices*:
  vertical change in direction

- *start* vertices

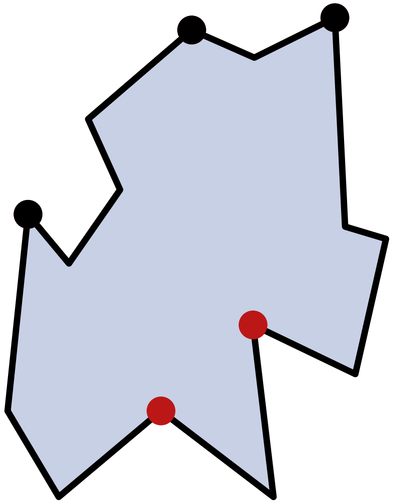


if $\alpha < 180°$

- *split* vertices



if $\beta > 180°$

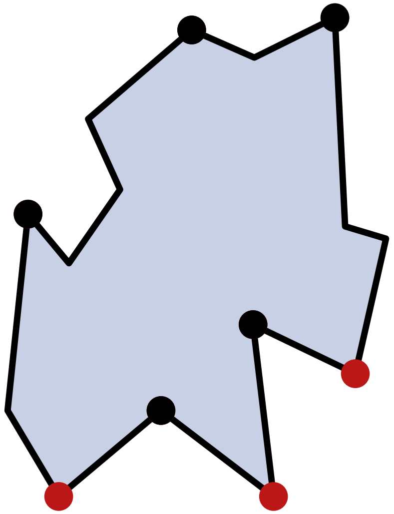


– *regular* vertices

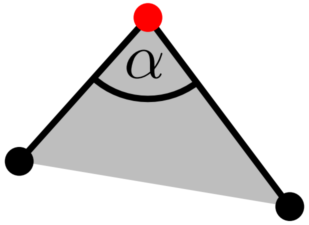
Partition into y-monotone Polygons

Idea: Five different types of vertices

– *Turn vertices*:
vertical change in direction

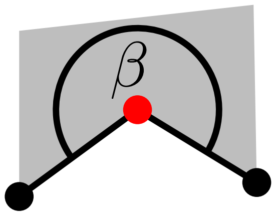


- *start* vertices



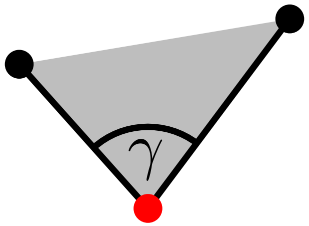
if $\alpha < 180°$

- *split* vertices



if $\beta > 180°$

- *end* vertices



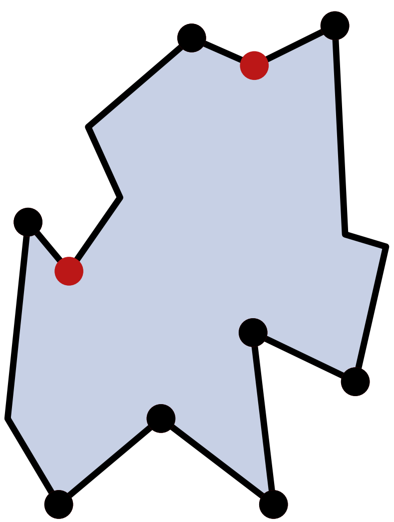
if $\gamma < 180°$

– *regular* vertices

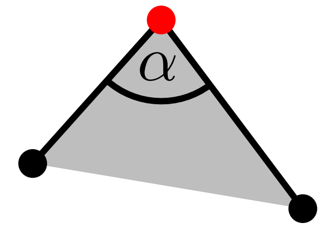
Partition into y-monotone Polygons

Idea: Five different types of vertices

– *Turn vertices*:
vertical change in direction

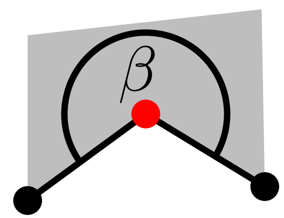


- *start* vertices



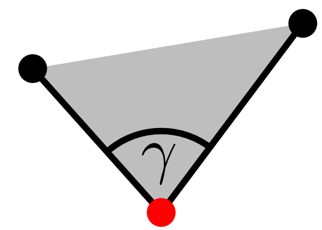
if $\alpha < 180°$

- *split* vertices



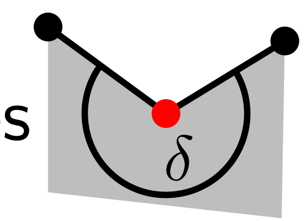
if $\beta > 180°$

- *end* vertices



if $\gamma < 180°$

- *merge* vertices



if $\delta > 180°$

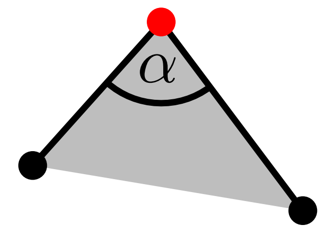
– *regular* vertices

Partition into y-monotone Polygons

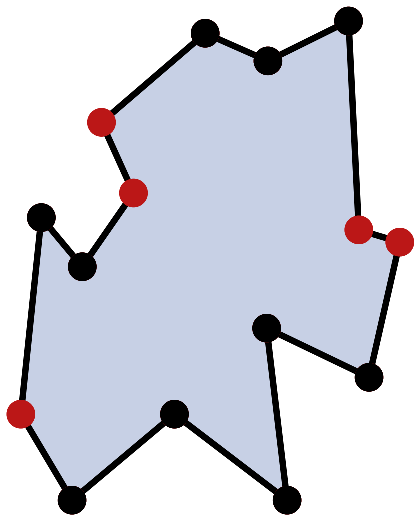
Idea: Five different types of vertices

– *Turn vertices*:
vertical change in direction

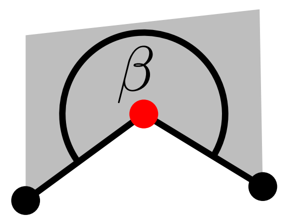
- *start* vertices



if $\alpha < 180°$

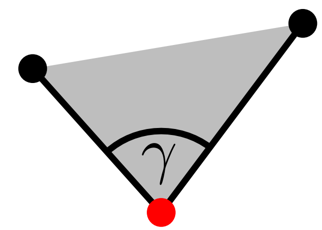


- *split* vertices



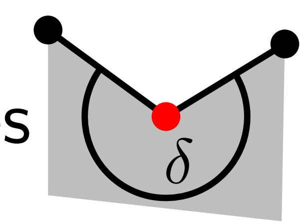
if $\beta > 180°$

- *end* vertices



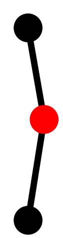
if $\gamma < 180°$

- *merge* vertices



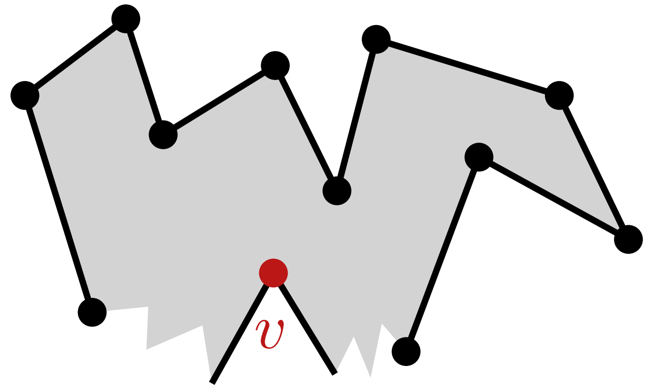
if $\delta > 180°$

– *regular* vertices



Ideas for Sweep-Line-Algorithm

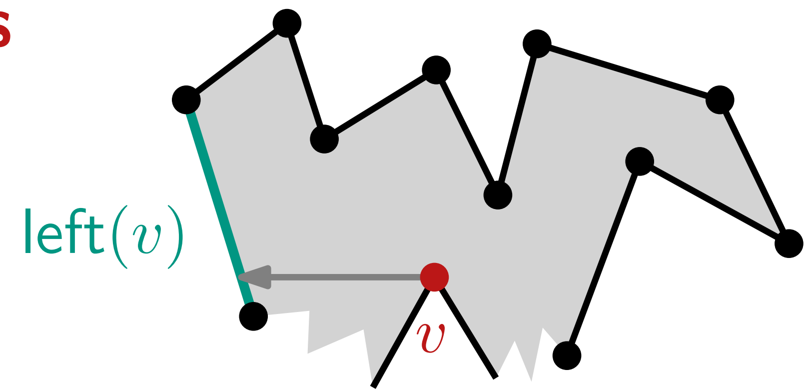
1) Diagonals for the split vertices



Ideas for Sweep-Line-Algorithm

1) Diagonals for the split vertices

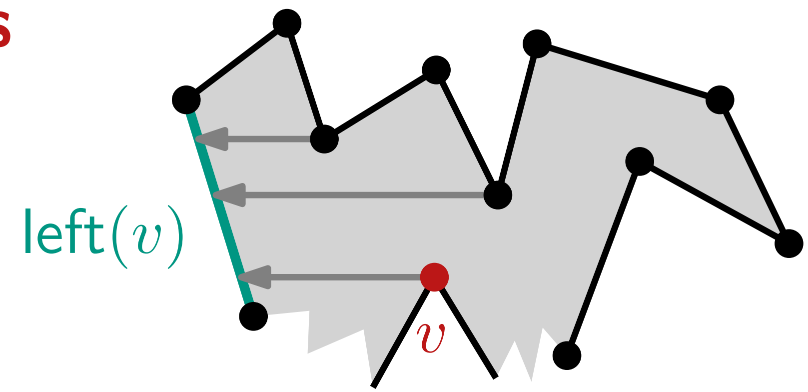
- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ



Ideas for Sweep-Line-Algorithm

1) Diagonals for the split vertices

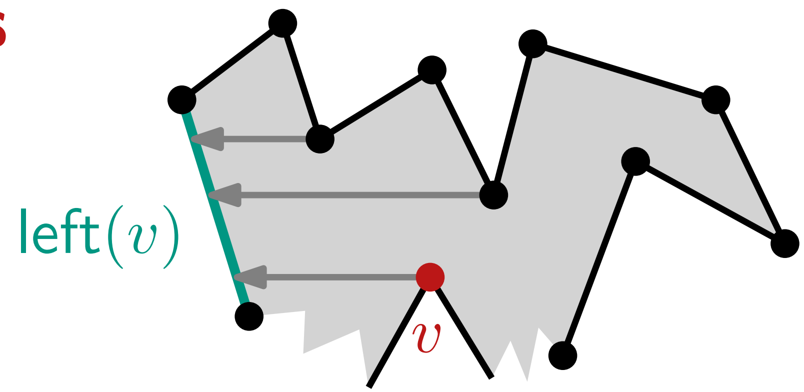
- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ



Ideas for Sweep-Line-Algorithm

1) Diagonals for the split vertices

- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ

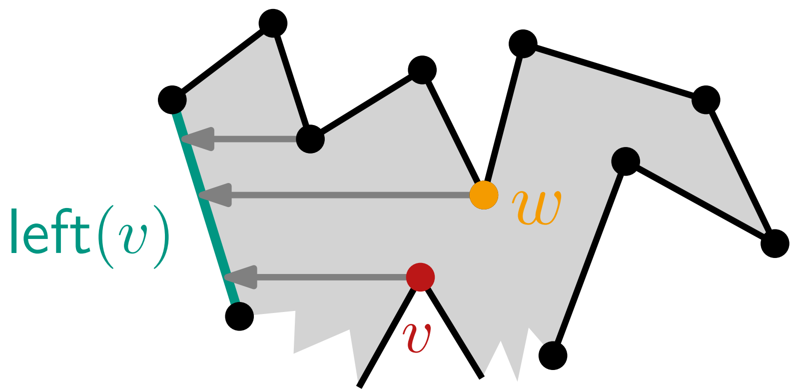


- connect split vertex v to the nearest vertex w above v, such that left$(w) =$ left(v)

Ideas for Sweep-Line-Algorithm

1) Diagonals for the split vertices

- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ



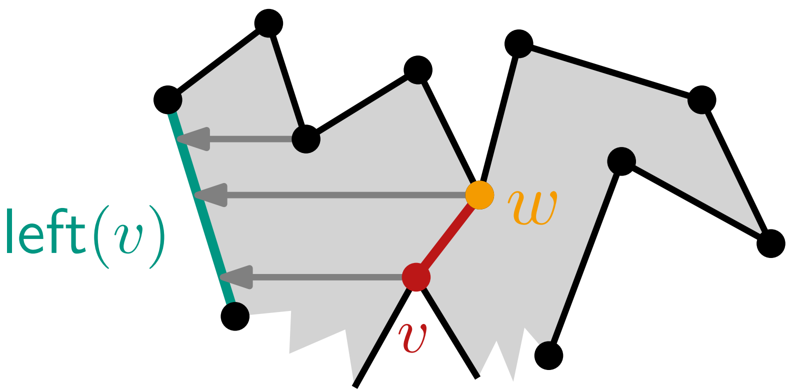
- connect split vertex v to the nearest vertex w above v, such that left(w) = left(v)

Ideas for Sweep-Line-Algorithm

Karlsruhe Institute of Technology

1) Diagonals for the split vertices

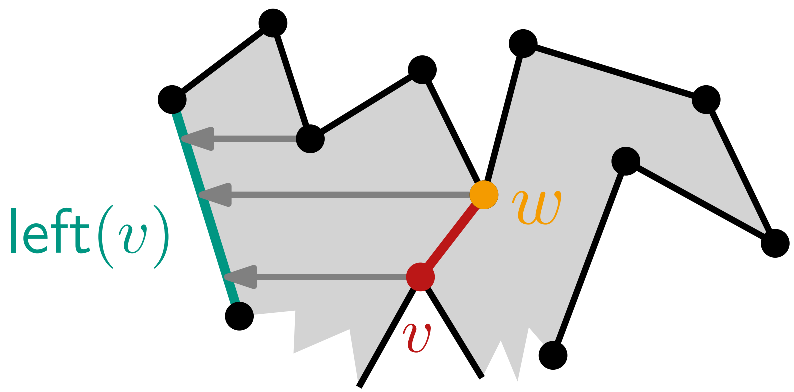
- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ



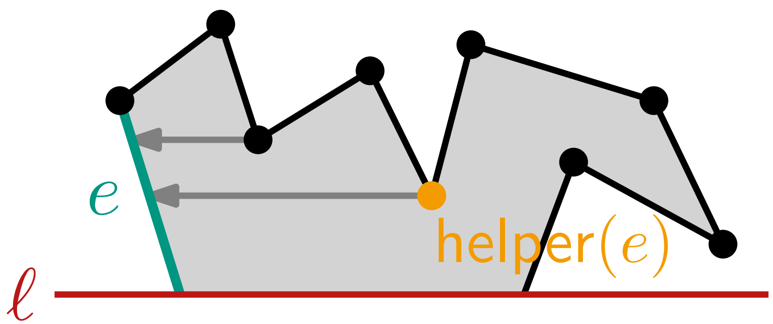
- connect split vertex v to the nearest vertex w above v, such that left$(w) =$ left(v)

Ideas for Sweep-Line-Algorithm

1) Diagonals for the split vertices



- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ
- connect split vertex v to the nearest vertex w above v, such that left$(w) =$ left(v)
- for each edge e save the bottommost vertex w such that left$(w) = e$; notation helper(e) := w

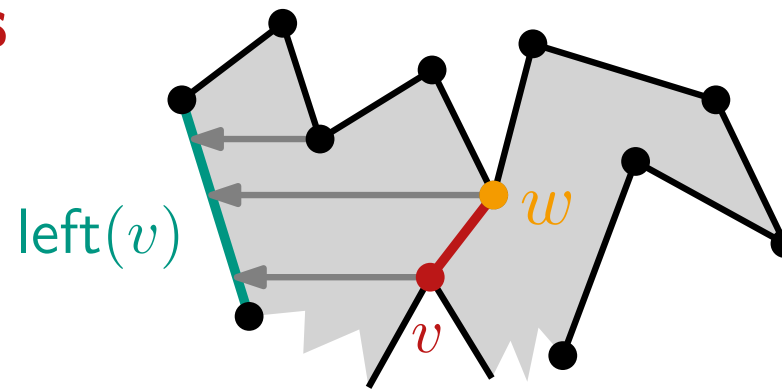


Ideas for Sweep-Line-Algorithm

Karlsruhe Institute of Technology

1) Diagonals for the split vertices

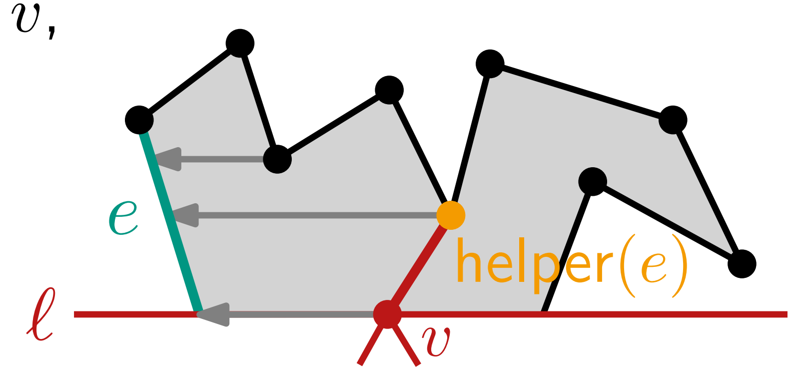
- compute for each vertex v its left adjacent edge left(v) with respect to the horizontal sweep line ℓ



- connect split vertex v to the nearest vertex w above v, such that left(w) = left(v)

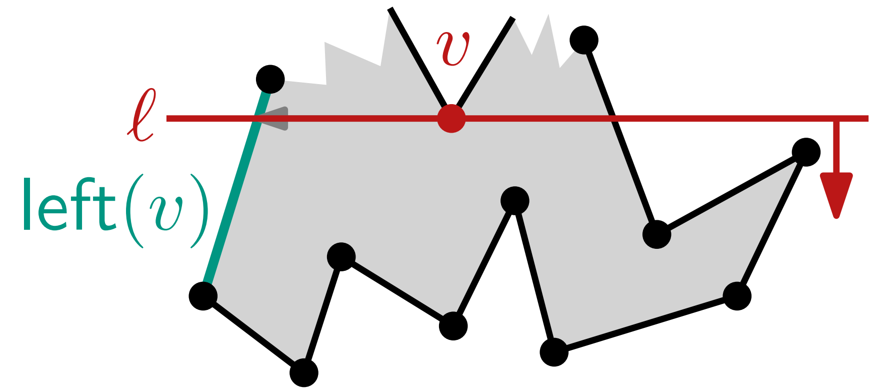
- for each edge e save the bottommost vertex w such that left$(w) = e$; notation helper(e) := w

- when ℓ passes through a split vertex v, we connect v with helper(left(v))



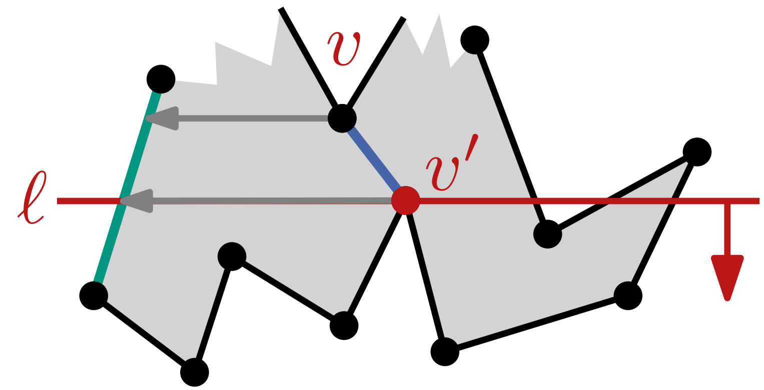
2) Diagonals for merge vertices

- when the vertex v is reached, we set helper(left(v)) $= v$



2) Diagonals for merge vertices

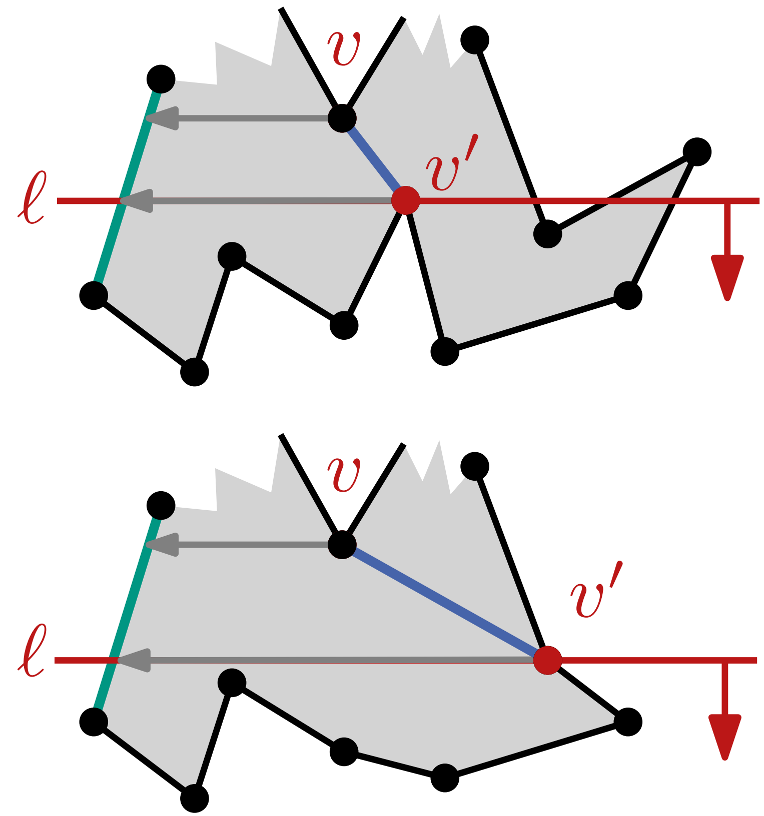
- when the vertex v is reached, we set helper(left$(v)) = v$
- when we reach a split vertex v' such that left(v') = left(v) the diagonal (v, v') is introduced



Ideas for Sweep-Line-Algorithm

2) Diagonals for merge vertices

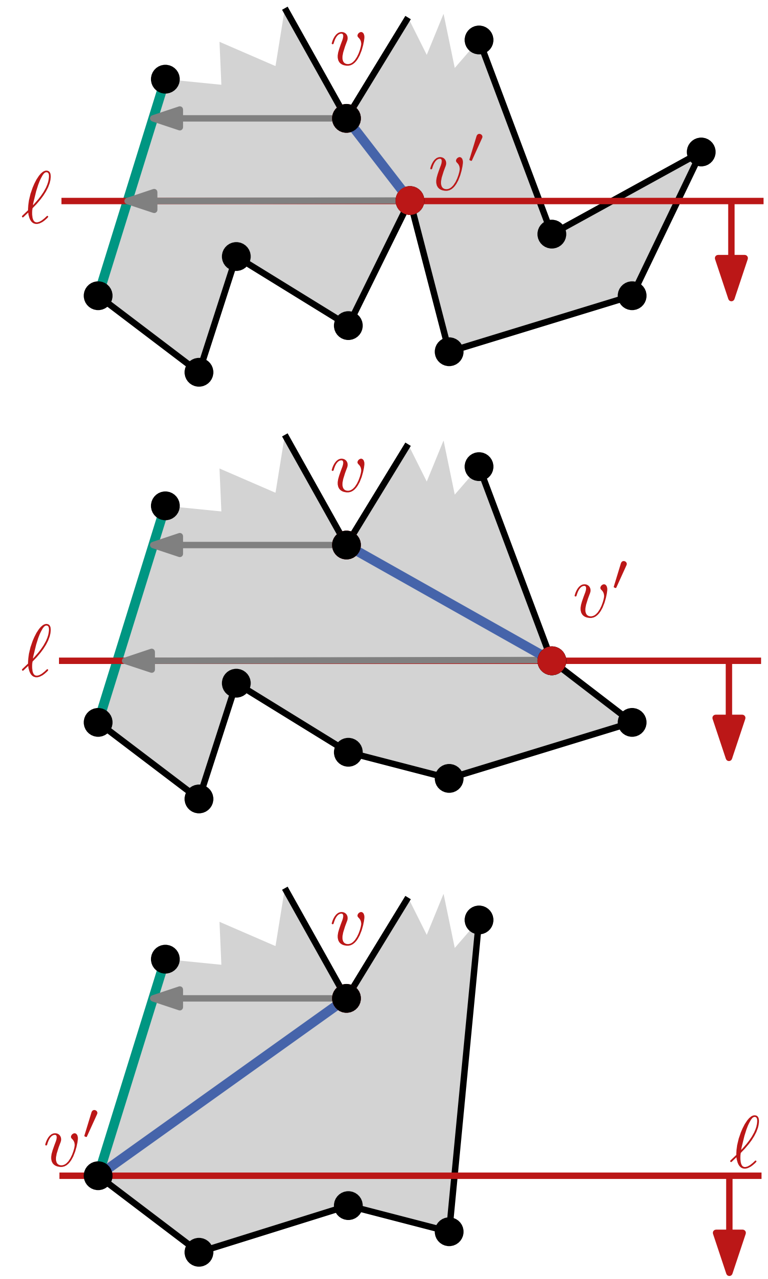
- when the vertex v is reached, we set helper(left$(v)) = v$
- when we reach a split vertex v' such that left(v') = left(v) the diagonal (v, v') is introduced
- in case we reach a regular vertex v' such that helper(left(v')) is v the diagonal (v, v') is introduced



Ideas for Sweep-Line-Algorithm

2) Diagonals for merge vertices

- when the vertex v is reached, we set helper(left(v)) $= v$
- when we reach a split vertex v' such that left(v') $=$ left(v) the diagonal (v, v') is introduced
- in case we reach a regular vertex v' such that helper(left(v')) is v the diagonal (v, v') is introduced
- if the end of v' of left(v) is reached, then the diagonal (v, v') is introduced



Algorithm MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
    $v \leftarrow \mathcal{Q}$.nextVertex()
    $\mathcal{Q}$.deleteVertex(v)
    handleVertex(v)
return $\mathcal{D}$

Algorithm MakeMonotone(P)

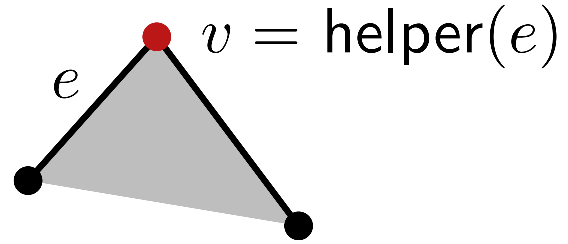
MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
 $v \leftarrow \mathcal{Q}$.nextVertex()
 $\mathcal{Q}$.deleteVertex(v)
 handleVertex(v)

return $\mathcal{D}$

handleStartVertex(vertex v)

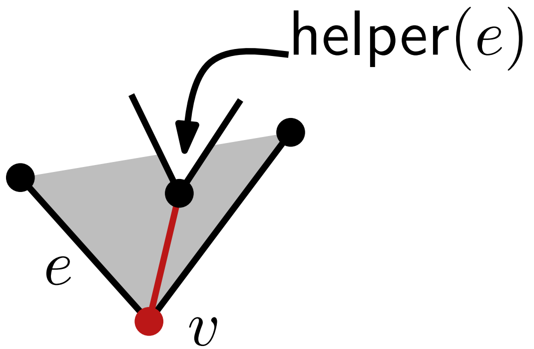
$\mathcal{T} \leftarrow$ add the left edge e
helper(e) $\leftarrow v$



Algorithm MakeMonotone(P)

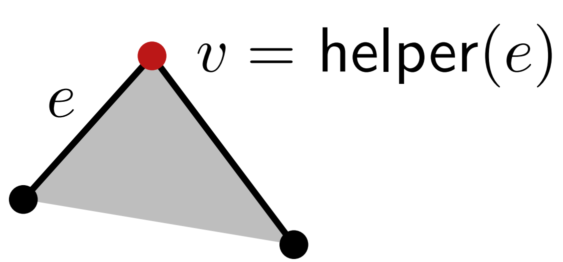
MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$



handleStartVertex(vertex v)

$\mathcal{T} \leftarrow$ add the left edge e
helper(e) $\leftarrow v$



handleEndVertex(vertex v)

$e \leftarrow$ left edge
if isMergeVertex(helper(e)) **then**
  $\mathcal{D} \leftarrow$ add edge (helper(e), v)
remove e from $\mathcal{T}$

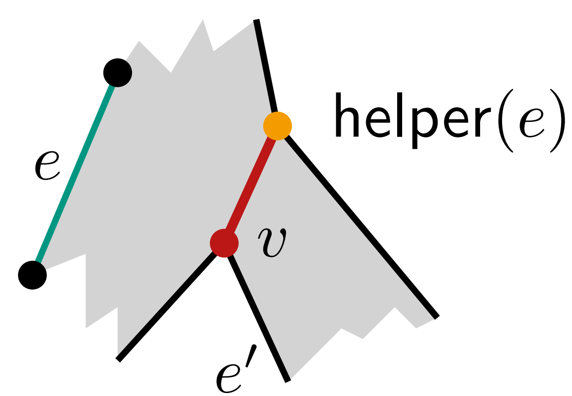
Algorithm MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$

handleSplitVertex(vertex v)

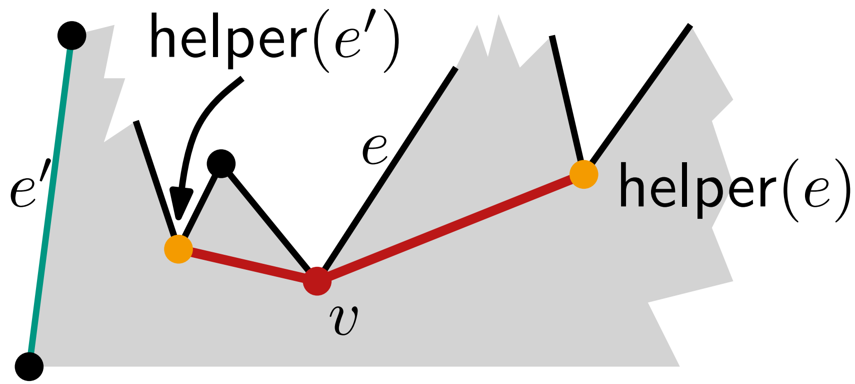
$e \leftarrow$ Edge to the left of v in $\mathcal{T}$
$\mathcal{D} \leftarrow$ add edge (helper(e), v)
helper(e) $\leftarrow v$
$\mathcal{T} \leftarrow$ add the right edge e' of v
helper(e') $\leftarrow v$



Algorithm MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$



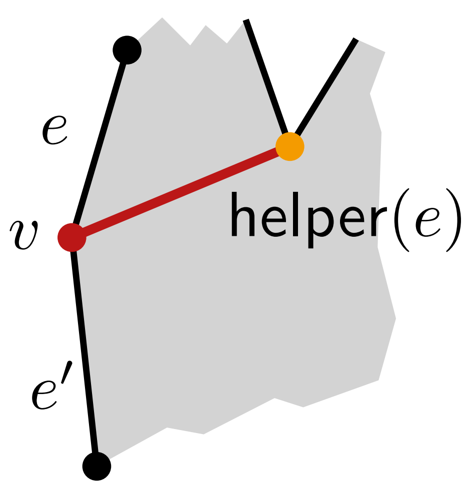
handleMergeVertex(vertex v)

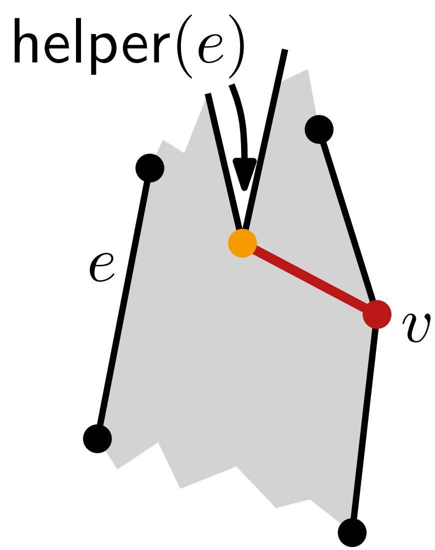
$e \leftarrow$ right edge
if isMergeVertex(helper(e)) **then**
  $\mathcal{D} \leftarrow$ add edge (helper(e), v)
remove e from $\mathcal{T}$
$e' \leftarrow$ edge to the left of v in $\mathcal{T}$
if isMergeVertex(helper(e'))
 then
  $\mathcal{D} \leftarrow$ add edge (helper(e'), v)
helper(e') $\leftarrow v$

Algorithm MakeMonotone(P)

MakeMonotone(Polygon P)

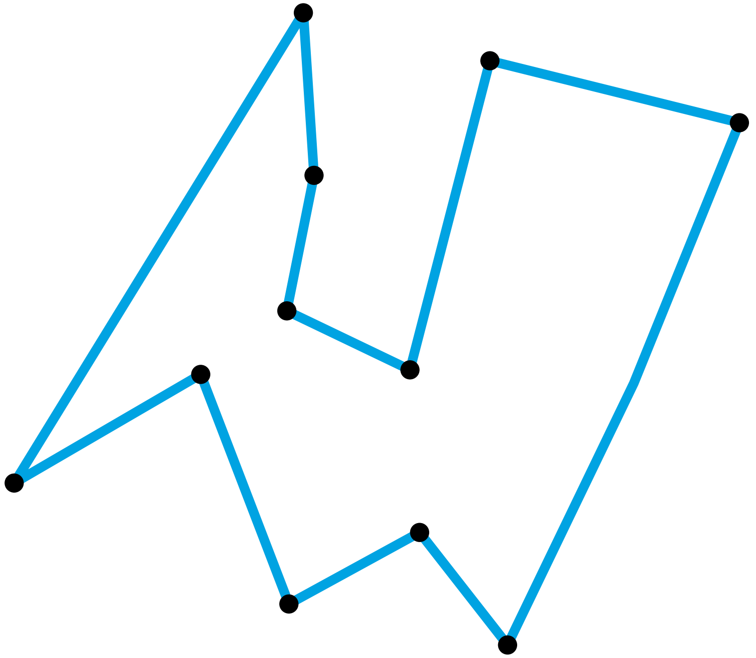
$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
    $v \leftarrow \mathcal{Q}$.nextVertex()
    $\mathcal{Q}$.deleteVertex(v)
    handleVertex(v)
return $\mathcal{D}$



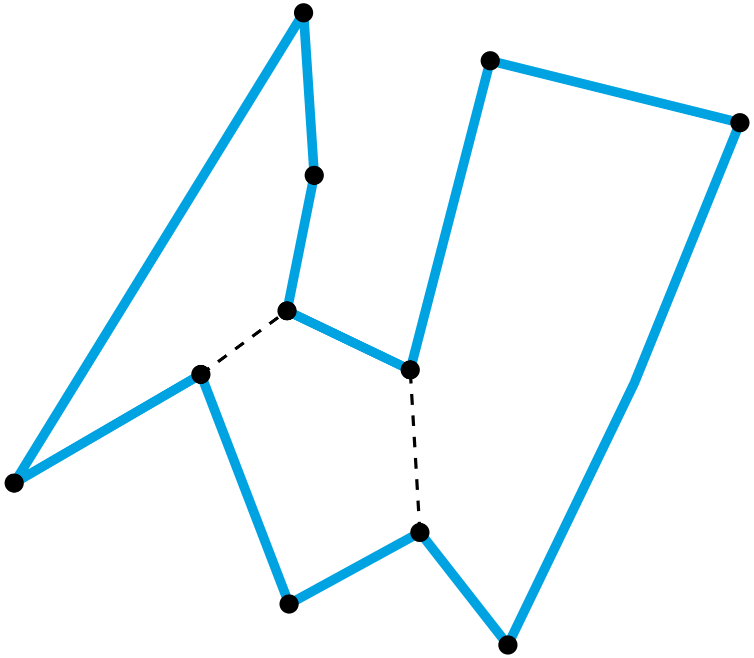


handleRegularVertex(vertex v)
if P lies locally to the right of v
then
    $e, e' \leftarrow$ above, below edge
    if isMergeVertex(helper(e))
    then
        $\mathcal{D} \leftarrow$ add edge (helper$(e), v)$
    remove e from $\mathcal{T}$
    $\mathcal{T} \leftarrow$ add e'; helper$(e') \leftarrow v$
else
    $e \leftarrow$ edge to the left of v
    ~~add e to $\mathcal{T}$~~
    if isMergeVertex(helper(e))
    then

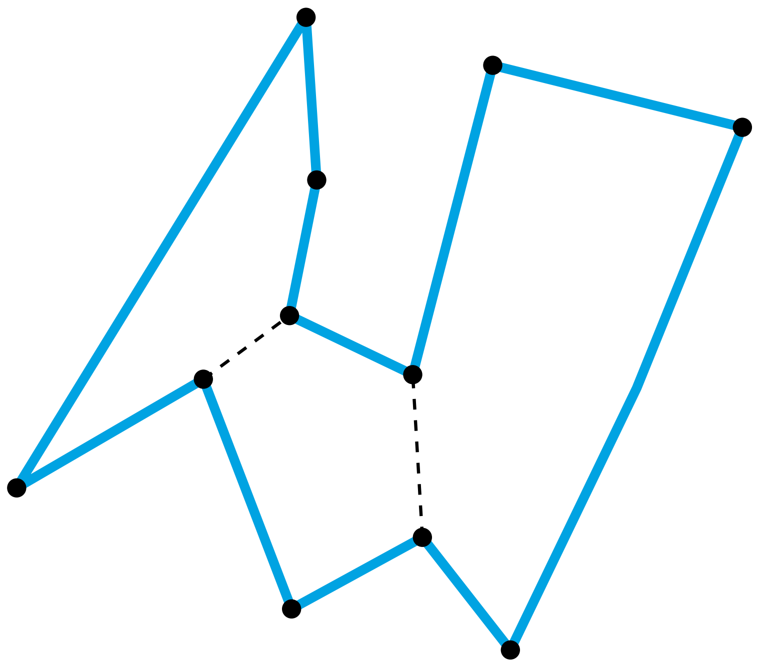
Insertion Diagonals



Insertion Diagonals

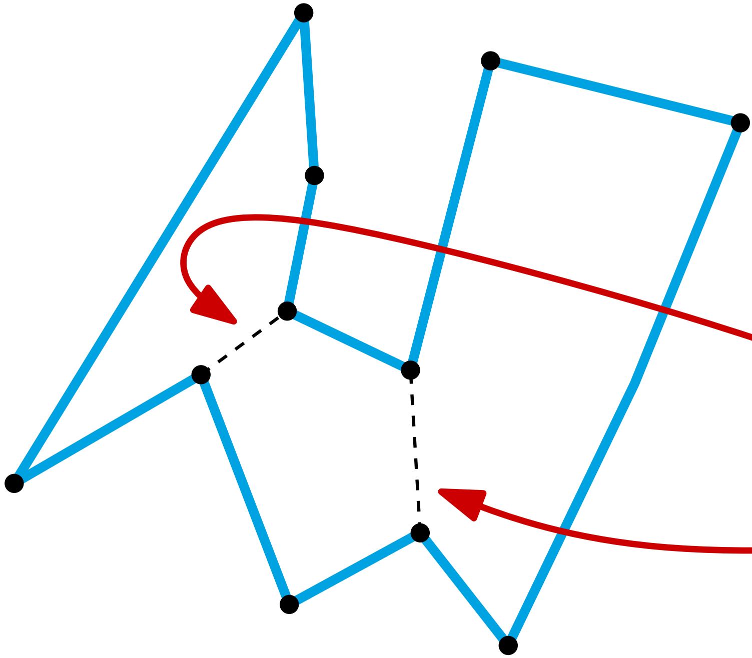


Insertion Diagonals



Data structure: Doubly-connected edge list (DCEL)

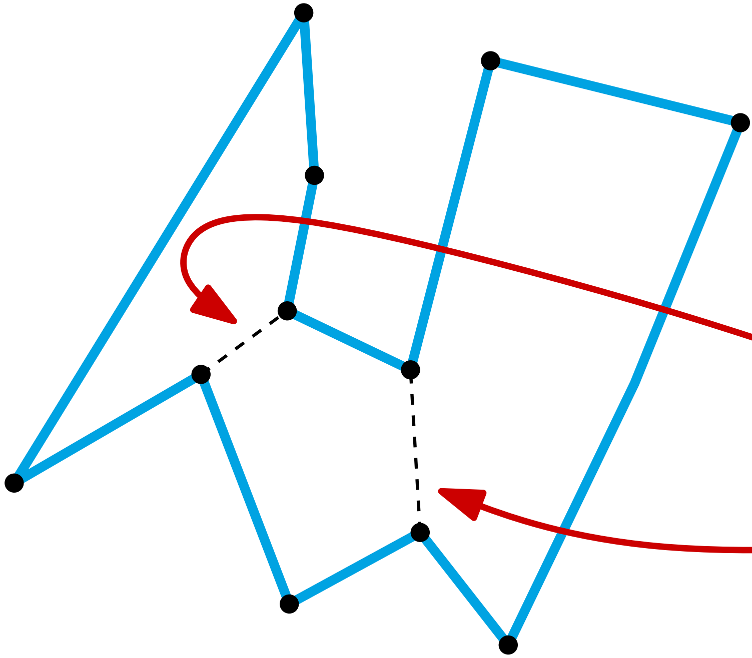
Insertion Diagonals



Claim:
Insertion of diagonals in $O(1)$ time.

Data structure: Doubly-connected edge list (DCEL)

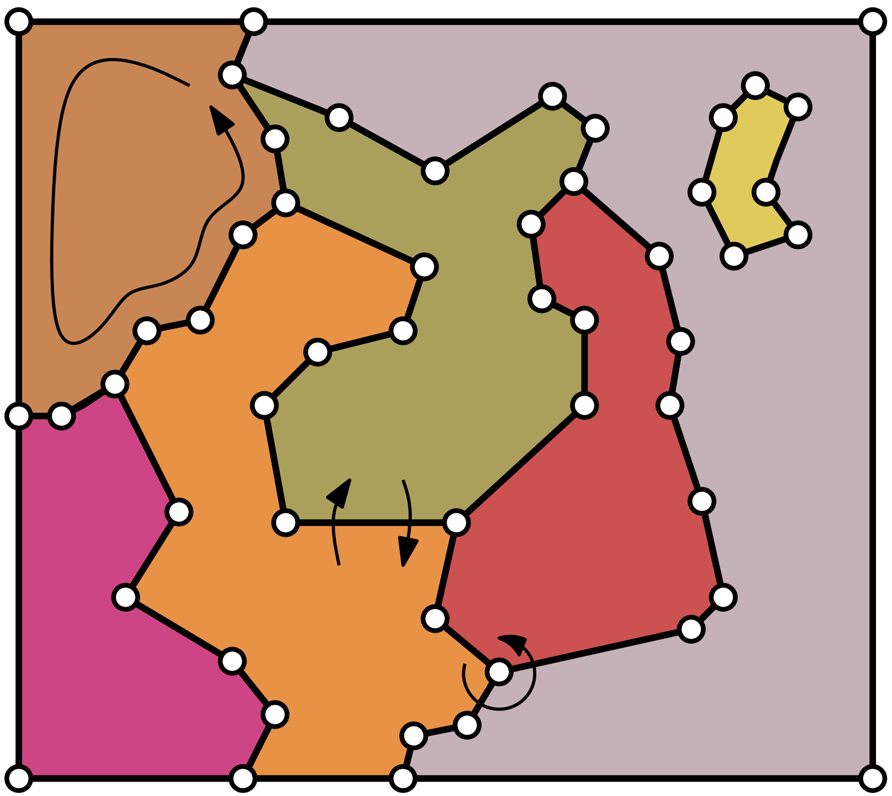
Insertion Diagonals



Claim:
Insertion of diagonals in $O(1)$ time.

Data structure: Doubly-connected edge list (DCEL)

Doubly Connected Edge List



- Map corresponds with subdivision of plane into polygons.
- Subdivision corresponds with embedding of planar graph with
  - vertices
  - edges
  - faces

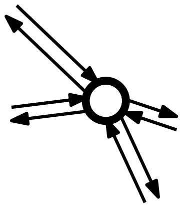
Which operations should be supported by the data structure?

- Traverse edges of face.
- Go from face to face by edges.
- Traverse neighboring vertices in cyclic order.

Doubly Connected Edge List(DCEL)

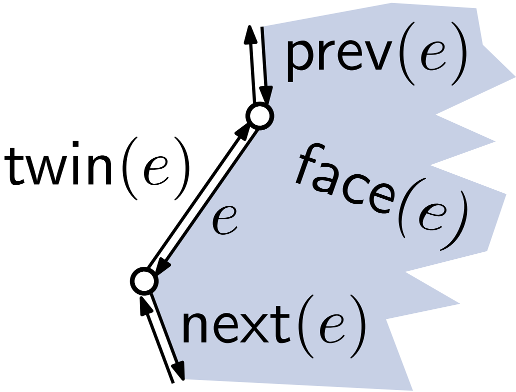
Ingredients:

- Vertices



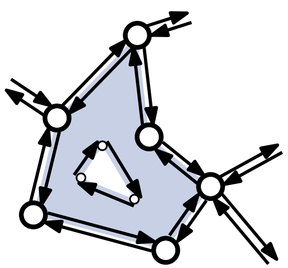
- Coordinates $(x(v), y(v))$
- (first) outgoing edge

- Edge $=$ two half-edges



- Vertex origin(v)
- Opposite edge twin(e)
- Predecessor prev(e) & Successor next(e)
- incident face

- Faces

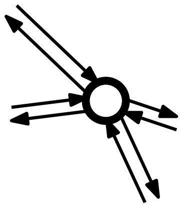


- Bounding edges for outer face.
- Edge list inner(f) for holes.

Doubly Connected Edge List (DCEL)

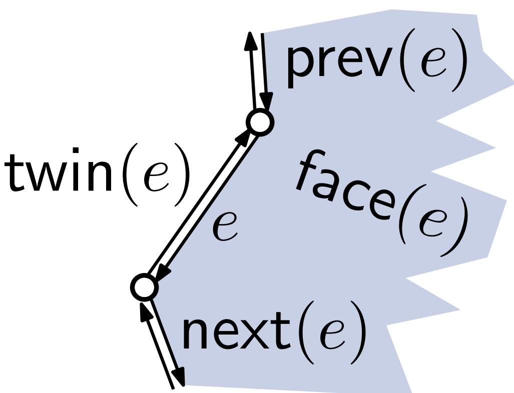
Ingredients:

- Vertices



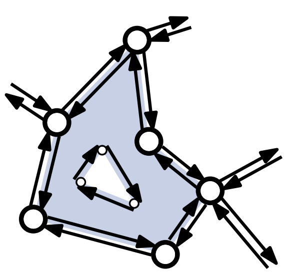
- Coordinates $(x(v), y(v))$
- (first) outgoing edge

- Edge $=$ two half-edges



- Vertex origin(v)
- Opposite edge twin(e)
- Predecessor prev(e) & Successor next(e)
- incident face

- Faces

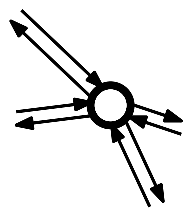


- Bounding edges for outer face.
- Edge list inner(f) for holes.

Doubly Connected Edge List (DCEL)

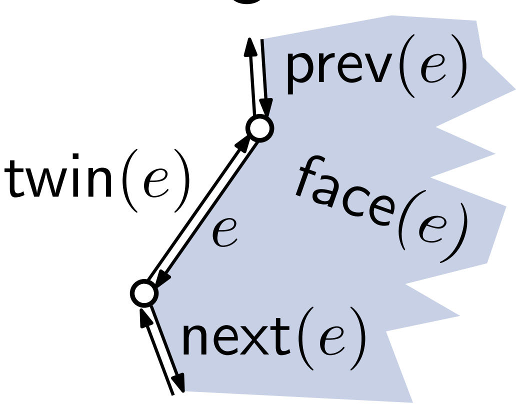
Ingredients:

- Vertices



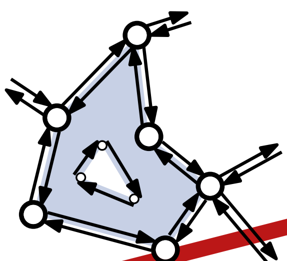
  - Coordinates $(x(v), y(v))$
  - (first) outgoing edge

- Edge $=$ two half-edges



  - Vertex origin(v)
  - Opposite edge twin(e)
  - Predecessor prev(e) & Successor next(e)
  - incident face

- ~~Faces~~

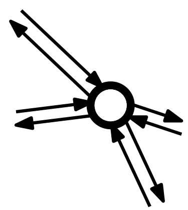


  - ~~Bounding edges for outer face.~~
  - ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

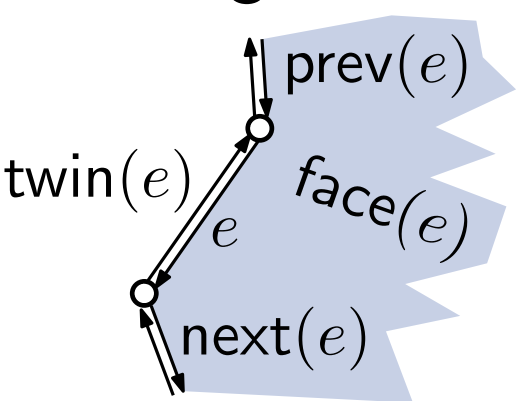
Ingredients:

- Vertices



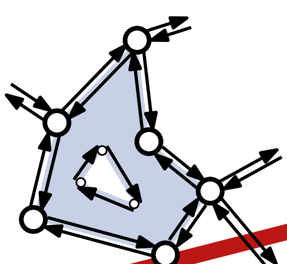
  - Coordinates $(x(v), y(v))$
  - (first) outgoing edge

- Edge $=$ two half-edges



  - Vertex origin(v)
  - Opposite edge twin(e)
  - Predecessor prev(e) & Successor next(e)
  - ~~incident face~~

- ~~Faces~~

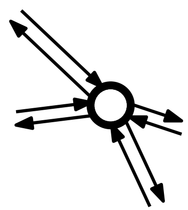


  - ~~Bounding edges for outer face.~~
  - ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

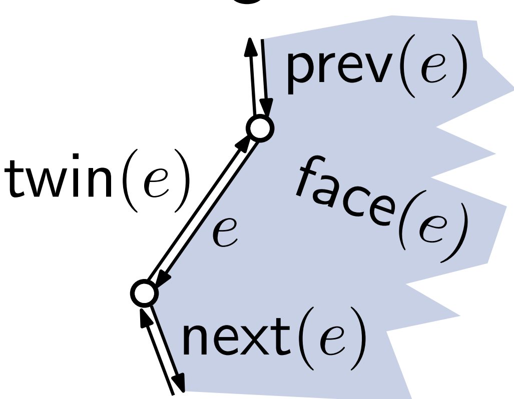
Ingredients:

- Vertices



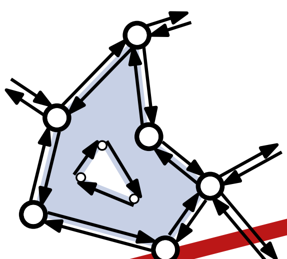
  - Coordinates $(x(v), y(v))$ ✓
  - (first) outgoing edge

- Edge $=$ two half-edges



  - Vertex origin(v)
  - Opposite edge twin(e)
  - Predecessor prev(e) & Successor next(e)
  - ~~incident face~~

- ~~Faces~~



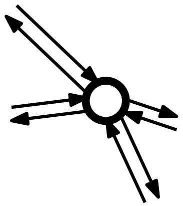
  - ~~Bounding edges for outer face.~~
  - ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

Karlsruhe Institute of Technology

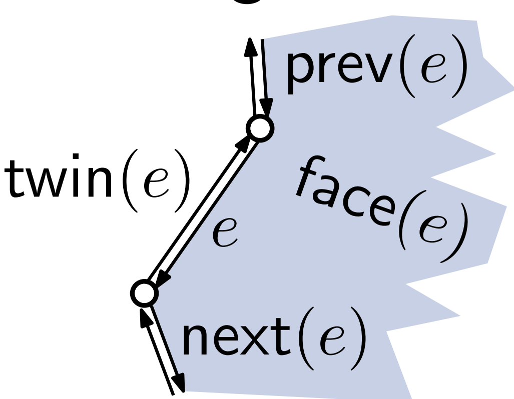
Ingredients:

- Vertices



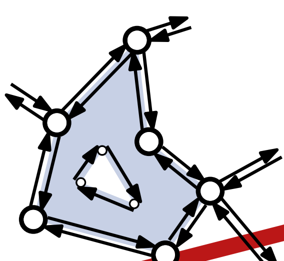
- Coordinates $(x(v), y(v))$ ✓
- (first) outgoing edge ✓

- Edge $=$ two half-edges



- Vertex origin(v)
- Opposite edge twin(e)
- Predecessor prev(e) & Successor next(e)
- ~~incident face~~

- ~~Faces~~

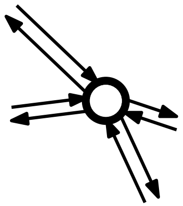


- ~~Bounding edges for outer face.~~
- ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

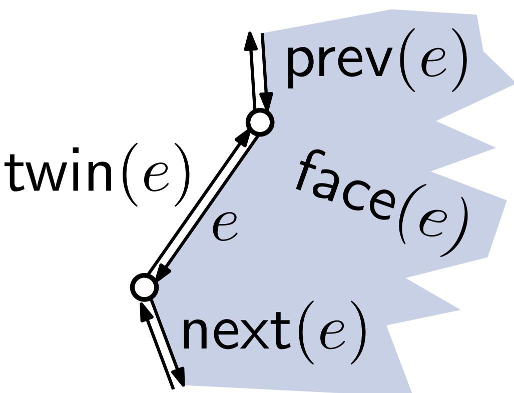
Ingredients:

- Vertices



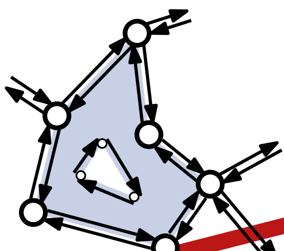
  - Coordinates $(x(v), y(v))$
  - (first) outgoing edge

- Edge $=$ two half-edges



  - Vertex origin(v)
  - Opposite edge twin(e)
  - Predecessor prev(e) & Successor next(e)
  - ~~incident face~~

- ~~Faces~~

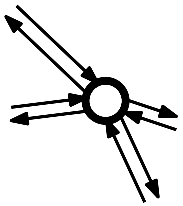


  - ~~Bounding edges for outer face.~~
  - ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

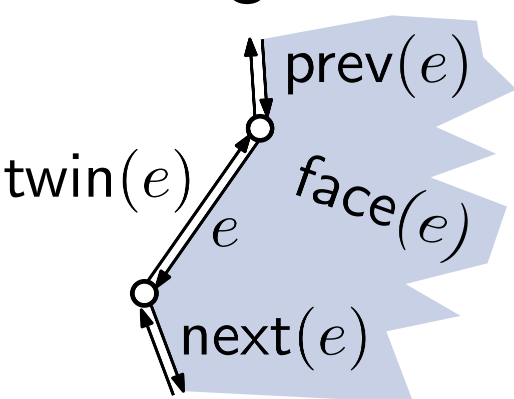
Ingredients:

- Vertices



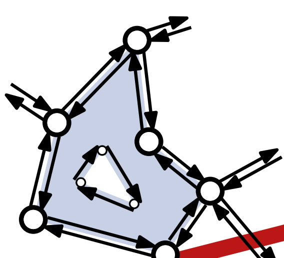
- Coordinates $(x(v), y(v))$ ✓
- (first) outgoing edge ✓

- Edge $=$ two half-edges



- Vertex origin(v) ✓
- Opposite edge twin(e) ✓
- Predecessor prev(e) & Successor next(e)
- ~~incident face~~

- ~~Faces~~

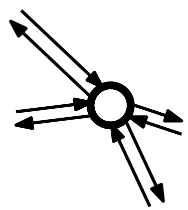


- ~~Bounding edges for outer face.~~
- ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

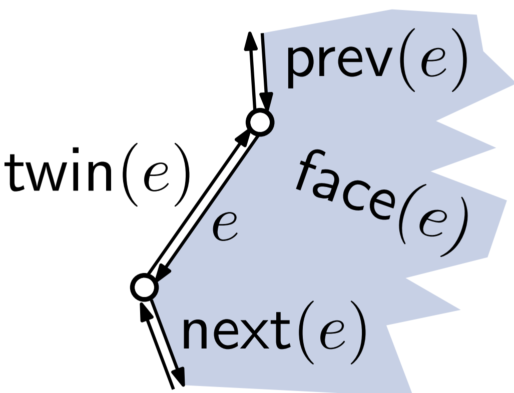
Ingredients:

- Vertices



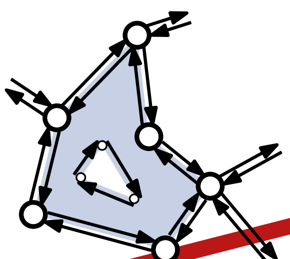
  - Coordinates $(x(v), y(v))$
  - (first) outgoing edge

- Edge $=$ two half-edges



  - Vertex origin(v)
  - Opposite edge twin(e)
  - Predecessor prev(e) & Successor next(e) ?
  - ~~incident face~~

- ~~Faces~~

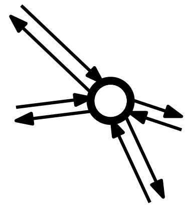


  - ~~Bounding edges for outer face.~~
  - ~~Edge list inner(f) for holes.~~

Doubly Connected Edge List (DCEL)

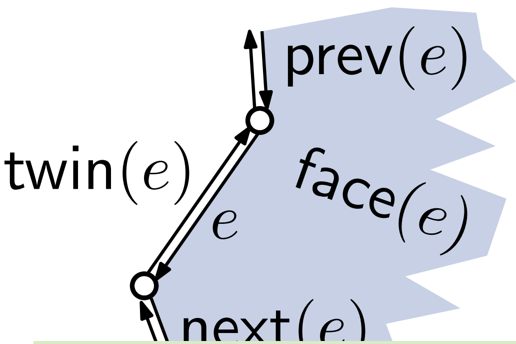
Ingredients:

- Vertices



  - Coordinates $(x(v), y(v))$
  - (first) outgoing edge

- Edge $=$ two half-edges



  - Vertex origin(v)
  - Opposite edge twin(e)
  - Predecessor prev(e) & Successor next(e) ?
  - ~~incident face~~

a) Each vertex has $O(1)$ incident edges.
- Initially each vertex has degree 2.
- Each vertex is at most once helper $+1$
- Each vertex is handled at most once : $+2$

b) Using a appropriate ordering, we can find the desired edges.

Linear Running Time

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$

Assumption: P contains $O(1)$ turn vertices.

Exercise: Adapt procedure such that it has $O(n)$ running time.

Linear Running Time

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doubly-connected edge list for $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue for $V(P)$ sorted lexicographically; $\mathcal{T} \leftarrow \emptyset$
(binary search tree for sweep-line status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$

Assumption: P contains $O(1)$ turn vertices.

Exercise: Adapt procedure such that it has $O(n)$ running time.

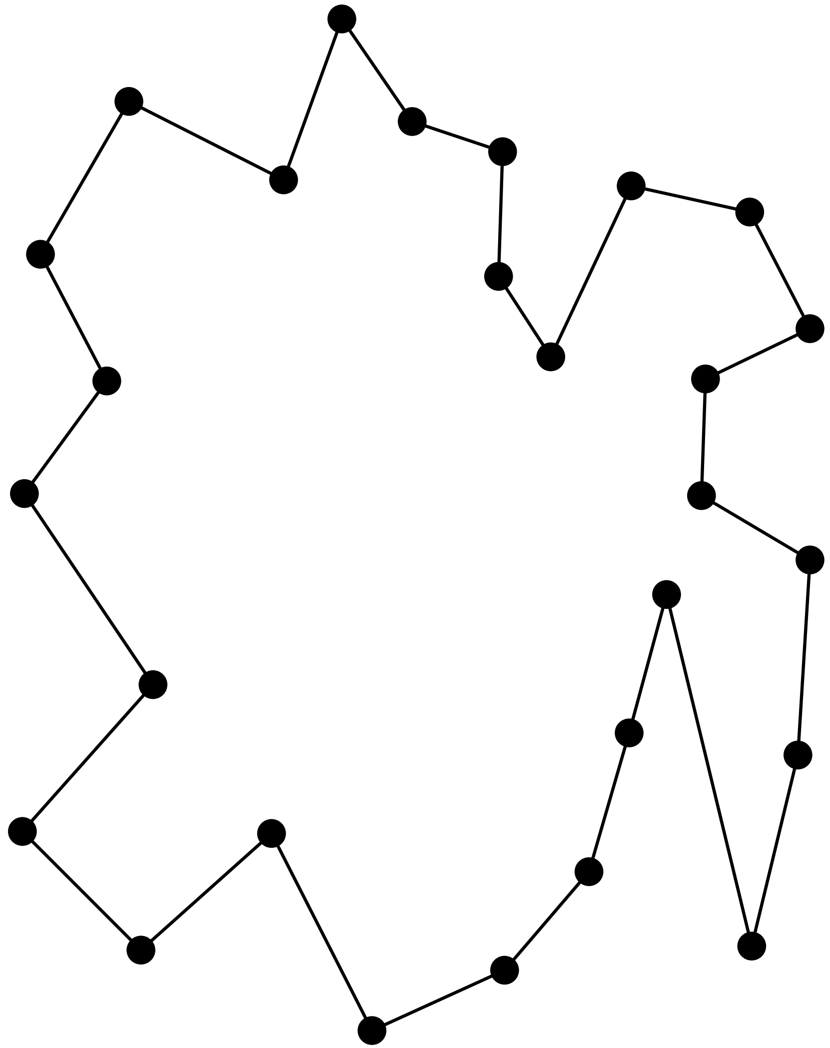
Observation:

Creation of $\mathcal{Q}$ costs $O(n \log n)$ time.

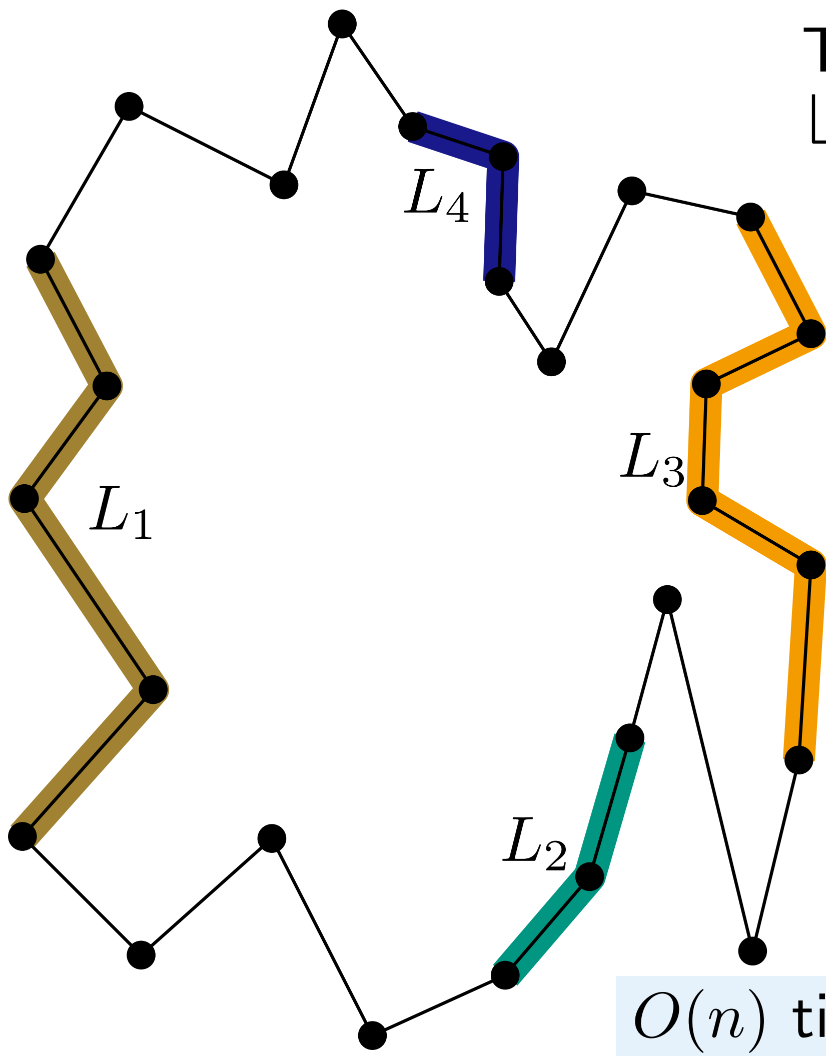
Querries in $\mathcal{T}$ cost $O(n \log n)$ time in total.

Linear Running Time

Step 1: Create queue $\mathcal{Q}$ in $O(n)$ time.



Linear Running Time



Step 1: Create queue $\mathcal{Q}$ in $O(n)$ time.

Traverse P in counter-clockwise order.
↳ Add consecutive regular vertices to a list.

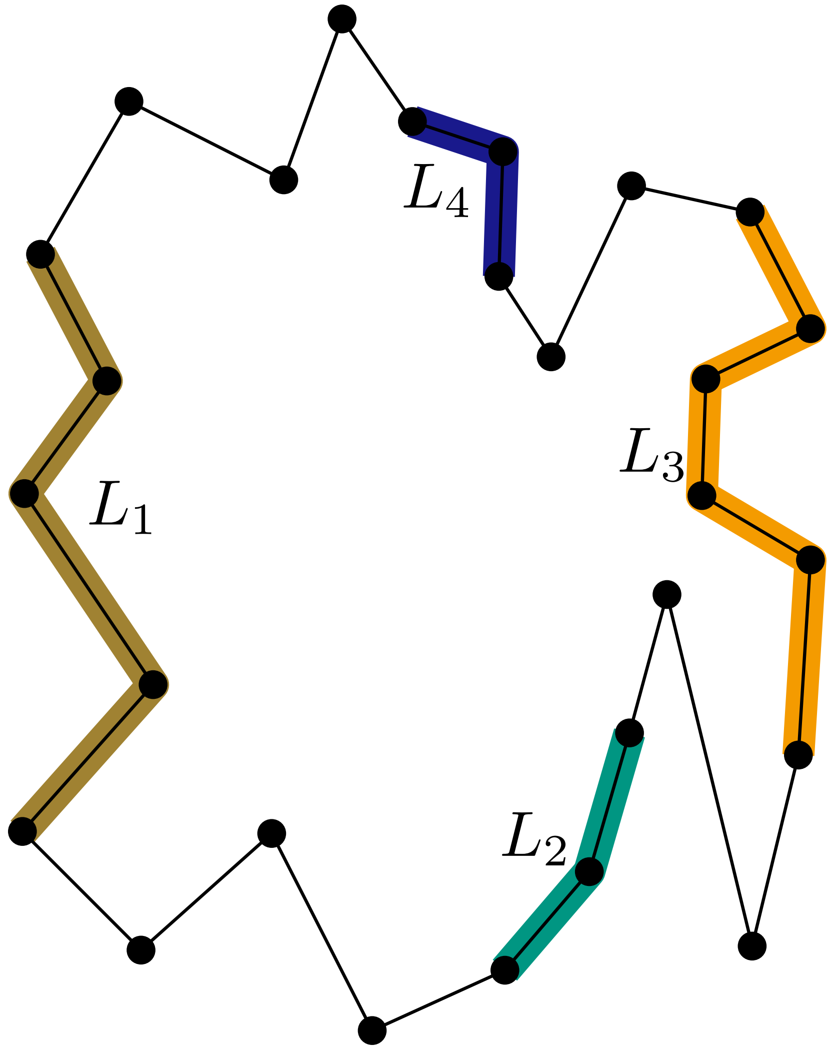
Observation:
- Lists are sorted by y-coordinate.
- $O(1)$ many lists.

1. Apply *merge*-step of Merge-Sort on lists, to obtain *one* list.
2. Insert turn vertices into list maintaining the sorting.

$O(n)$ time, since
$O(1)$ many lists and $O(1)$ many turn vertices.

⟶ Queue $\mathcal{Q}$ can be created in $O(n)$ time.

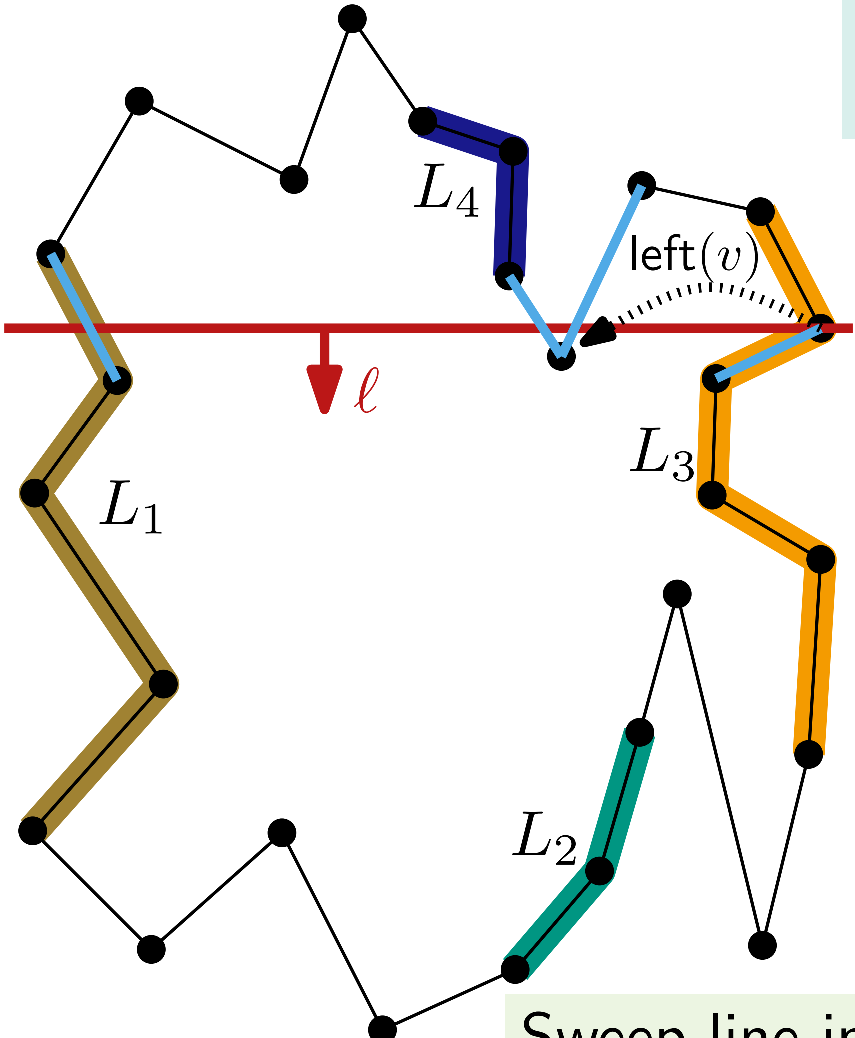
Linear Running Time



Step 2: Replace $\mathcal{T}$.

Task of $\mathcal{T}$: Determine for vertex v the edge left(v) directly left to v.

Linear Running Time



Step 2: Replace $\mathcal{T}$.

Task of $\mathcal{T}$: Determine for vertex v the edge left(v) directly left to v.

Idea: For each vertex v precompute left(v).

Sweep-Line: from top to bottom.
Sweep-State:
Edges that intersect sweep-line
Event: Vertices of polygon.
Determine edge that intersects sweep-line directly left to current node.

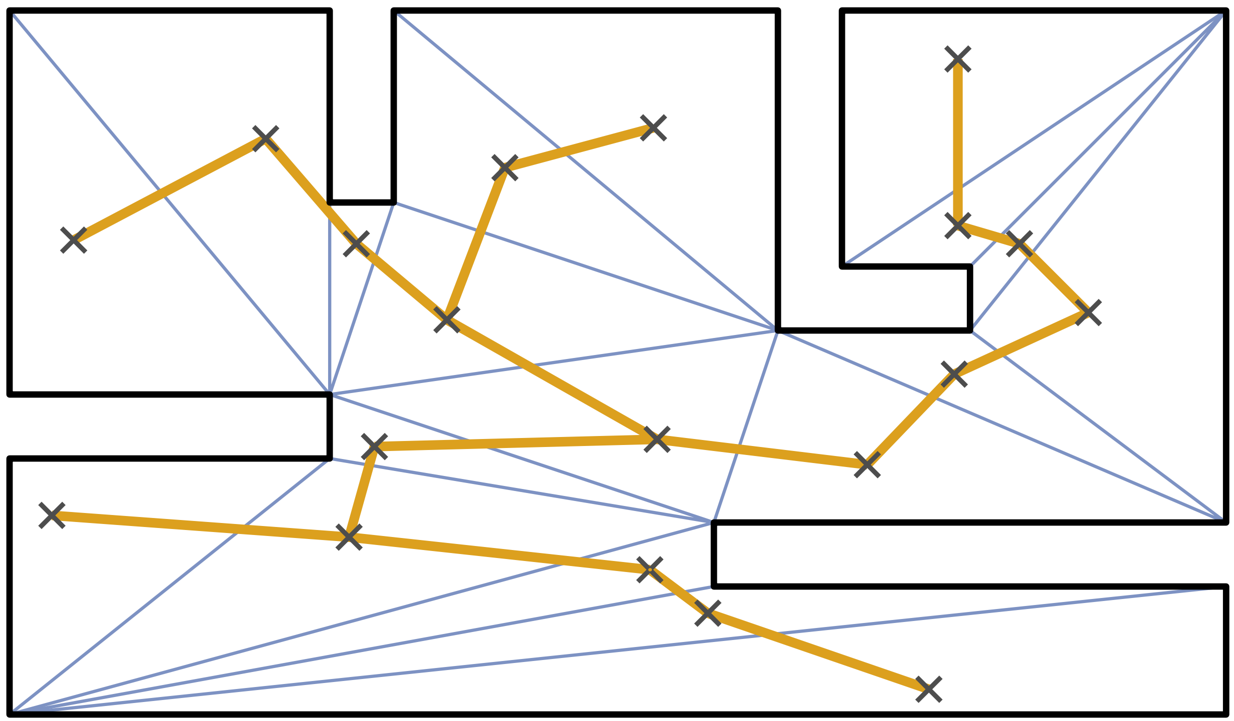
Sweep-line intersects $O(1)$ many edges,
since $O(1)$ many lists and $O(1)$ many turn vertices.

Splitting Polygons.

Given: Polygon P with n vertices.

Find: $O(n \log n)$-Algorithm, that splits P into two simple polygons such that each has at most $\lfloor 2n/3 \rfloor + 2$ vertices.

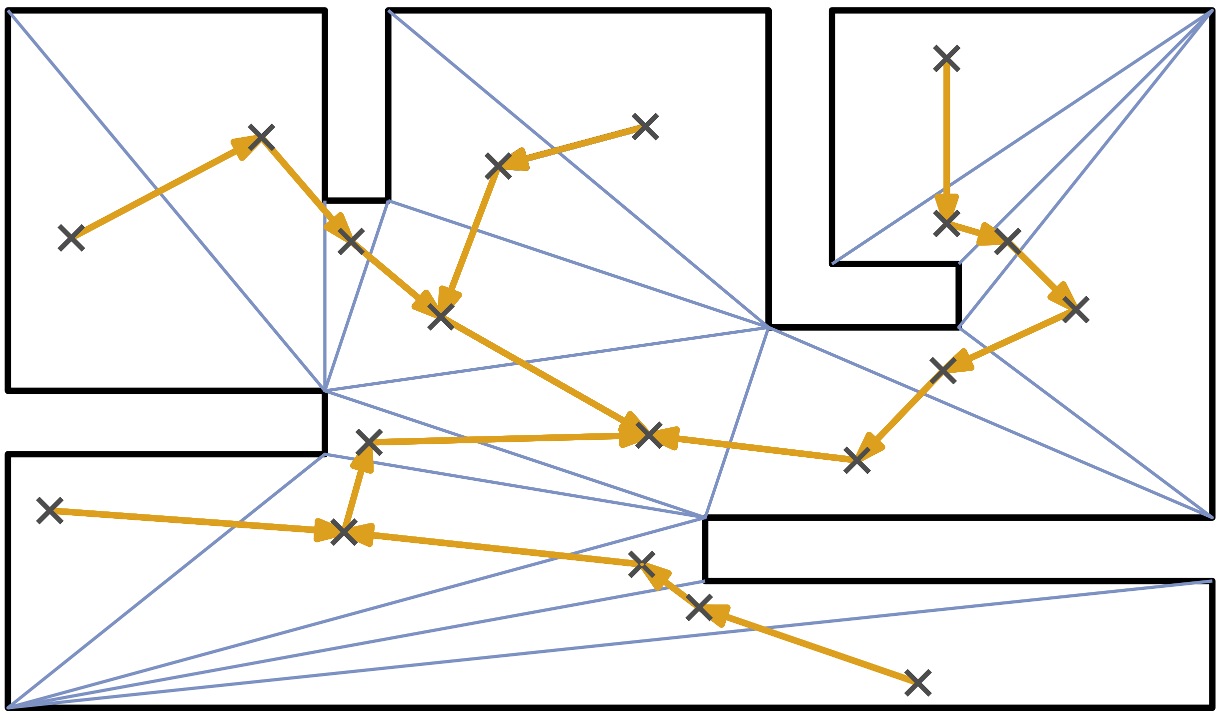
Hint: Triangulate P and make use of the dual graph of the triangulation.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

while $\textsc{True}$ **do**
  Let u be leaf of T
  while u has degree 1 **do**
    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
      return Sub-tree of u induces desired partition
    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
    Delete u from T
    $u \leftarrow parent(u)$



$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.

Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

while $\textsc{True}$ **do**

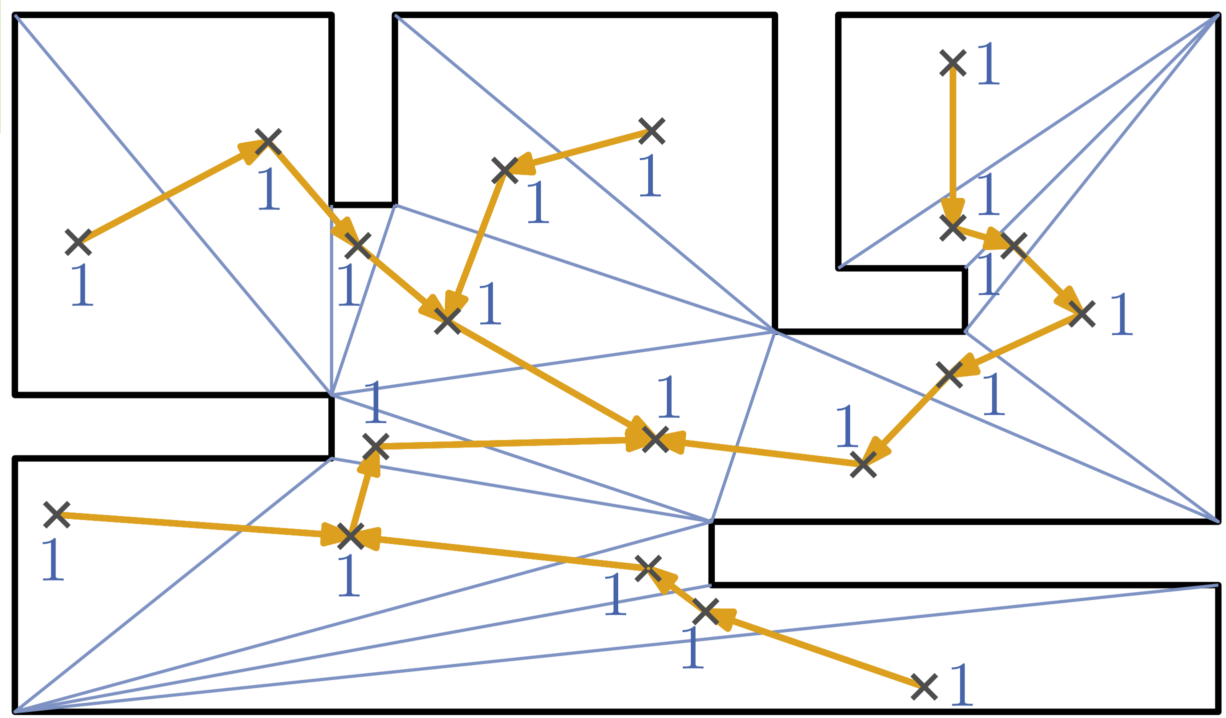
- Let u be leaf of T
- **while** u has degree 1 **do**
  - **if** $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
    - **return** Sub-tree of u induces desired partition
  - $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
  - Delete u from T
  - $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

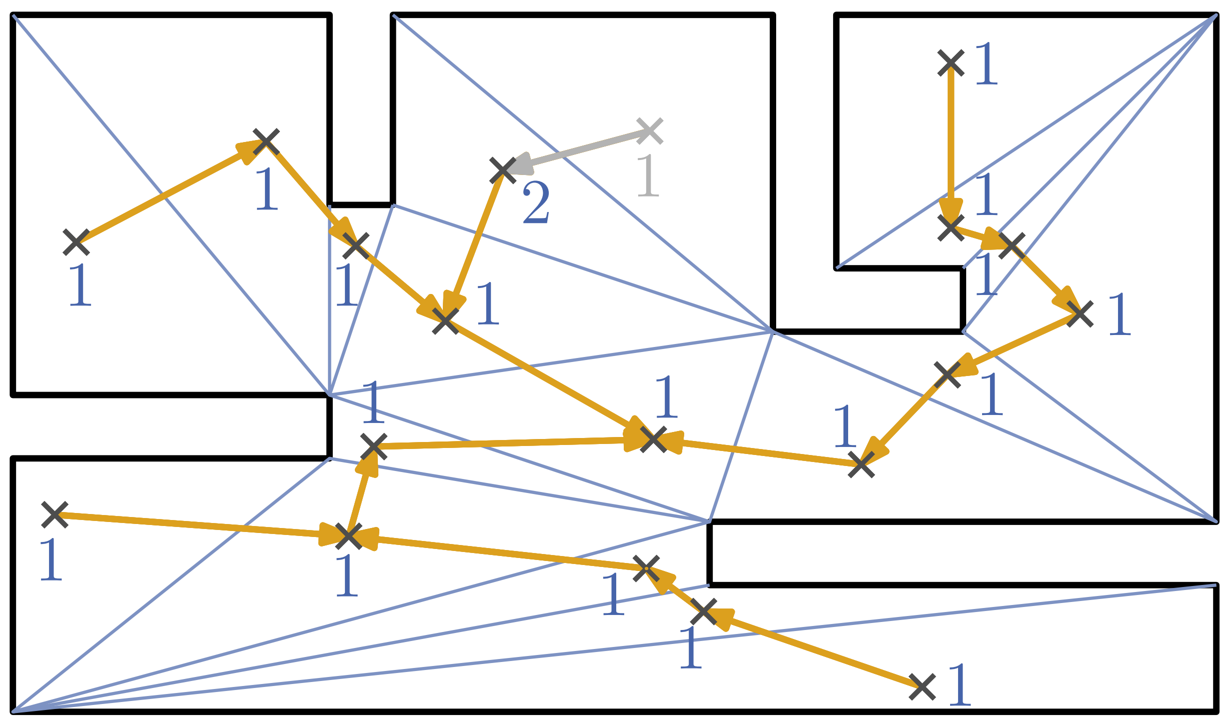
while $\textsc{True}$ **do**
  Let u be leaf of T
  while u has degree 1 **do**
    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
      return Sub-tree of u induces desired partition
    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
    Delete u from T
    $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

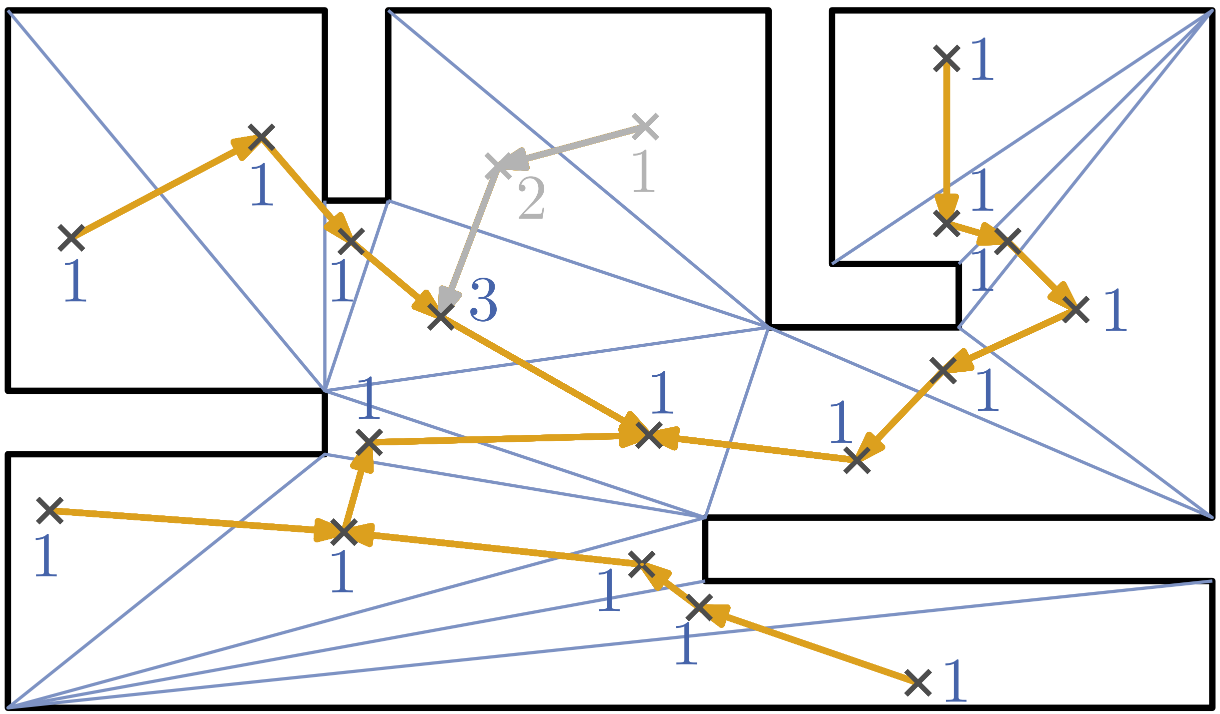
while $\textsc{True}$ **do**
  Let u be leaf of T
  while u has degree 1 **do**
    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
      return Sub-tree of u induces desired partition
    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
    Delete u from T
    $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

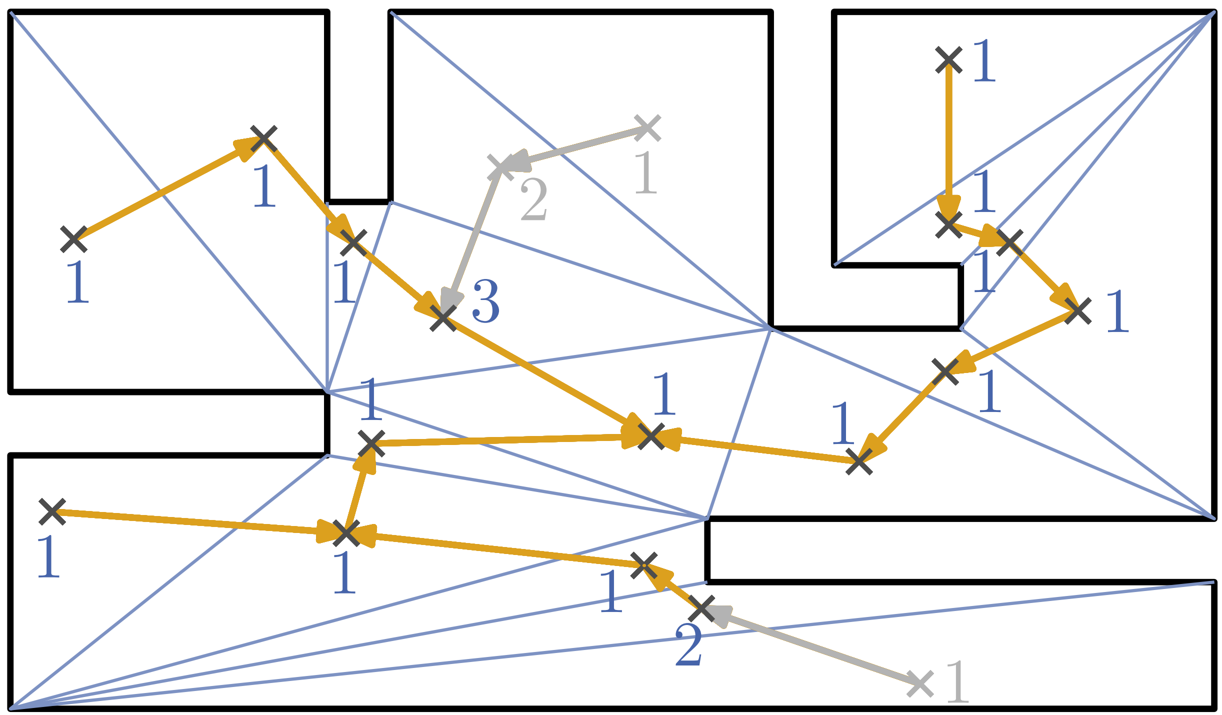
while $\textsc{True}$ **do**
- Let u be leaf of T
- **while** u has degree 1 **do**
  - **if** $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
    - **return** Sub-tree of u induces desired partition
  - $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
  - Delete u from T
  - $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



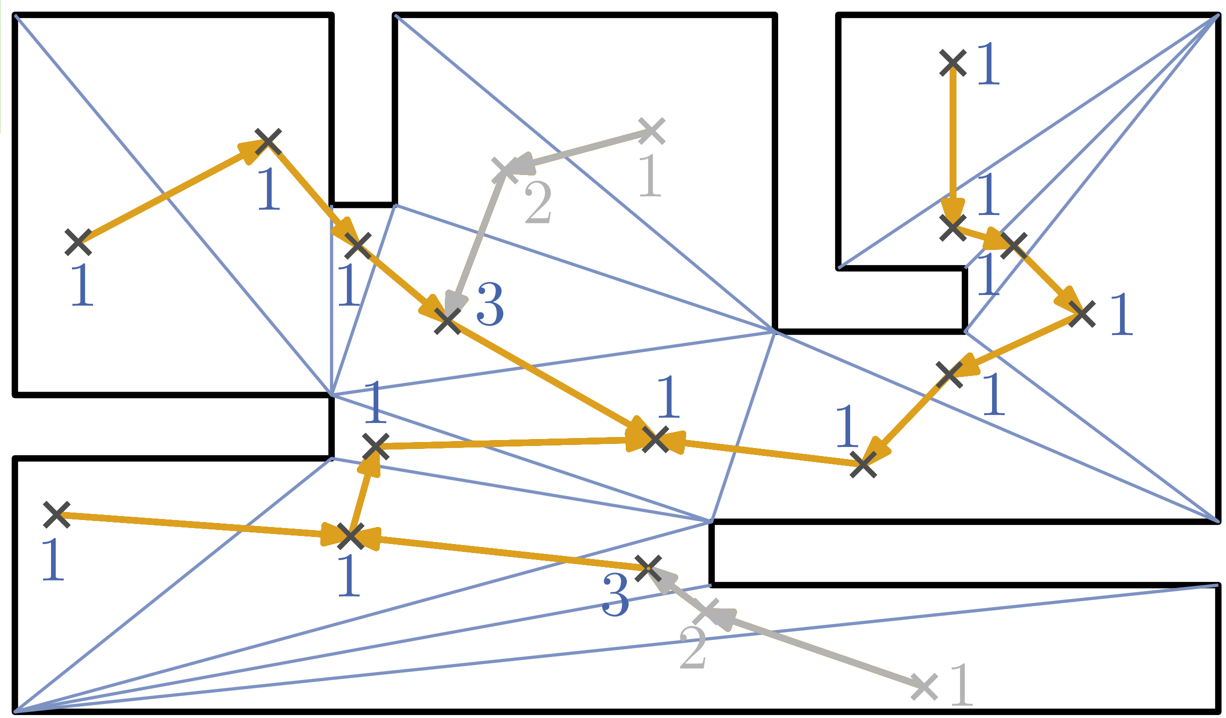
Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

while $\textsc{True}$ **do**
  Let u be leaf of T
  while u has degree 1 **do**
    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
      return Sub-tree of u induces desired partition
    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
    Delete u from T
    $u \leftarrow parent(u)$

$n = 19$
$n - (\lfloor 2n/3 \rfloor + 2) = 5$
$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

while $\textsc{True}$ **do**

  Let u be leaf of T

  while u has degree 1 **do**

    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**

      return Sub-tree of u induces desired partition

    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$

    Delete u from T

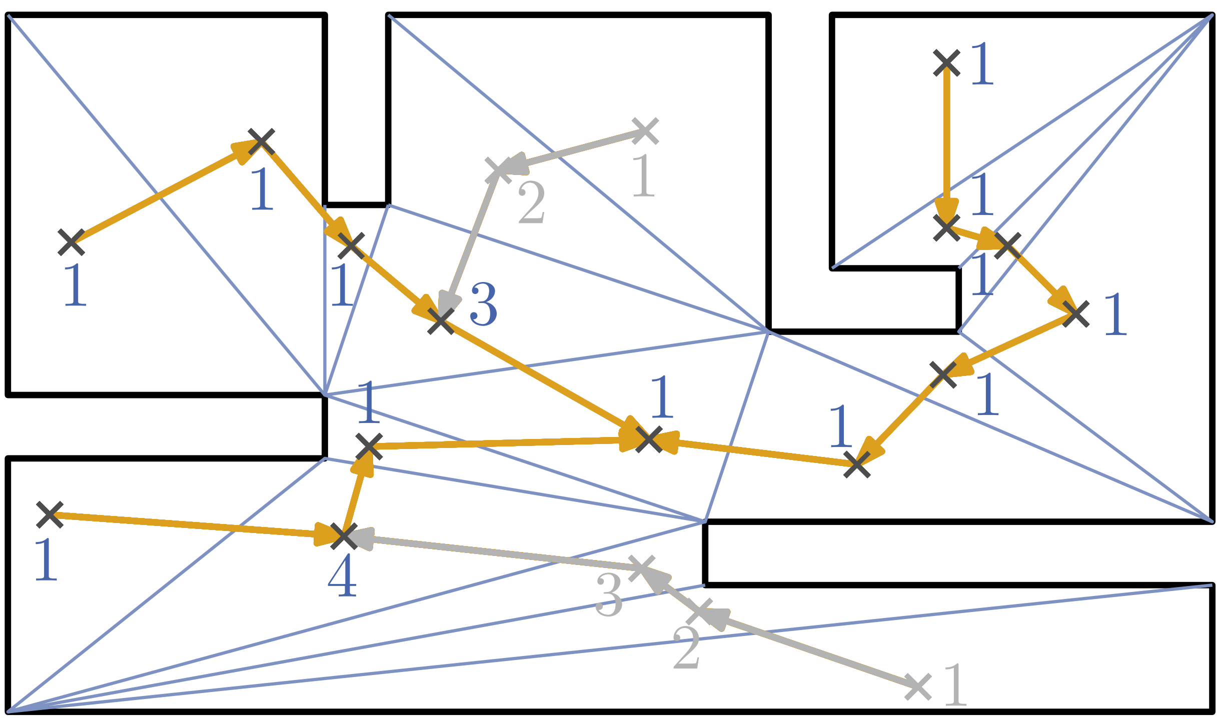
    $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

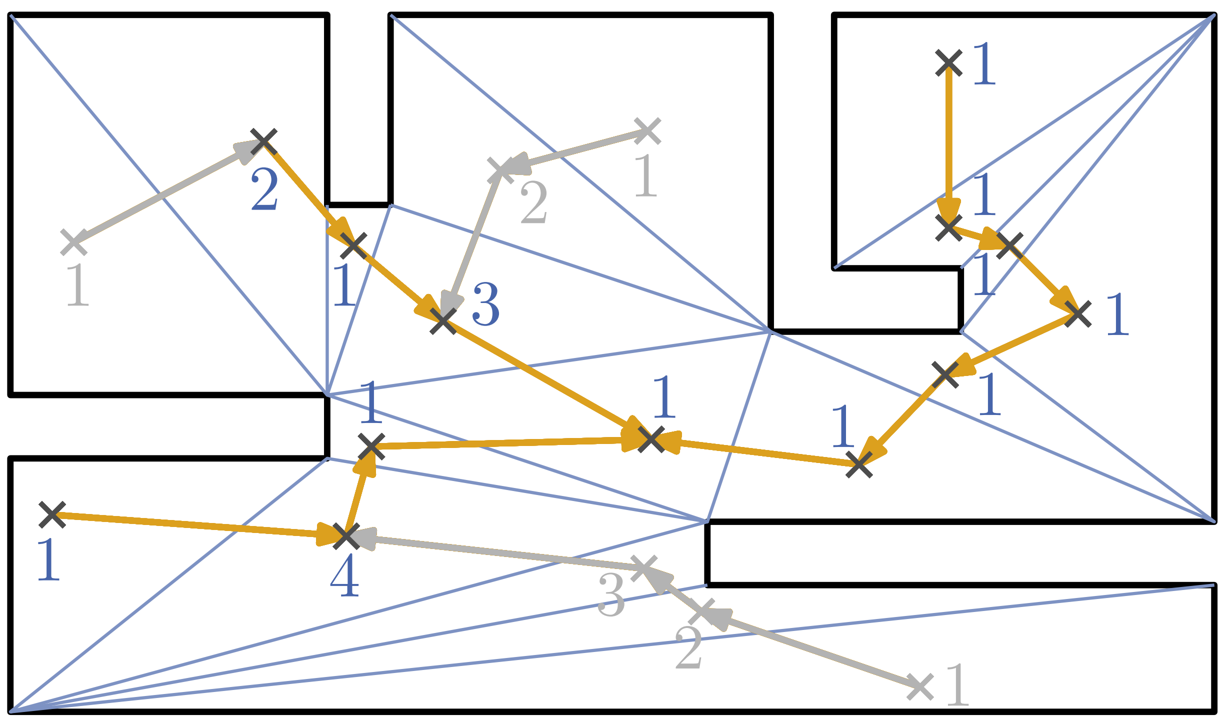
while $\textsc{True}$ **do**
- Let u be leaf of T
- **while** u has degree 1 **do**
  - **if** $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
    - **return** Sub-tree of u induces desired partition
  - $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
  - Delete u from T
  - $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

while $\textsc{True}$ **do**

  Let u be leaf of T

  while u has degree 1 **do**

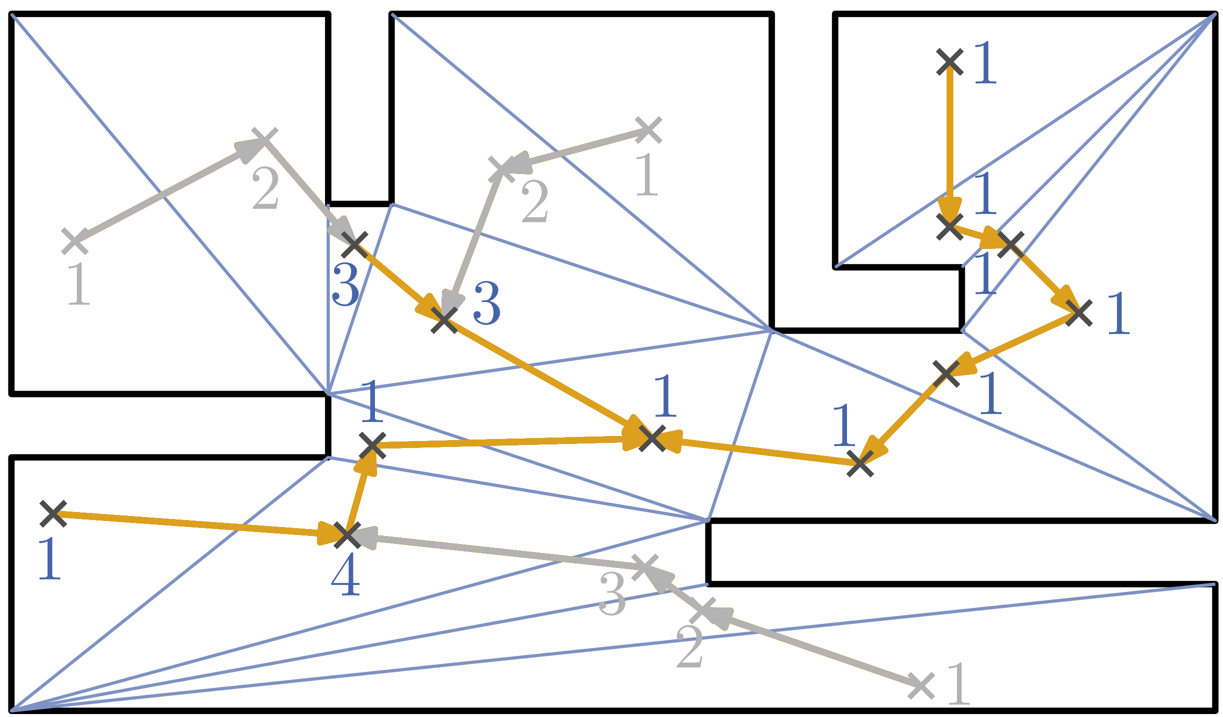
    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**

      return Sub-tree of u induces desired partition

    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$

    Delete u from T

    $u \leftarrow parent(u)$



$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.

Solution

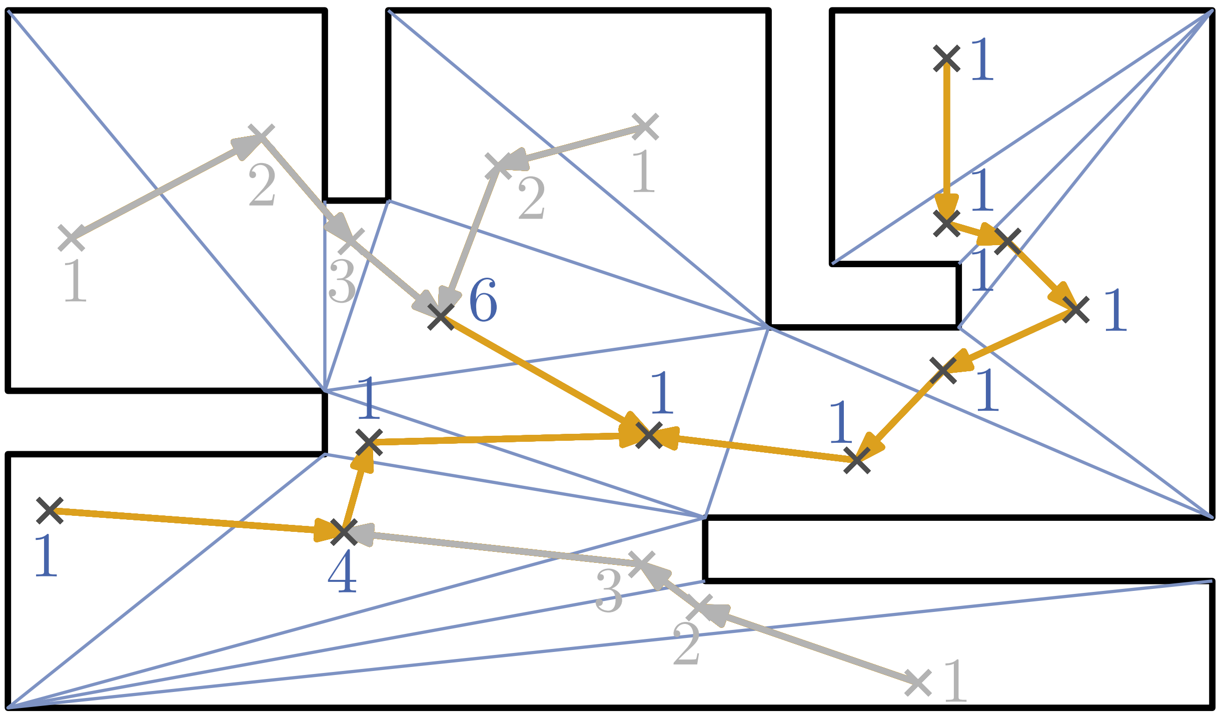
Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

while $\textsc{True}$ **do**
- Let u be leaf of T
- **while** u has degree 1 **do**
  - **if** $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
    - **return** Sub-tree of u induces desired partition
  - $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
  - Delete u from T
  - $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$



Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.

Solution

Initialitation: Each vertex $u \in V$ receives weight $w(u) = 1$.

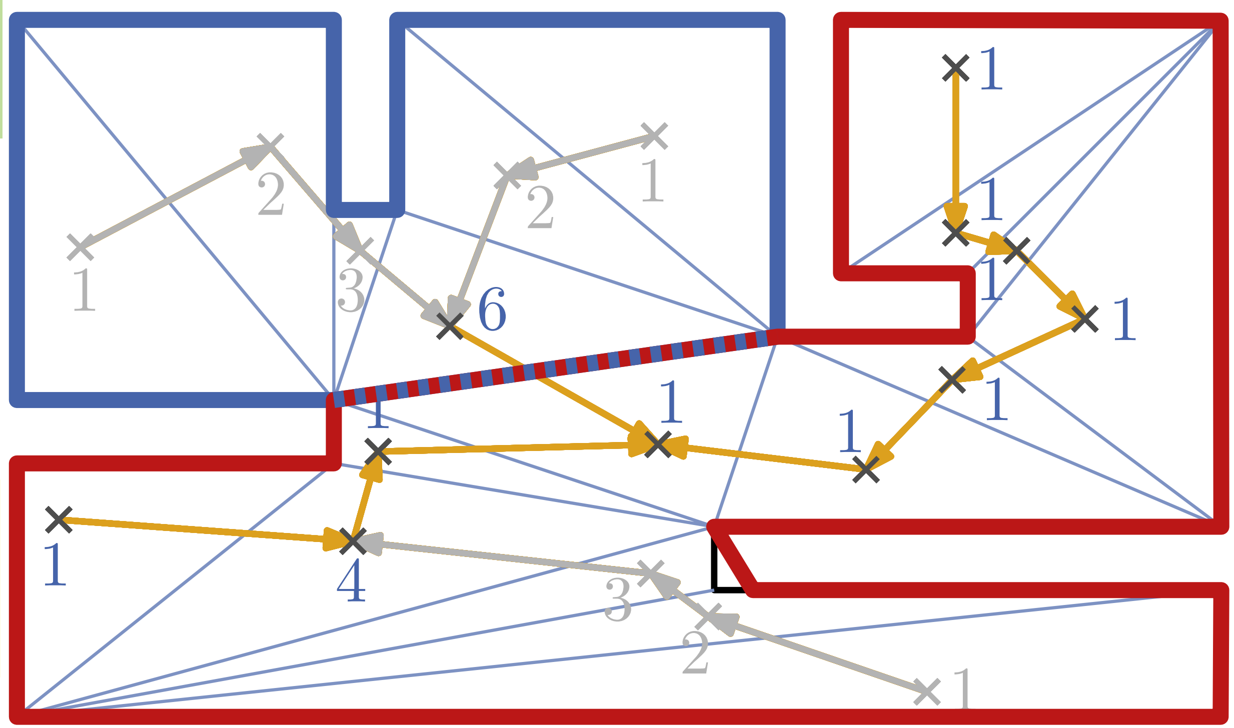
while $\textsc{True}$ **do**
  Let u be leaf of T
  while u has degree 1 **do**
    if $n - (\lfloor 2n/3 \rfloor + 2) \leq w(u) \leq \lfloor 2n/3 \rfloor + 2$ **then**
      return Sub-tree of u induces desired partition
    $w(parent(u)) \leftarrow w(parent(u)) + w(u)$
    Delete u from T
    $u \leftarrow parent(u)$

$n = 19$

$n - (\lfloor 2n/3 \rfloor + 2) = 5$

$\lfloor 2n/3 \rfloor + 2 = 14$

Annahme:
Tree has root with degree ≥ 2.
Edges are directed to the root.



Linear programming

Definition: Given a set of linear constraints H and a linear objective function c in $\mathbb{R}^d$, a **linear program** (LP) is formulated as follows:

$$\begin{array}{rl} \text{maximize} & c_1x_1 + c_2x_2 + \cdots + c_dx_d \\[1em] \text{under constr.} & \left.\begin{array}{lcl} a_{1,1}x_1 + \cdots + a_{1,d}x_d & \leq & b_1 \\ a_{2,1}x_1 + \cdots + a_{2,d}x_d & \leq & b_2 \\ & \vdots & \\ a_{n,1}x_1 + \cdots + a_{n,d}x_d & \leq & b_n \end{array}\right\} H \end{array}$$

Linear programming

Definition: Given a set of linear constraints H and a linear objective function c in $\mathbb{R}^d$, a **linear program** (LP) is formulated as follows:

$$
\begin{array}{ll}
\text{maximize} & c_1x_1 + c_2x_2 + \cdots + c_dx_d \\[1em]
\text{under constr.} & \left.\begin{array}{rcl}
a_{1,1}x_1 + \cdots + a_{1,d}x_d & \leq & b_1 \\
a_{2,1}x_1 + \cdots + a_{2,d}x_d & \leq & b_2 \\
& \vdots & \\
a_{n,1}x_1 + \cdots + a_{n,d}x_d & \leq & b_n
\end{array}\right\} H
\end{array}
$$

- H is a set of half-spaces in $\mathbb{R}^d$.
- We are searching for a point $x \in \bigcap_{h \in H} h$, that maximizes $c^T x$, i.e. $\max\{c^\mathrm{T} x \mid Ax \leq b, x \geq 0\}$.
- Linear programming is a central method in operations research.

Algorithms for LPs

There are many algorithms to solve LPs:

- Simplex-Algorithm [Dantzig, 1947]
- Ellipsoid-Method [Khatchiyan, 1979]
- Interior-Point-Method [Karmarkar, 1979]

They work well in practice, especially for large values of n (number of constraints) and d (number of variables).

Algorithms for LPs

There are many algorithms to solve LPs:

- Simplex-Algorithm [Dantzig, 1947]
- Ellipsoid-Method [Khatchiyan, 1979]
- Interior-Point-Method [Karmarkar, 1979]

They work well in practice, especially for large values of n (number of constraints) and d (number of variables).

Today: Special case $d = 2$

Algorithms for LPs

There are many algorithms to solve LPs:

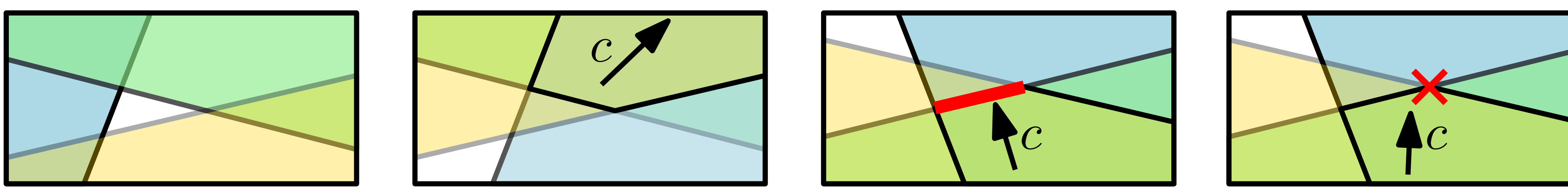
- Simplex-Algorithm [Dantzig, 1947]
- Ellipsoid-Method [Khatchiyan, 1979]
- Interior-Point-Method [Karmarkar, 1979]

They work well in practice, especially for large values of n (number of constraints) and d (number of variables).

Today: Special case $d = 2$

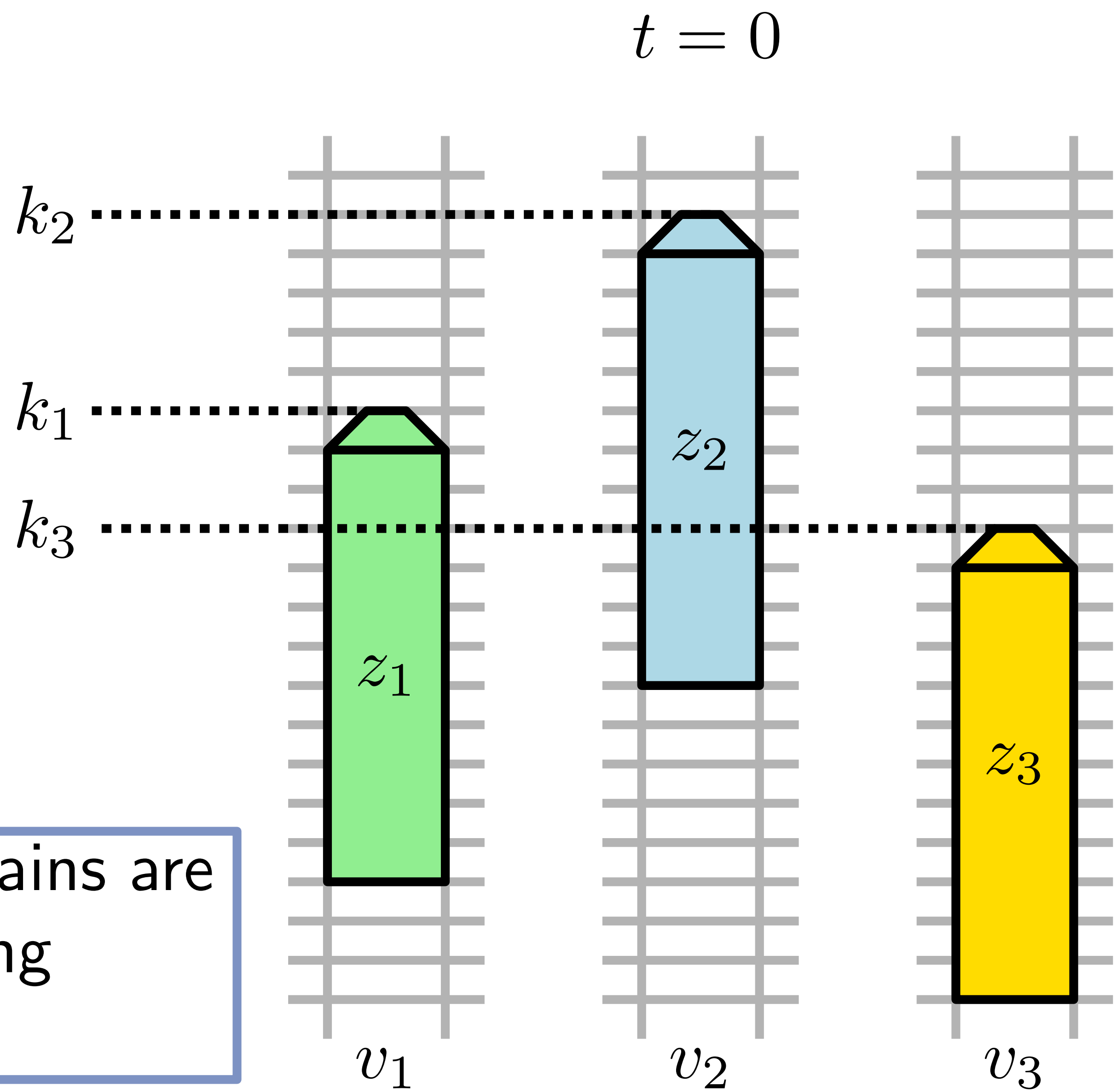
Possibilities for the solution space

feasible region $\bigcap H$ is bounded



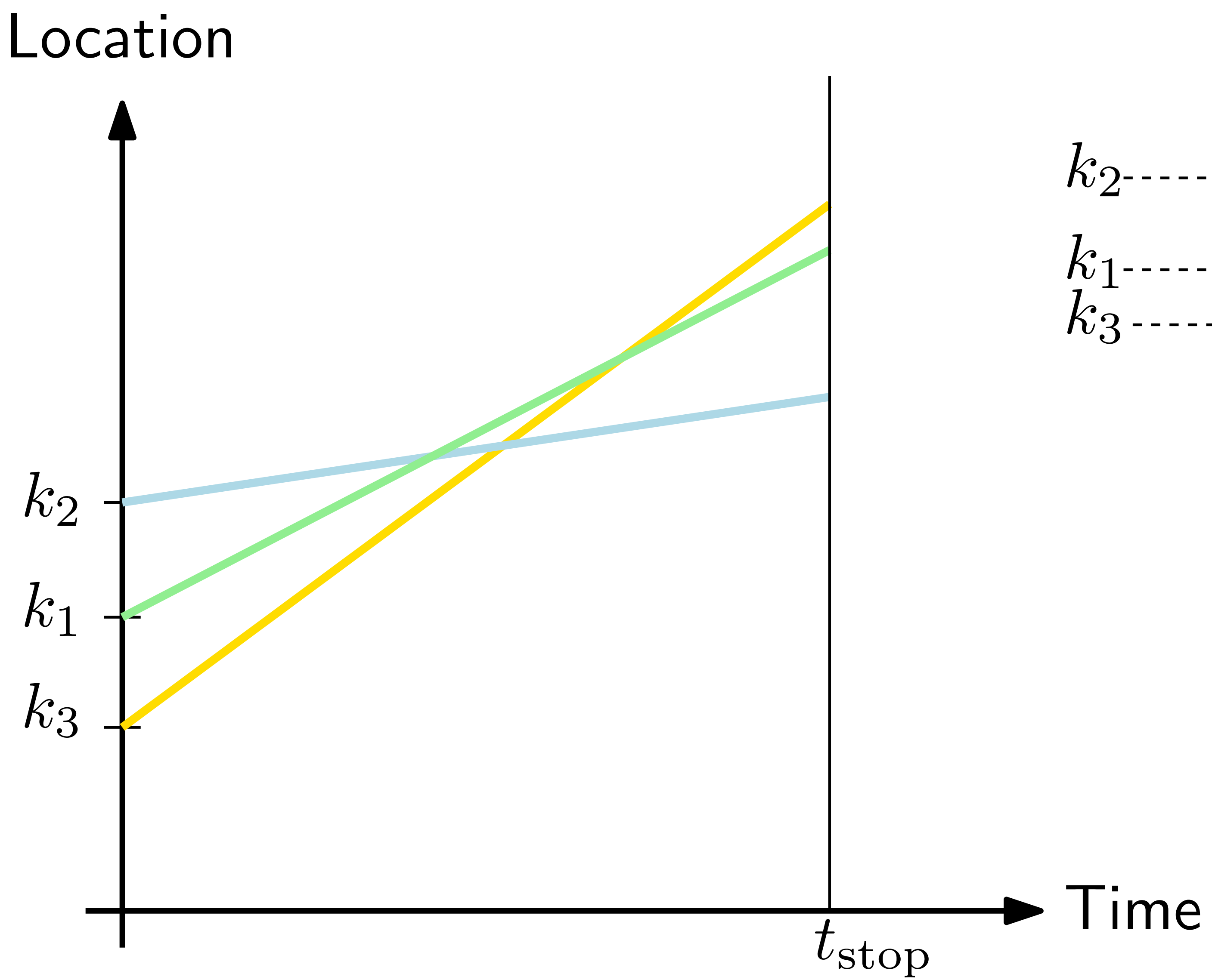
$\bigcap H = \emptyset$ infeasible | $\bigcap H$ is unbounded in the direction c | solution is not unique | unique solution

Trains

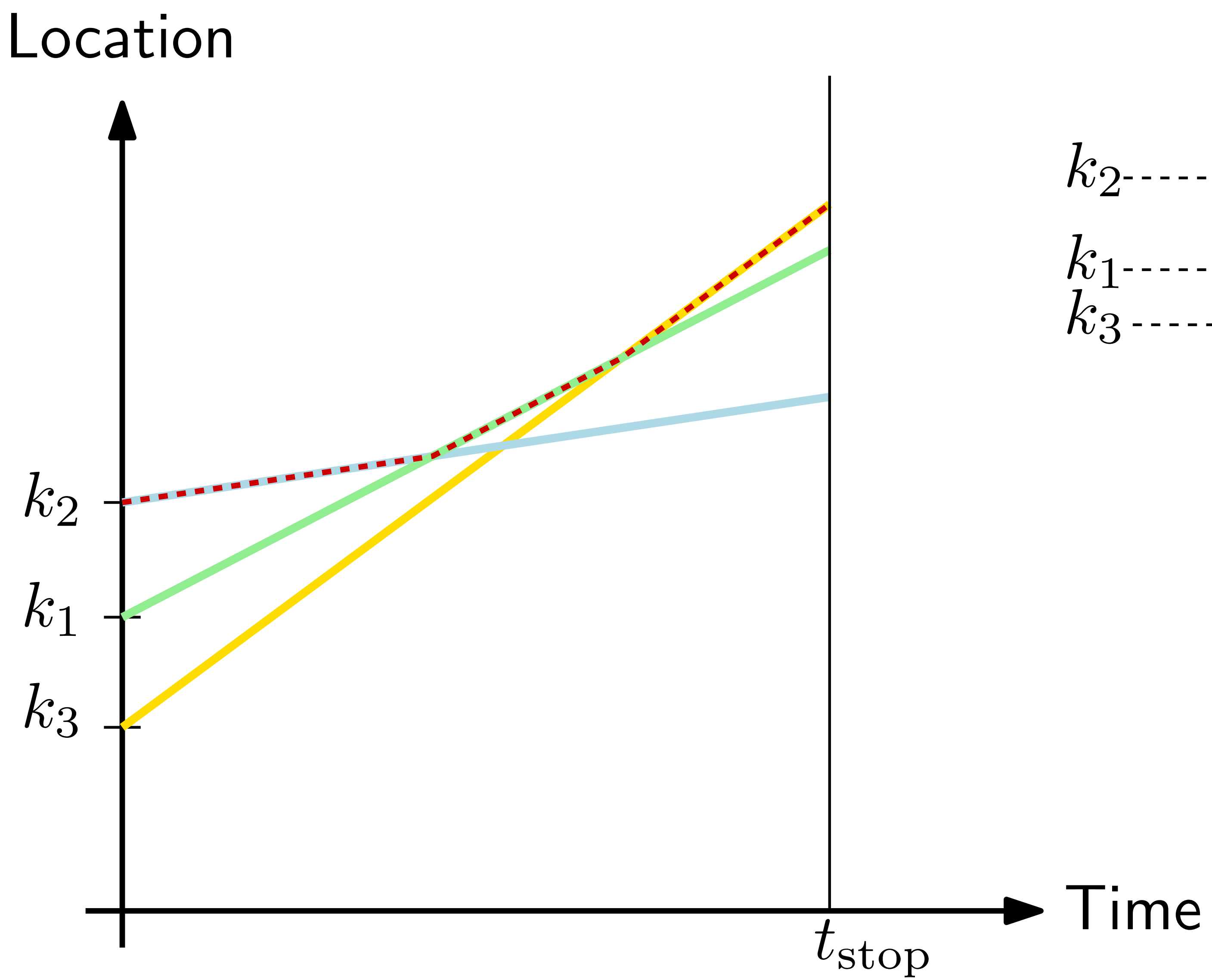


Algorithm: Which trains are at least once in leading position until time t_s

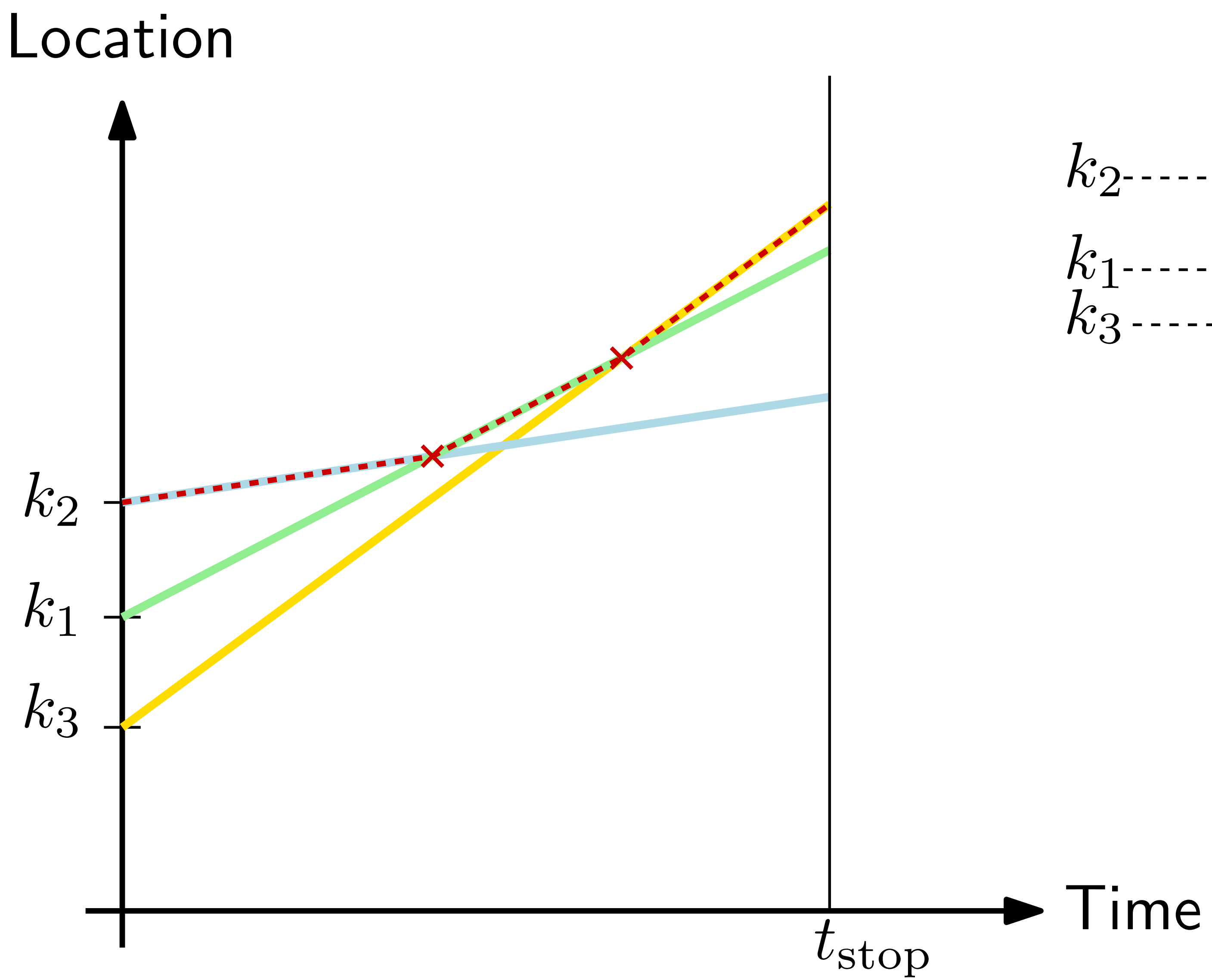
Trains



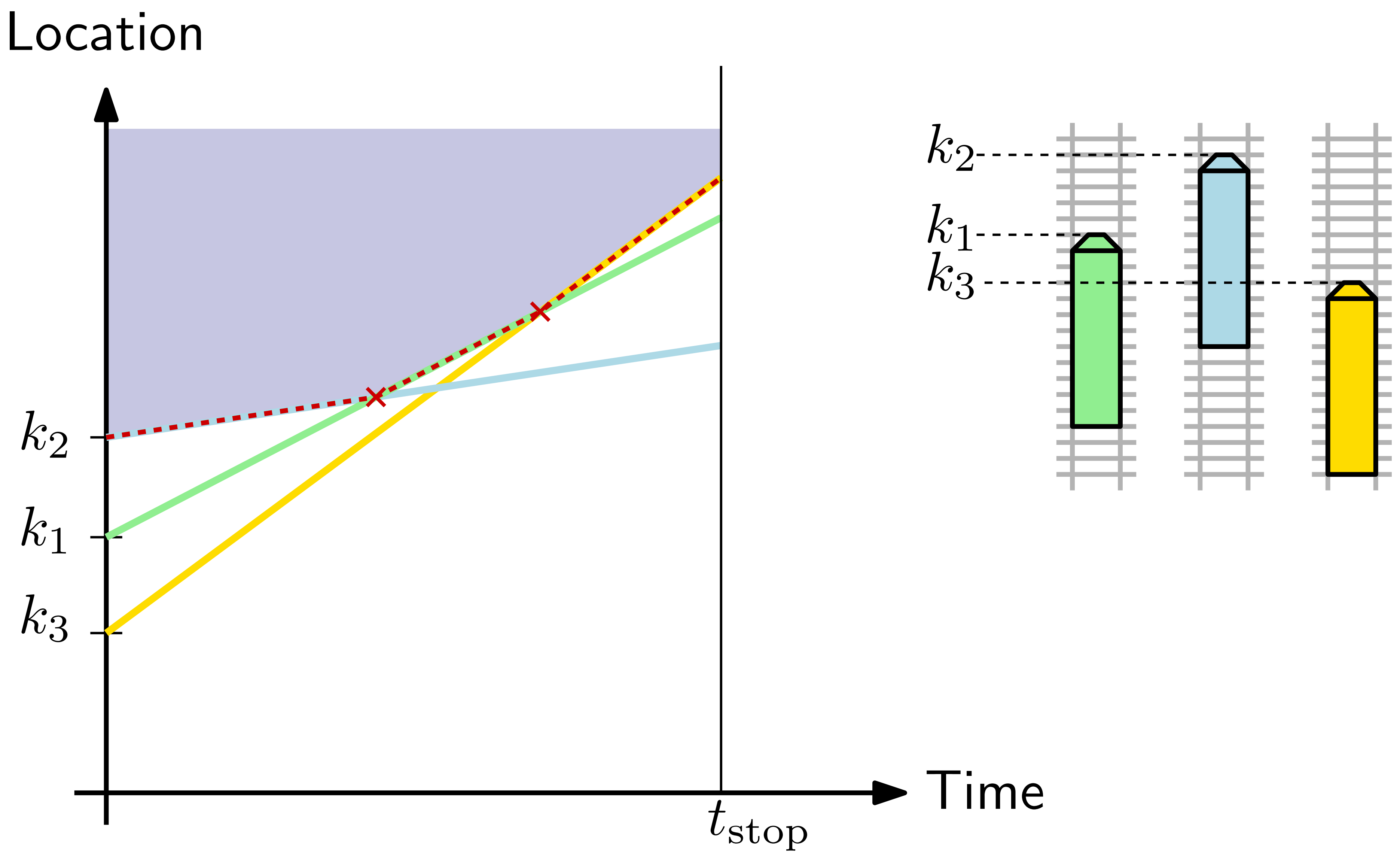
Trains



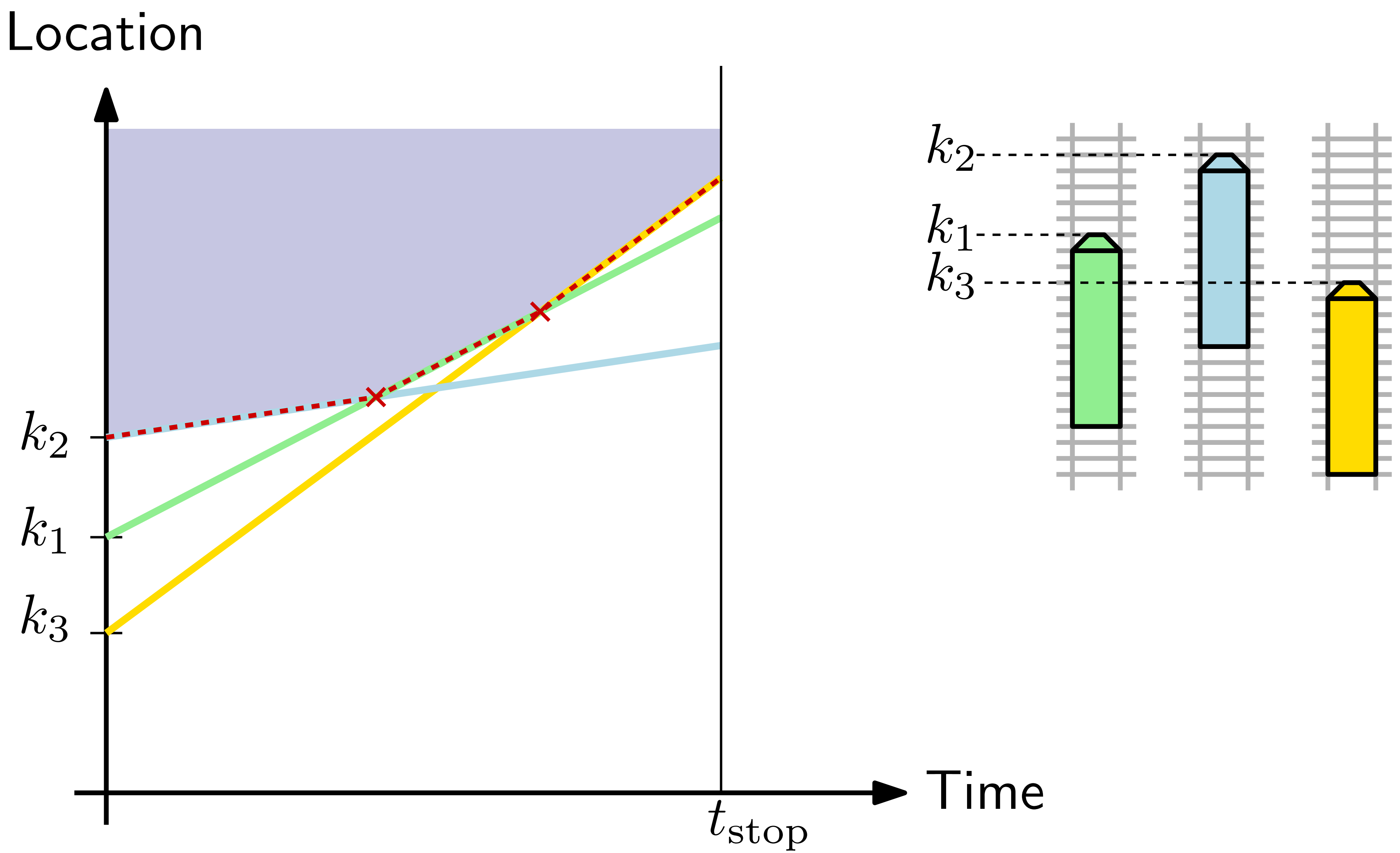
Trains



Trains



Trains



First approach

Idea: Compute the feasible region $\bigcap H$ and search for the vertex p, that maximizes $c^T p$.

- The half-planes are convex
- Let's try a simple Divide-and-Conquer Algorithm

IntersectHalfplanes(H)

if $|H| = 1$ **then**
    $C \leftarrow H$
else
    $(H_1, H_2) \leftarrow$ SplitInHalves(H)
    $C_1 \leftarrow$ IntersectHalfplanes(H_1)
    $C_2 \leftarrow$ IntersectHalfplanes(H_2)
    $C \leftarrow$ IntersectConvexRegions(C_1, C_2)
return C

Bounded LPs

Idea: Instead of computing the feasible region and then searching for the optimal angle, do this incrementally.

Bounded LPs

Idea: Instead of computing the feasible region and then searching for the optimal angle, do this incrementally.

Invariant: Current best solution is a unique corner of the current feasible polygon

Bounded LPs

Idea: Instead of computing the feasible region and then searching for the optimal angle, do this incrementally.

Invariant: Current best solution is a unique corner of the current feasible polygon

How to deal with the unbounded feasible regions?

Bounded LPs

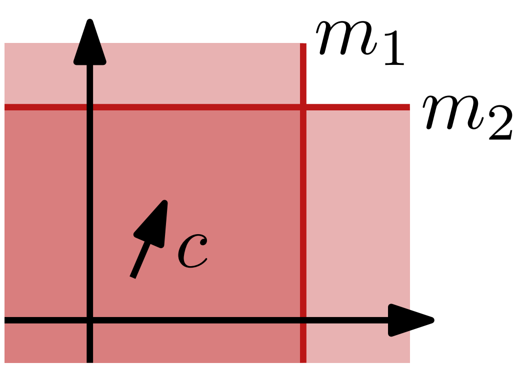
Idea: Instead of computing the feasible region and then searching for the optimal angle, do this incrementally.

Invariant: Current best solution is a unique corner of the current feasible polygon

How to deal with the unbounded feasible regions?

Define two half-planes for a big enough value M

$$m_1 = \begin{cases} x \leq M & \text{if } c_x > 0 \\ -x \leq M & \text{otherwise} \end{cases} \qquad m_2 = \begin{cases} y \leq M & \text{if } c_y > 0 \\ -y \leq M & \text{otherwise} \end{cases}$$



Bounded LPs

Idea: Instead of computing the feasible region and then searching for the optimal angle, do this incrementally.

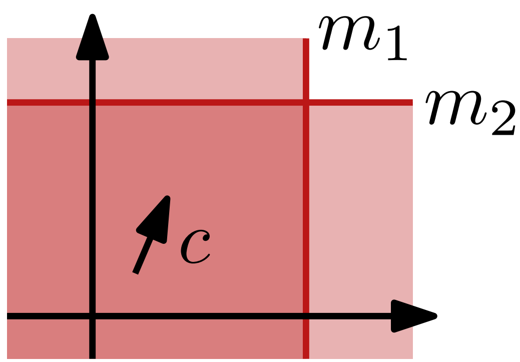
Invariant: Current best solution is a unique corner of the current feasible polygon

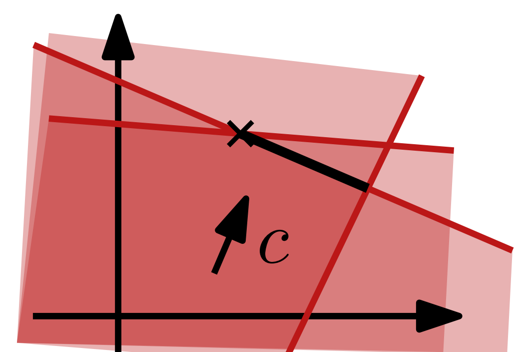
How to deal with the unbounded feasible regions?

When the optimal point is not unique, select lexicographically smallest one!

Define two half-planes for a big enough value M

$$m_1 = \begin{cases} x \leq M & \text{if } c_x > 0 \\ -x \leq M & \text{otherwise} \end{cases} \qquad m_2 = \begin{cases} y \leq M & \text{if } c_y > 0 \\ -y \leq M & \text{otherwise} \end{cases}$$





Bounded LPs

Idea: Instead of computing the feasible region and then searching for the optimal angle, do this incrementally.

Invariant: Current best solution is a unique corner of the current feasible polygon

How to deal with the unbounded feasible regions?

When the optimal point is not unique, select lexicographically smallest one!

Define two half-planes for a big enough value M

$$m_1 = \begin{cases} x \leq M & \text{if } c_x > 0 \\ -x \leq M & \text{otherwise} \end{cases} \qquad m_2 = \begin{cases} y \leq M & \text{if } c_y > 0 \\ -y \leq M & \text{otherwise} \end{cases}$$

Consider a LP (H, c) with $H = \{h_1, \ldots, h_n\}$, $c = (c_x, c_y)$. We denote the first i constraints by $H_i = \{m_1, m_2, h_1, \ldots, h_i\}$, and the feasible polygon defineed by them by $C_i = m_1 \cap m_2 \cap h_1 \cap \cdots \cap h_i$

Properties

- each region C_i has a single optimal vertex v_i

Properties

- each region C_i has a single optimal vertex v_i
- it holds that: $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Properties

- each region C_i has a single optimal vertex v_i
- it holds that: $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

How the optimal vertex v_{i-1} changes when the half plane h_i is added?

Properties

- each region C_i has a single optimal vertex v_i
- it holds that: $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

How the optimal vertex v_{i-1} changes when the half plane h_i is added?

Lemma 1: For $1 \leq i \leq n$ and bounding line ℓ_i of h_i holds that:

1. If $v_{i-1} \in h_i$ then $v_i = v_{i-1}$,
2. otherwise, either $C_i = \emptyset$ or $v_i \in \ell_i$.

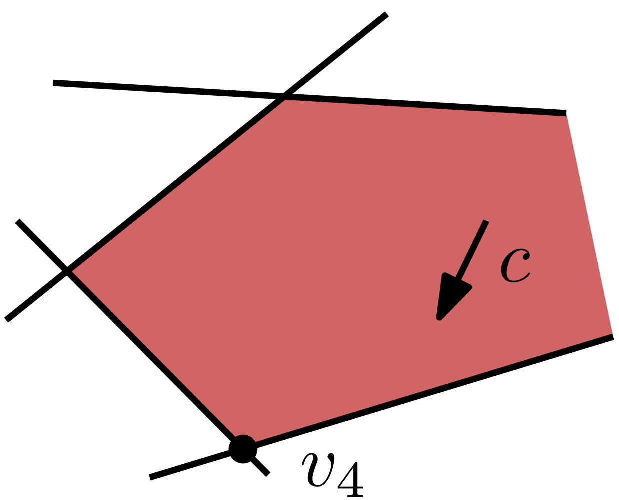
Properties

- each region C_i has a single optimal vertex v_i
- it holds that: $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

How the optimal vertex v_{i-1} changes when the half plane h_i is added?

Lemma 1: For $1 \leq i \leq n$ and bounding line ℓ_i of h_i holds that:

1. If $v_{i-1} \in h_i$ then $v_i = v_{i-1}$,
2. otherwise, either $C_i = \emptyset$ or $v_i \in \ell_i$.



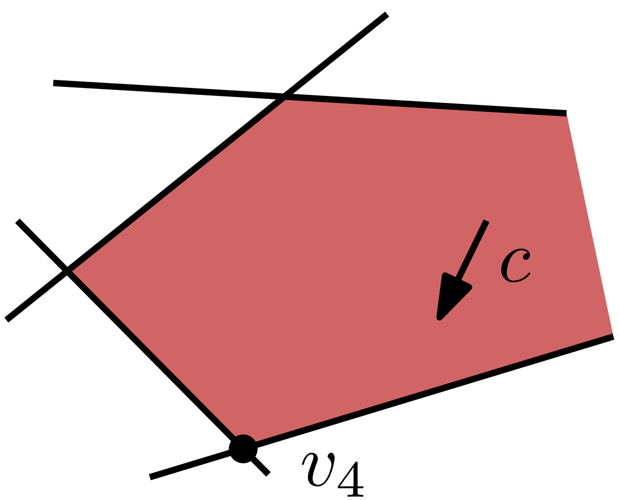
Properties

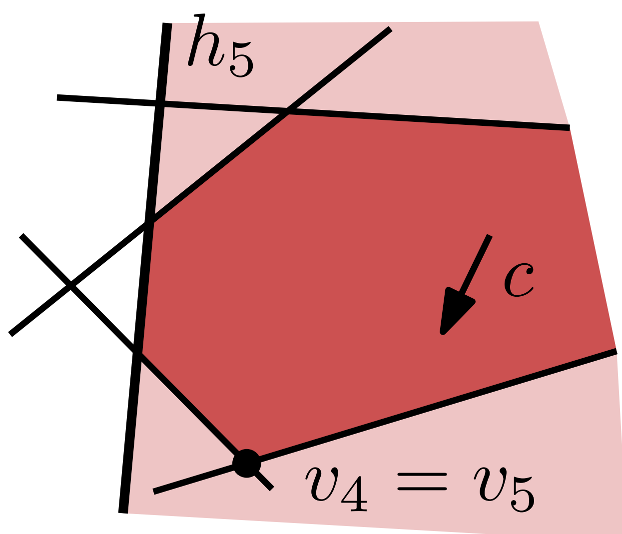
- each region C_i has a single optimal vertex v_i
- it holds that: $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

How the optimal vertex v_{i-1} changes when the half plane h_i is added?

Lemma 1: For $1 \leq i \leq n$ and bounding line ℓ_i of h_i holds that:

1. If $v_{i-1} \in h_i$ then $v_i = v_{i-1}$,
2. otherwise, either $C_i = \emptyset$ or $v_i \in \ell_i$.





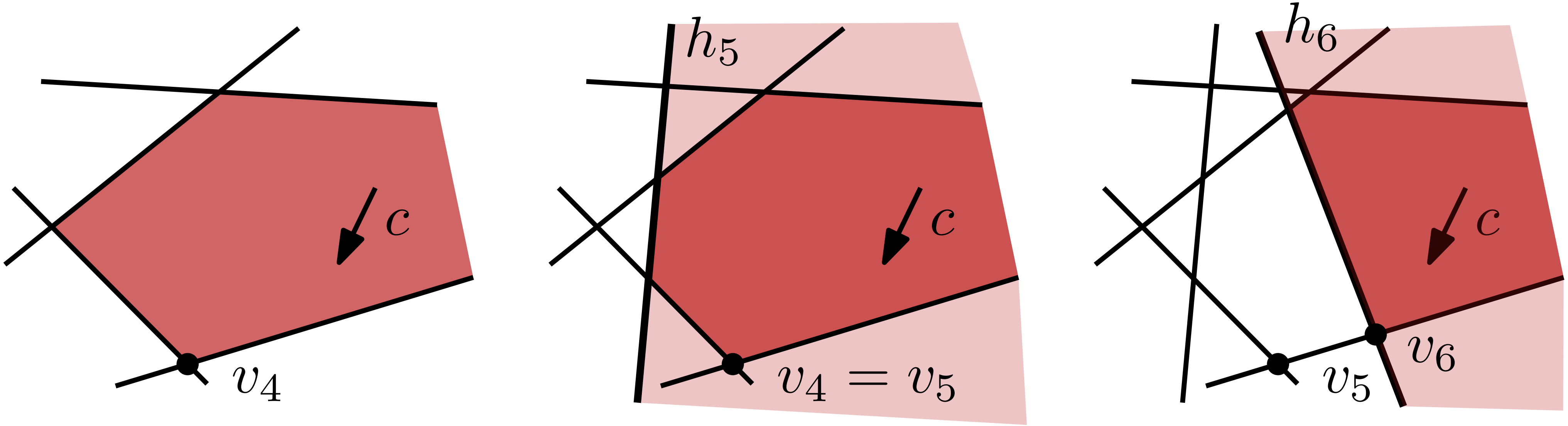
Properties

- each region C_i has a single optimal vertex v_i
- it holds that: $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

How the optimal vertex v_{i-1} changes when the half plane h_i is added?

Lemma 1: For $1 \leq i \leq n$ and bounding line ℓ_i of h_i holds that:

1. If $v_{i-1} \in h_i$ then $v_i = v_{i-1}$,
2. otherwise, either $C_i = \emptyset$ or $v_i \in \ell_i$.



Randomized incremental algorithm

2dRandomizedBoundedLP(H, c, m_1, m_2)

$C_0 \leftarrow m_1 \cap m_2$
$v_0 \leftarrow$ unique angle of C_0
$H \leftarrow$ **RandomPermutation**(H)
for $i \leftarrow 1$ **to** n **do**
  if $v_{i-1} \in h_i$ **then**
    $v_i \leftarrow v_{i-1}$
  else
    $v_i \leftarrow$ 1dBoundedLP$(\sigma(H_{i-1}), f_c^i)$
    if $v_i =$ nil **then**
      return infeasible
return v_n

Fisher-Yates Shuffle

Proof of Correctness:

a) Prove that each permutation of A has the same probability to be chosen.

RandomPermutation(A)

Input: Array $A[1 \ldots n]$
Output: Array A, rearranged into a random permutation
for $k \leftarrow n$ **to** 2 **do**
  $r \leftarrow$ Random(k)
  exchange $A[r]$ and $A[k]$

Fisher-Yates Shuffle

Proof of Correctness:

a) Prove that each permutation of A has the same probability to be chosen.

RandomPermutation(A)

Input: Array $A[1 \dots n]$
Output: Array A, rearranged into a random permutation
for $k \leftarrow n$ **to** 2 **do**
  $r \leftarrow$ Random(k)
  exchange $A[r]$ and $A[k]$

b) Prove, that the stamtent of a) if not true, if we replace k by n in the second line.

Fisher-Yates Shuffle

RandomPermutation(A)

Input: Array $A[1 \ldots n]$
Output: Array A, rearranged into a random permutation
for $k \leftarrow 2$ **to** n **do**
  $r \leftarrow$ Random(k)
  exchange $A[r]$ and $A[k]$

Each permutation of A has the same probability to be chosen.

Proof by induction:

- $A[1]$ is uniformly distributed
- $A[1, \ldots, n-1]$ is uniformly distributed
- $A[n]$ is chosen uniformly at random

Fisher-Yates Shuffle

RandomPermutation(A)

Input: Array $A[1 \dots n]$
Output: Array A, rearranged into a random permutation
for $k \leftarrow 2$ **to** n **do**
    $r \leftarrow$ Random(n)
    exchange $A[r]$ and $A[k]$

The permutations of A are not chosen with the same probability.

- the algorithm uniformly generates n^n (non-distinct) permutations
- there are $n!$ distinct permutations
- since $n-1$ does not divide n, n^n is not a multiple of $n!$