Final

Voronoi-Delaunay-Path Planning

Dimitrios Gangas
Theodore Lymperopoulos

National and Kapodistrian
University of Athens

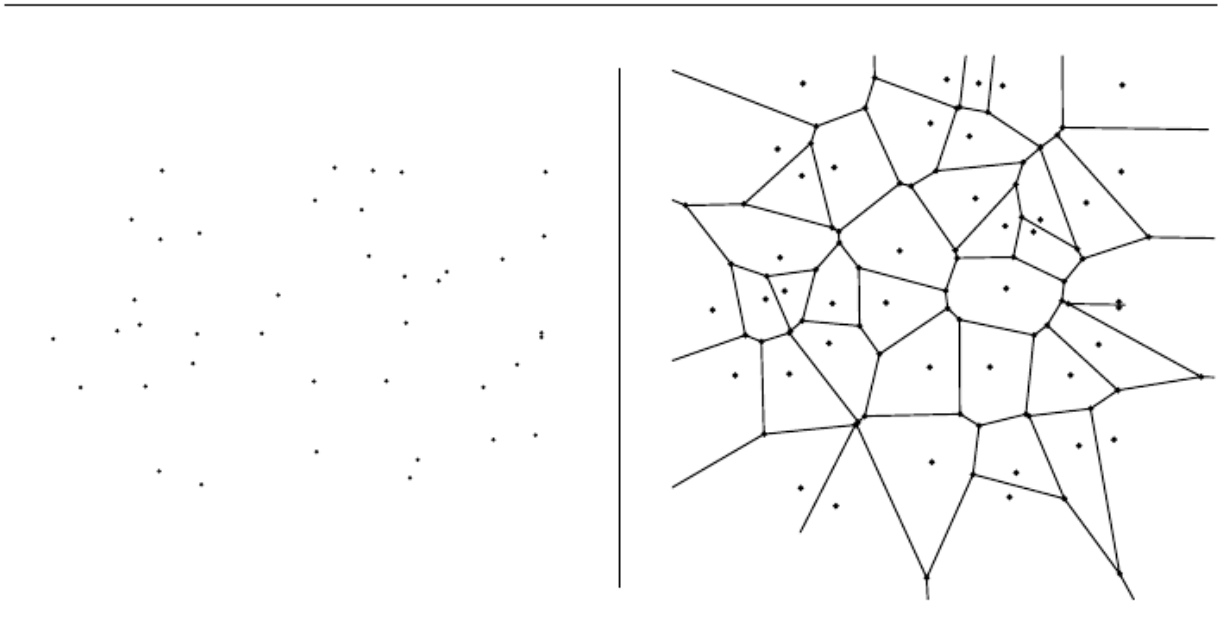
May 9, 2019

Contents

Voronoi

Overview

In mathematics, a Voronoi diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. That set of points (called seeds, sites, or generators) is specified beforehand, and for each seed there is a corresponding region consisting of all points closer to that seed than to any other. These regions are called Voronoi cells. The Voronoi diagram of a set of points is dual to its Delaunay triangulation.

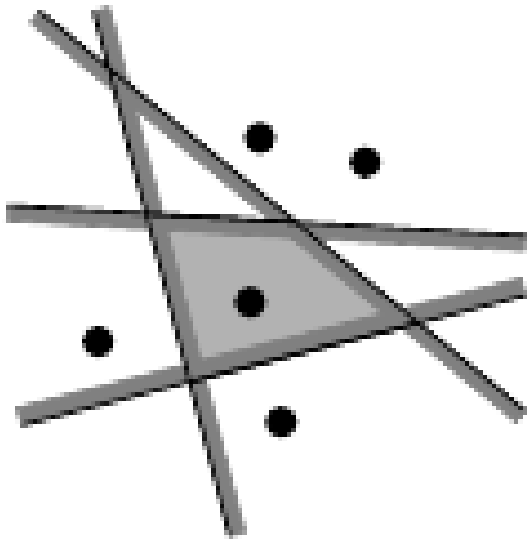


Formula

- The cell of Vor(P) that corresponds to a site pi is denoted V(pi).
- For two points p and q in the plane we define the bisector of p and q as the perpendicular bisector of the line segment $\overline{pq}$.

Observation: This bisector splits the plane into two half-planes.

$$V(p_i) = \cap_{1 \leq j \leq n, j \neq i} h(p_i, p_j)$$



Thus V(pi) is the intersection of $n - 1$ half-planes and, hence, a (possibly unbounded) open convex polygonal region bounded by at most $n - 1$ vertices and at most $n - 1$ edges.

Applications

Nearest neighbor search

Finding the nearest neighbor of query point q from among a fixed set of points S is simply a matter of determining the cell in the Voronoi diagram of S that contains q.

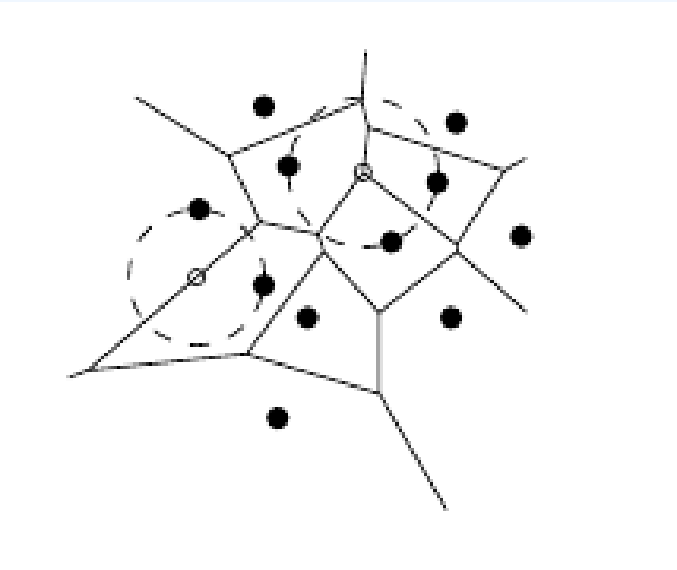
Facility location

Suppose McDonald's wants to open another restaurant. To minimize interference with existing McDonald's, it should be located as far away from the closest restaurant as possible. This location is always at a vertex of the Voronoi diagram, and can be found in a linear-time search through all the Voronoi vertices.

Applications

Largest empty circle

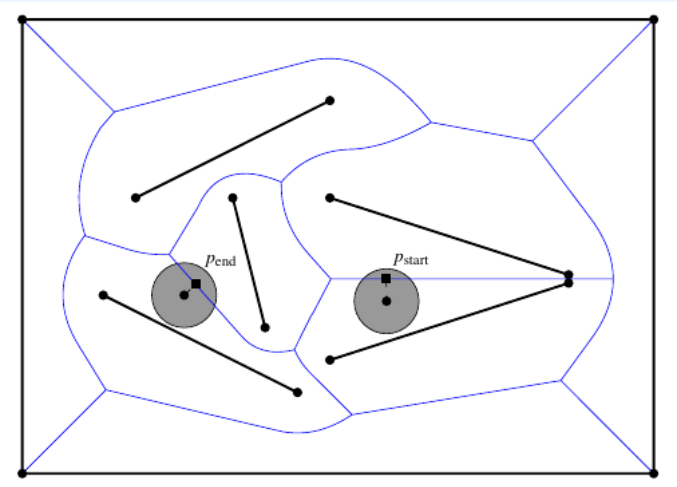
Suppose you needed to obtain a large, contiguous, un-developed piece of land on which to build a factory. The same condition used to select McDonald's locations is appropriate for other undesirable facilities, namely that they be as far as possible from any relevant sites of interest. A Voronoi vertex defines the center of the largest empty circle among the points.



Applications

Path planning

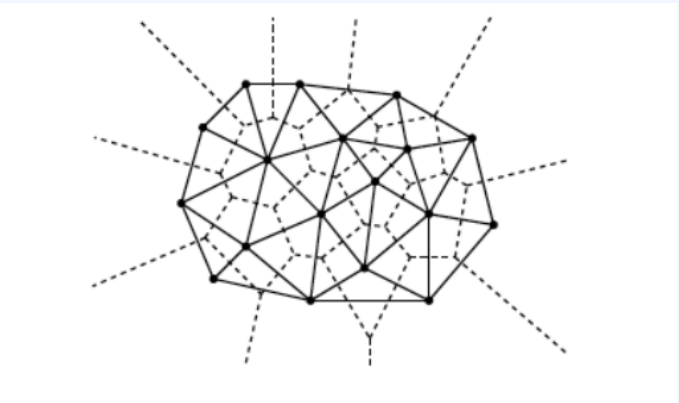
If the sites of S are the centers of obstacles we seek to avoid, the edges of the Voronoi diagram define the possible channels that maximize the distance to the obstacles. Thus the “safest” path among the obstacles will stick to the edges of the Voronoi diagram.



Applications

Quality triangulations

In triangulating a set of points, we often desire nice, fat triangles that avoid small angles and skinny triangles. The Delaunay triangulation maximizes the minimum angle over all triangulations. Furthermore, it is easily constructed as the dual of the Voronoi diagram.



Algorithms

Computing the Voronoi Diagram

Several $O(n^2)$ incremental algorithms were known for many years

- Divide-and-Conquer (1975, Shamos Hoey)
  - Works in O(nlog n) time
  - But it is somewhat complex to understand and implement
- Plane sweep (1987, Fortune)
  - Works in O(nlog n) time
  - Was tricky to invent
  - (will be discussed in depth later)
- Randomized incremental algorithm (1992, Guibas, Knuth Sharir)
- Transformation to Convex hull of points in 3D

Plane sweep (Fortune's Algorithm)

The strategy in a plane sweep algorithm is to sweep a horizontal line the sweep line from top to bottom over the plane. While the sweep is performed information is maintained regarding the structure that one wants to compute.

But which information should we keep?

Plane sweep (Fortune's Algorithm)

- Instead of maintaining the intersection of the Voronoi diagram with the sweep line, we maintain information about the part of the Voronoi diagram of the sites above ℓ that cannot be changed by sites below ℓ.
- The locus of points that are closer to some site $p_i \in S$ than to ℓ is bounded by a parabola. Hence, the locus of points that are closer to any site above ℓ than to ℓ itself is bounded by parabolic arcs. We call this sequence of parabolic arcs the beach line.

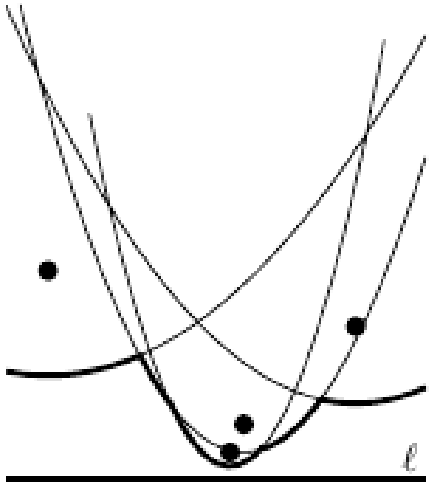


Figure: The Beach line

Plane sweep (Fortune's Algorithm)

The Beach Line

- **Lemma** The beach line is an x-monotone curve made up of parabolic arcs. The breakpoints (vertices) are incident to (move along) Voronoi edges.
- The breakpoints move along bisectors and trace out the Voronoi edges.

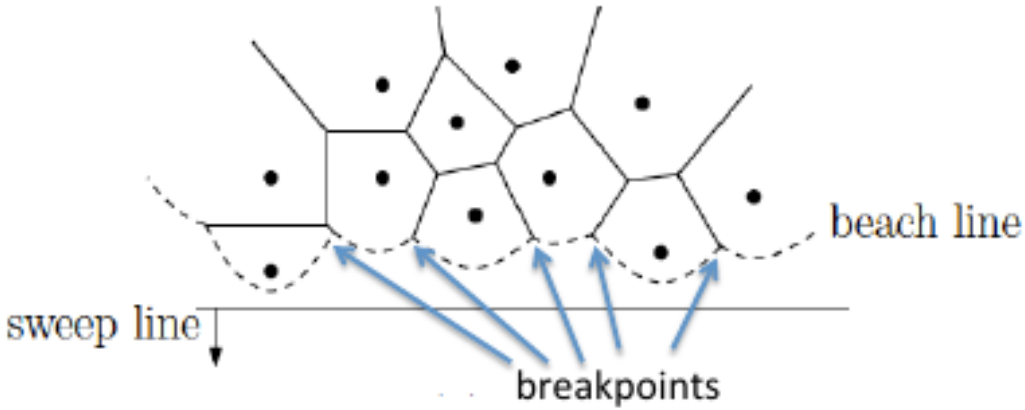


Figure: The Beach line II

Plane sweep (Fortune's Algorithm)

Events

There are two types of events:

1. **Site events** (insert a new arc) When the sweep line sees a new site a new arc is inserted into the beach line.
2. **Voronoi vertex events** (delete arc): When the length of a beachline arc shrinks to zero, the arc disappears a new Voronoi vertex is determined

Plane sweep (Fortune's Algorithm)

Algorithm VORONOIDIAGRAM(P)
Input. A set $P := \{p_1, \ldots, p_n\}$ of point sites in the plane.
Output. The Voronoi diagram Vor(P) given inside a bounding box in a doubly-connected edge list $\mathcal{D}$.

1. Initialize the event queue $\mathcal{Q}$ with all site events, initialize an empty status structure $\mathcal{T}$ and an empty doubly-connected edge list $\mathcal{D}$.
2. **while** $\mathcal{Q}$ is not empty
3. **do** Remove the event with largest y-coordinate from $\mathcal{Q}$.
4. **if** the event is a site event, occurring at site p_i
5. **then** HANDLESITEEVENT(p_i)
6. **else** HANDLECIRCLEEVENT(γ), where γ is the leaf of $\mathcal{T}$ representing the arc that will disappear
7. The internal nodes still present in $\mathcal{T}$ correspond to the half-infinite edges of the Voronoi diagram. Compute a bounding box that contains all vertices of the Voronoi diagram in its interior, and attach the half-infinite edges to the bounding box by updating the doubly-connected edge list appropriately.
8. Traverse the half-edges of the doubly-connected edge list to add the cell records and the pointers to and from them.

Figure: Fortune pseudocode

Data Structures to Maintain

1. Partial Voronoi diagram as DCEL.
2. Beach line as a balanced binary tree.
   - Stores sorted sequence of sites whose arcs form the beach line. We don't explicitly store the parabolic arcs.
3. Event queue as a priority queue.
   - each event has a cross link back to the triple of sites that generated it and vice versa

What the facts

British physician John Snow used a Voronoi diagram in 1854 to illustrate how the majority of people who died in the Broad Street cholera outbreak lived closer to the infected Broad Street pump than to any other water pump.

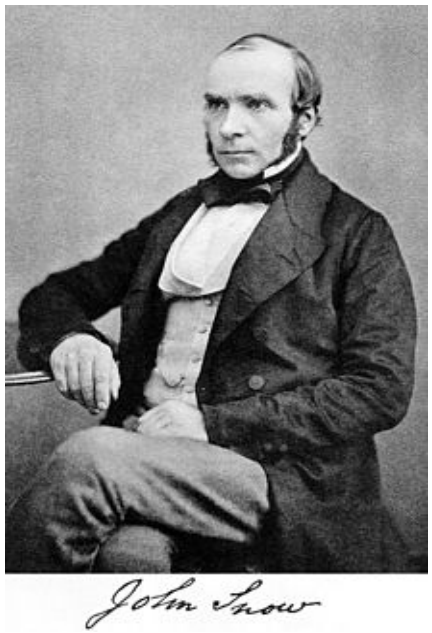


Figure: John Snow
(The physician)

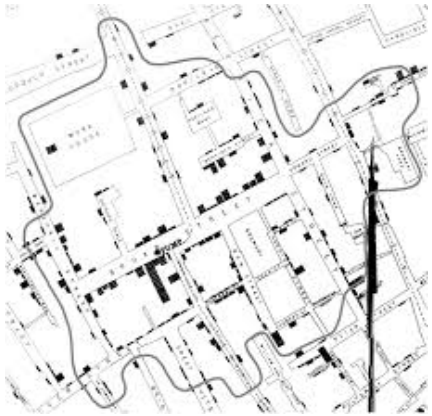


Figure: district in London

Voronoi in Nature

Figure: Giraffe

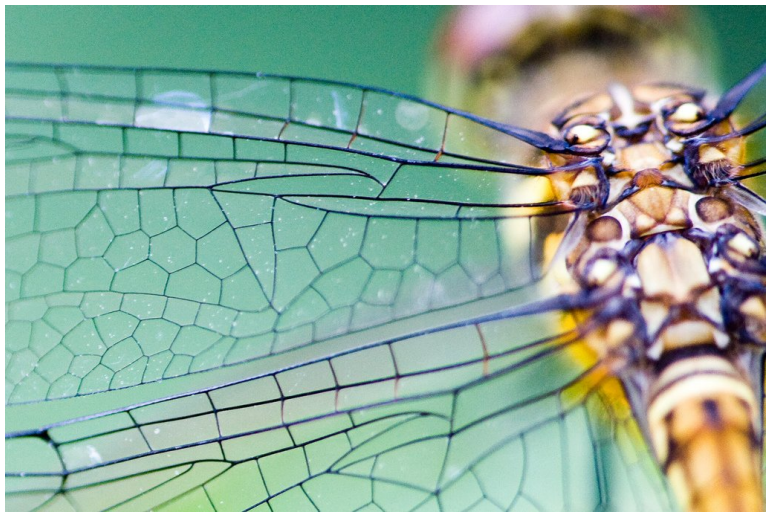


Figure: Flying insects

Voronoi in Nature

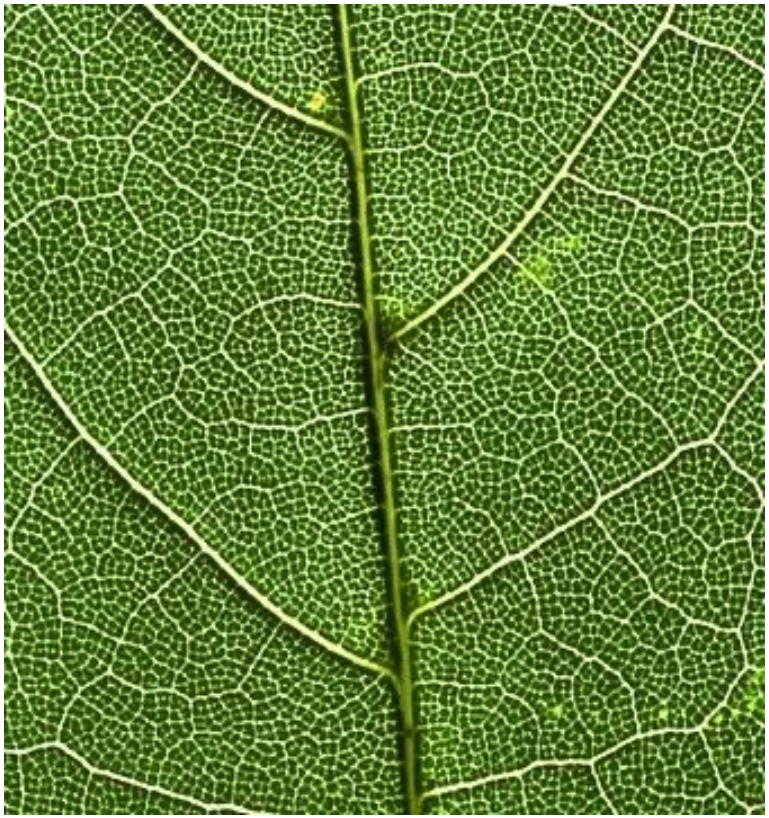


Figure: Leaf

Figure: Sea turtle

Voronoi in Nature

Figure: Honeycomb

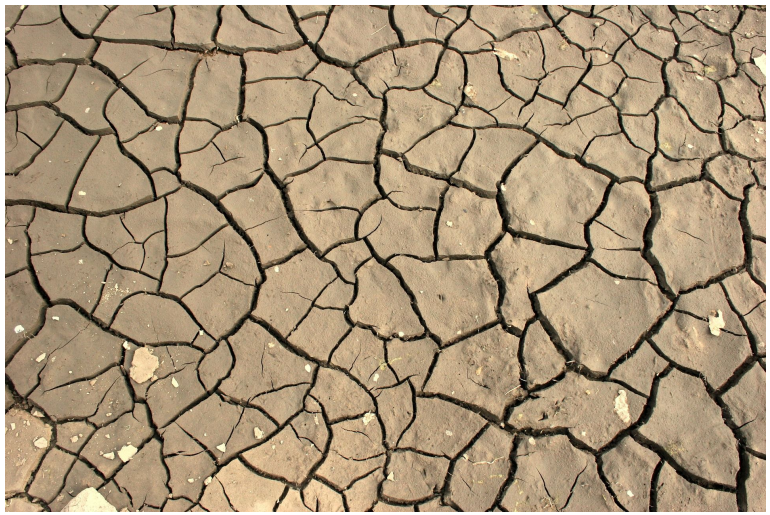


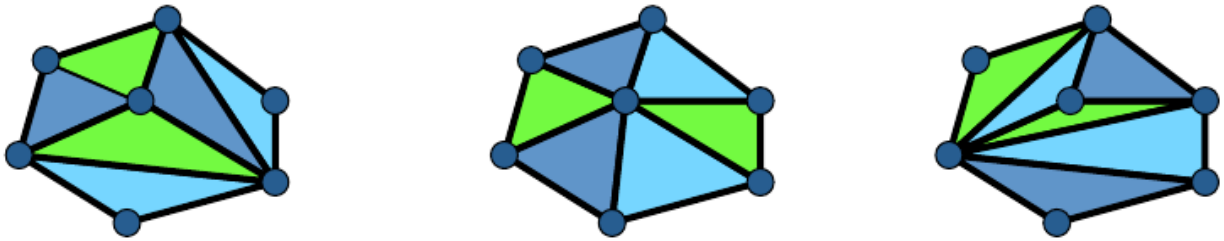
Figure: Ground

Delaunay

General Idea

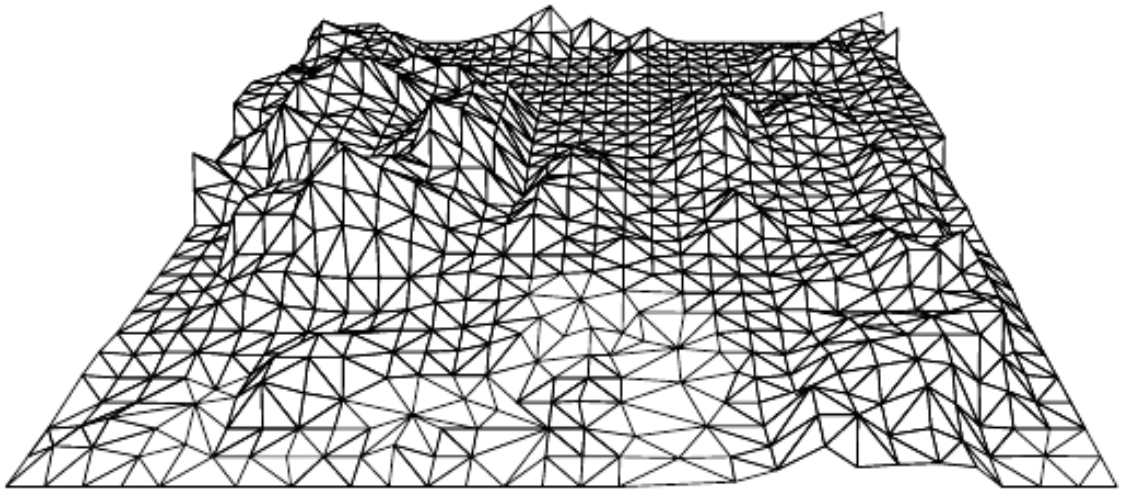
Triangulation

- of a set S of points in the plane is a partition of the convex hull of S into triangles whose vertices are the points, and do not contain other points.
- There are an exponential number of triangulations of a point set.



Motivation Example

- We can model a piece of the earth's surface as a terrain. A terrain is a 2-dimensional surface in 3-dimensional space with a special property. It is the graph of a function $f : A \subset \mathcal{R}^2 \rightarrow \mathcal{R}$ that assigns a height $f(p)$ to every point p in the domain, A, of the terrain.
- A terrain can be visualized with a perspective drawing.



Motivation Example

- We only know it where we've measured it. This means that when we talk about some terrain, we only know the value of the function f at a finite set $\mathcal{P} \subset \mathcal{A}$ of sample points. From the height of the sample points we somehow have to approximate the height at the other points in the domain.
- We first determine a triangulation of $\mathcal{P}$.
- We then lift each sample point to its correct height.

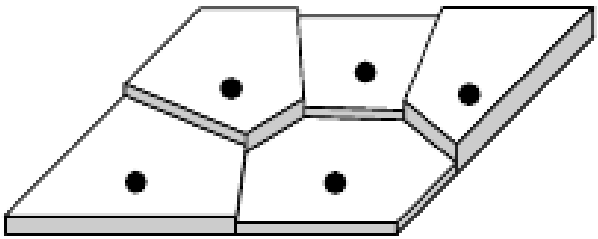


Figure: Discrete terrain

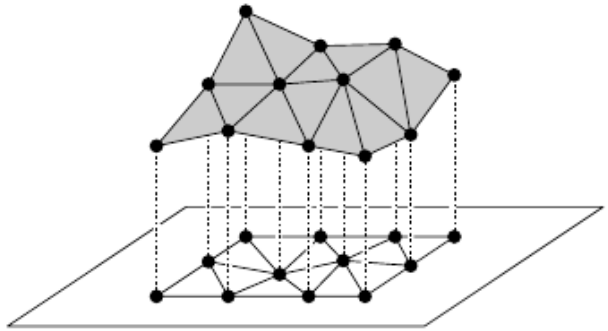


Figure: Obtaining a polyhedral terrain from a set of sample points

Motivation Example

The question remains: how do we triangulate the set of sample points?

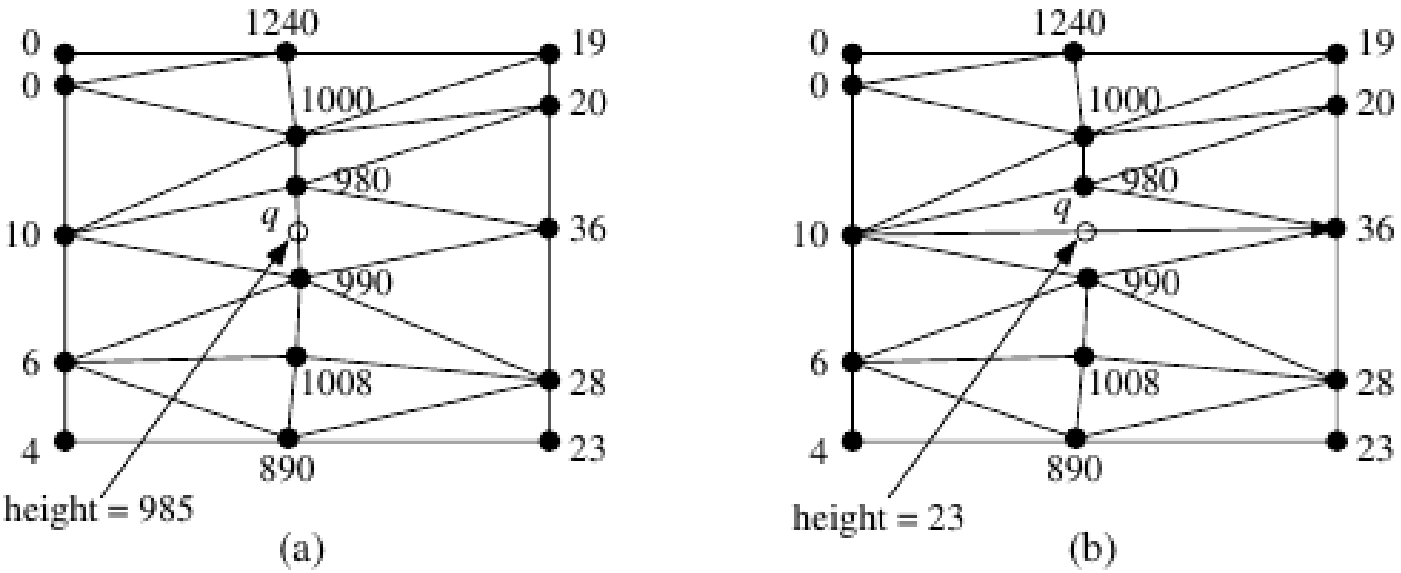


Figure: Flipping one edge can make a big difference

Motivation Example

The question remains: how do we triangulate the set of sample points?

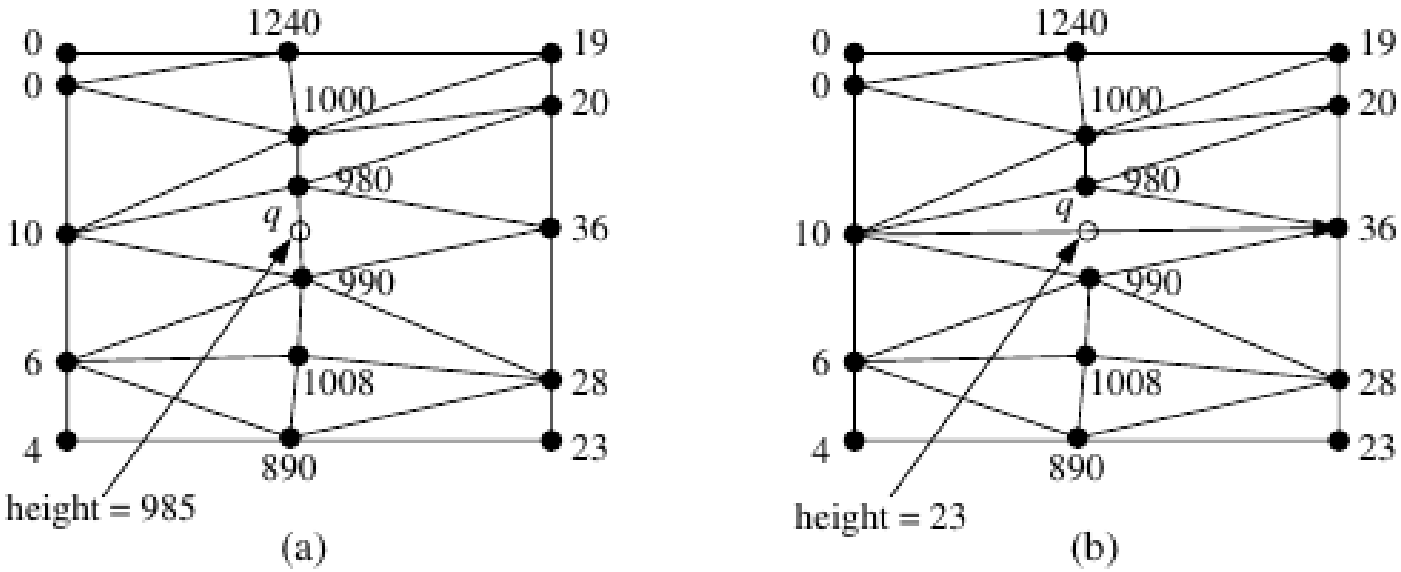


Figure: Flipping one edge can make a big difference

It seems that a triangulation that contains small angles is bad. Therefore we will rank triangulations by comparing their smallest angle.

Theorem

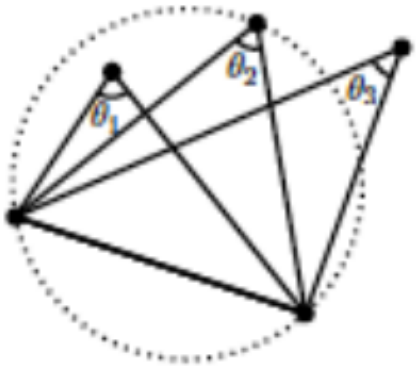
Theorem Let P be a set of n points in the plane, not all collinear, and let k denote the number of points in P that lie on the boundary of the convex hull of P. Then any triangulation of P has $2n - 2 - k$ triangles and $3n - 3 - k$ edges.

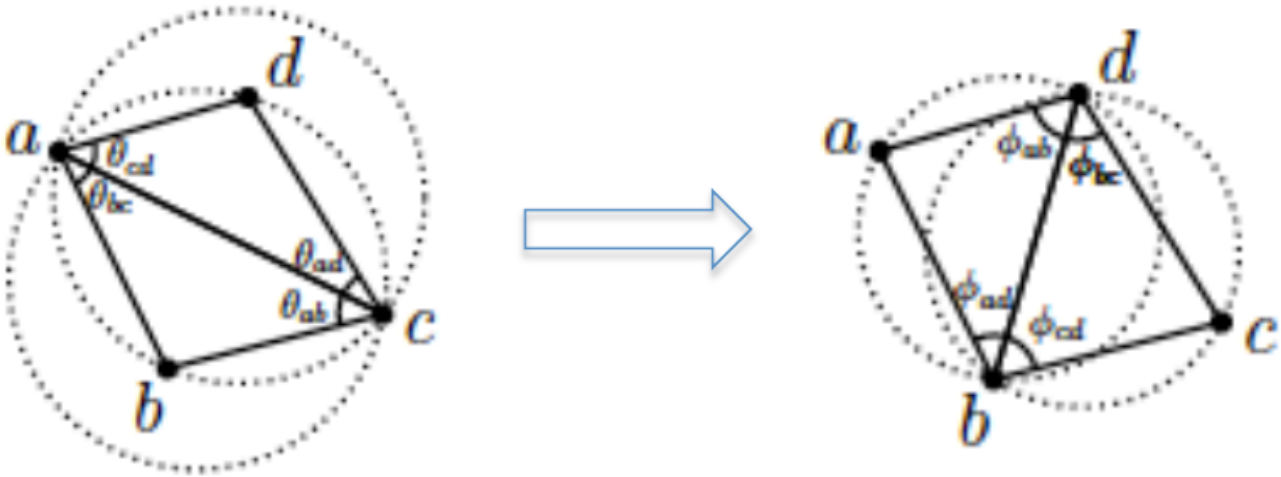
Let $\mathcal{T}$ be a triangulation of P, and suppose it has m triangles. Consider the $3m$ angles of the triangles of $\mathcal{T}$, sorted by increasing value. Let $\alpha_1, \alpha_2, ..., \alpha_{3m}$ be the resulting sequence of angles; hence, $\alpha_i \leq \alpha_j$, for $i < j$. We call $A(\mathcal{T}) = (\alpha_1, \alpha_2, ..., \alpha_{3m})$ the angle-vector of $\mathcal{T}$. Let $\mathcal{T}$ be another triangulation of the same point set P, and let A($\mathcal{T}'$) := $(\alpha_1, \alpha_2, ..., \alpha_{3m})$ be its angle-vector. We say that the angle-vector of $\mathcal{T}$ is larger than the angle-vector of $\mathcal{T}'$ if A($\mathcal{T}$) is lexicographically larger than A($\mathcal{T}'$), or, in other words, if there exists an index i with $1 \leq i \leq 3m$ such that $\alpha_j = \alpha_{j'}$ for all j < i, and $\alpha_i > \alpha_{i'}$. We denote this as A($\mathcal{T}$) > A($\mathcal{T}'$). A triangulation $\mathcal{T}$ is called angle-optimal if $A(\mathcal{T}) \geq$ A($\mathcal{T}'$) for all triangulations $\mathcal{T}'$ of P.

Edge Flip

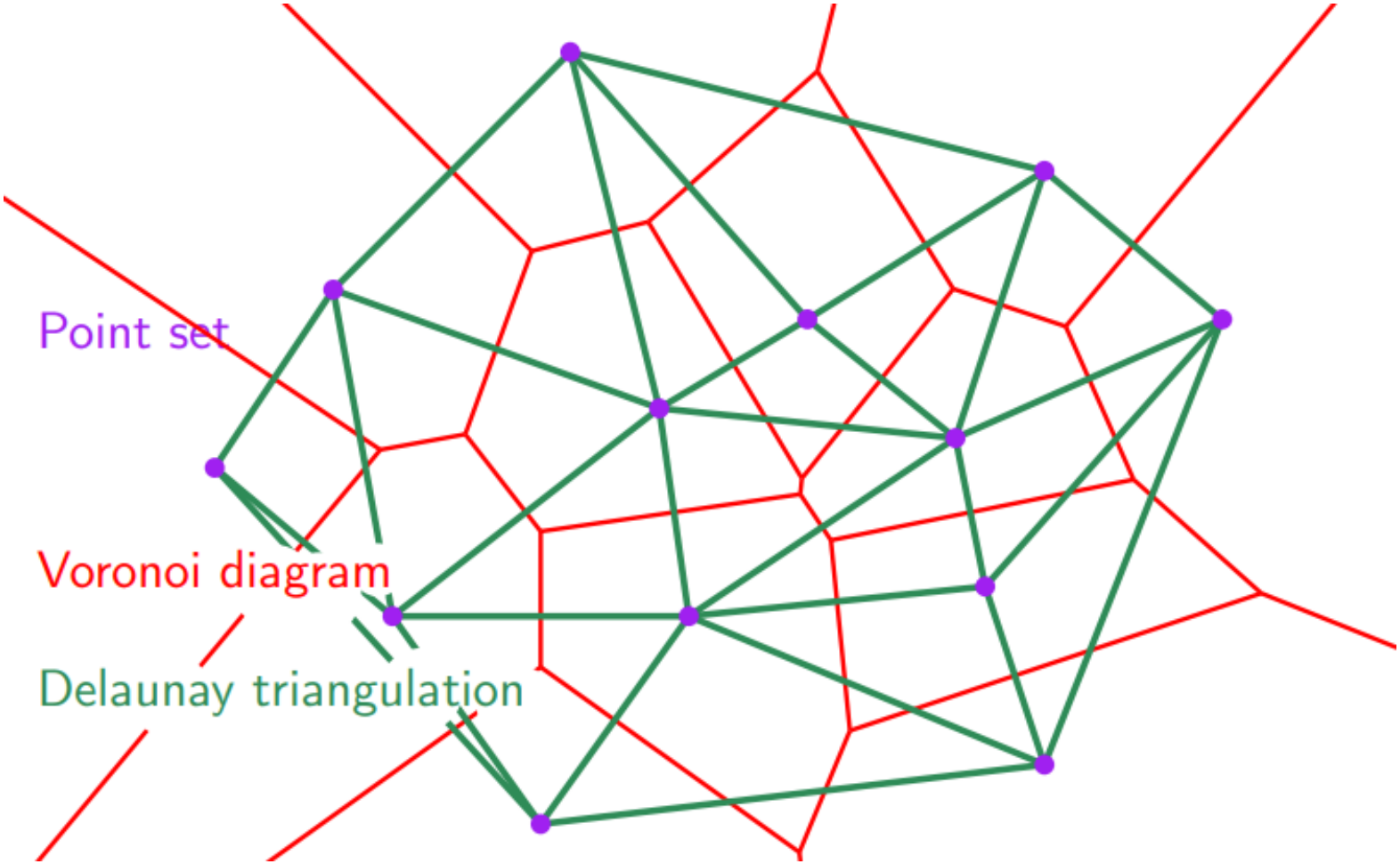
- If we flip the edge it will follow that the two new triangles have empty circumcircles.
- And the minimum angle of the pair of triangles will increase.

$\theta_1 > \theta_2 > \theta_3$

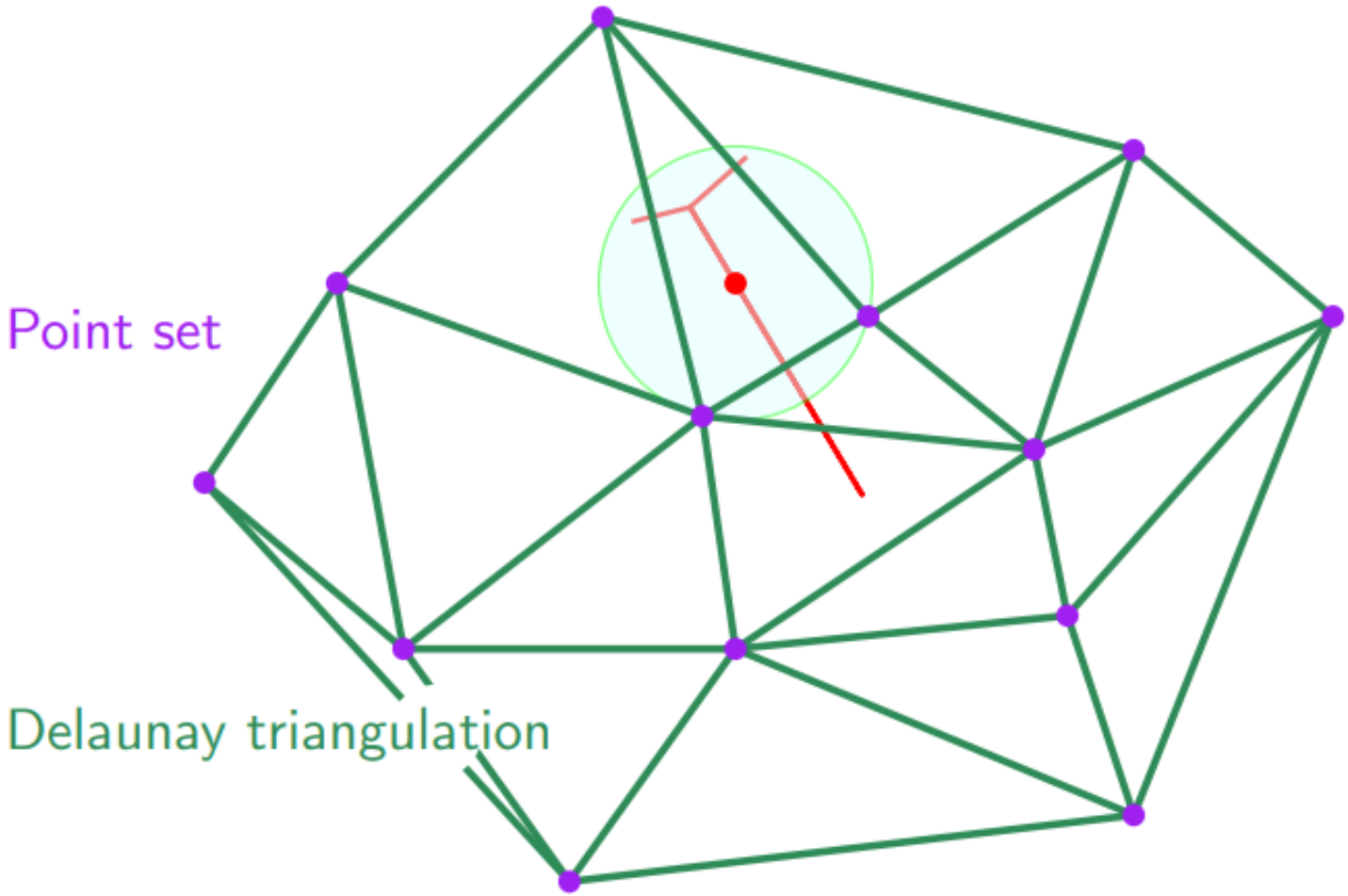




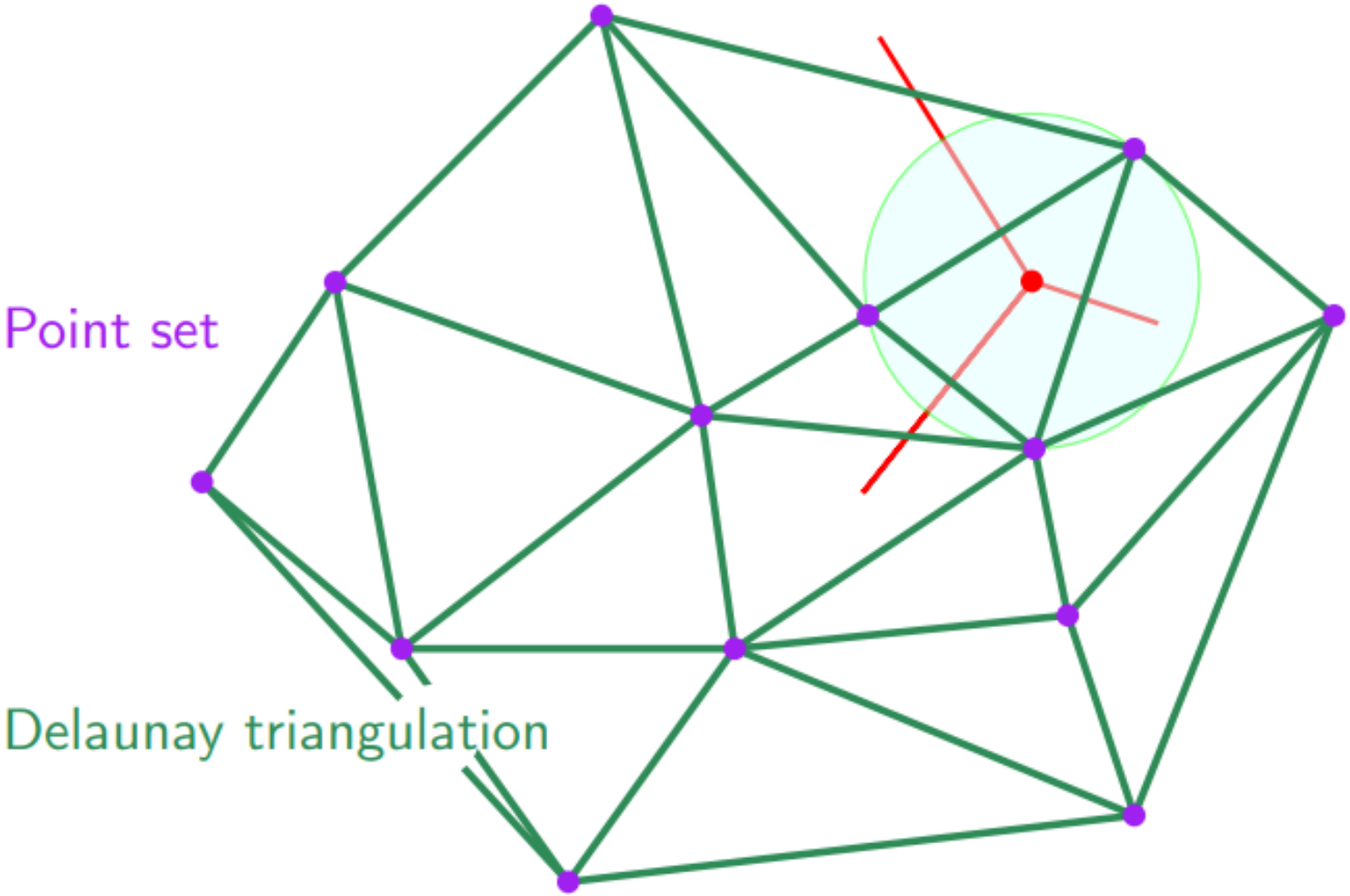
Properties - Duality



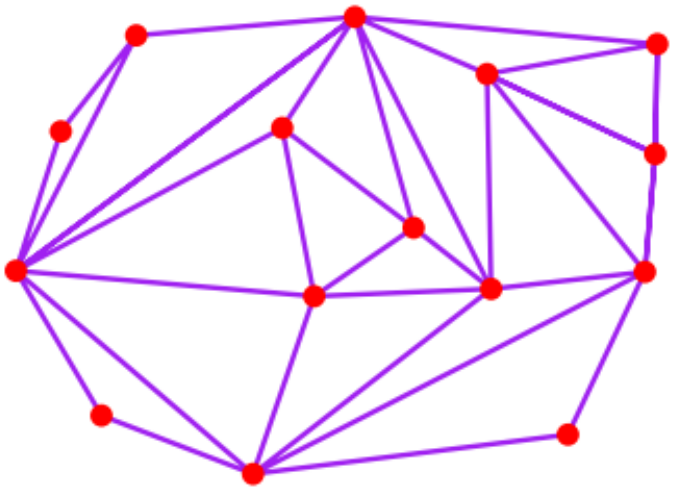
Properties - Largest empty circle [1]



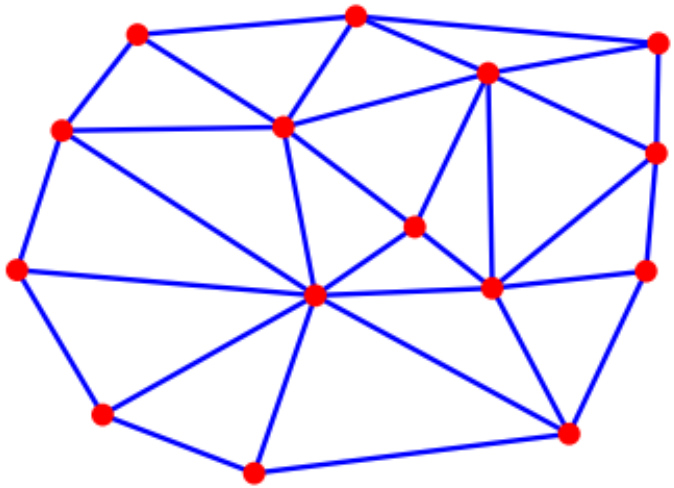
Properties - Largest empty circle [2]



Properties - Quality triangulations [1]



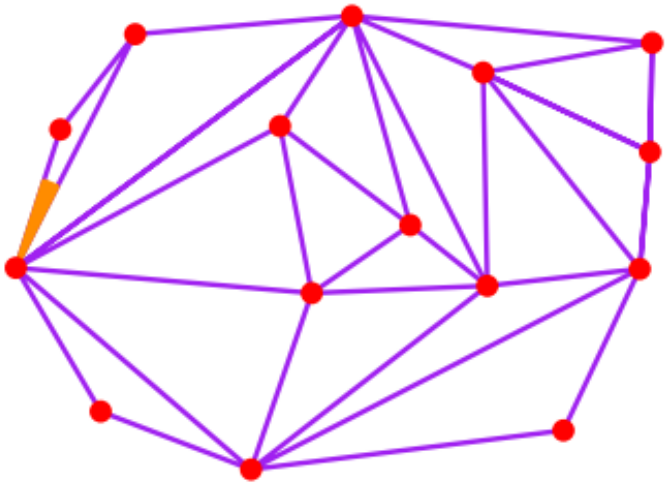
Triangulation



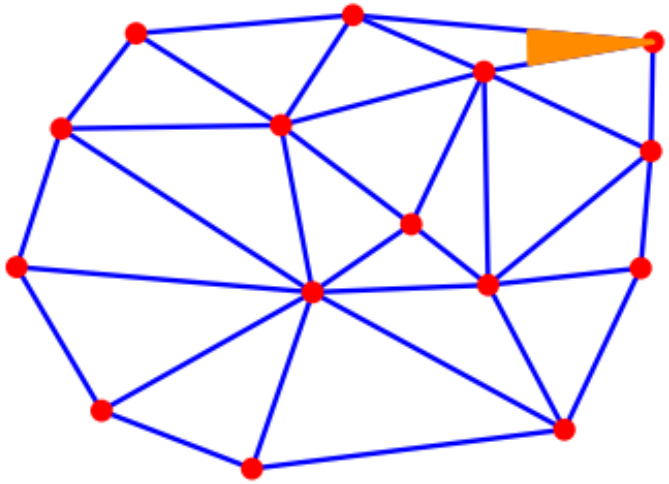
Delaunay

Triangulation 1 vs Delaunay Triangulation. The smallest angle is maximized.

Properties - Quality triangulations [2]



Triangulation

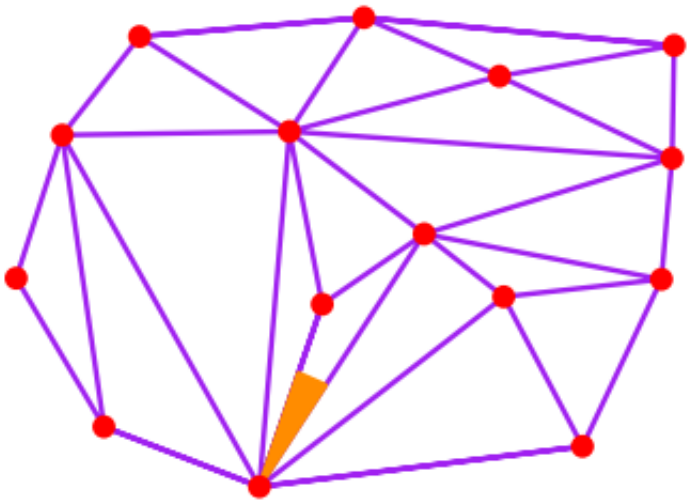


Delaunay

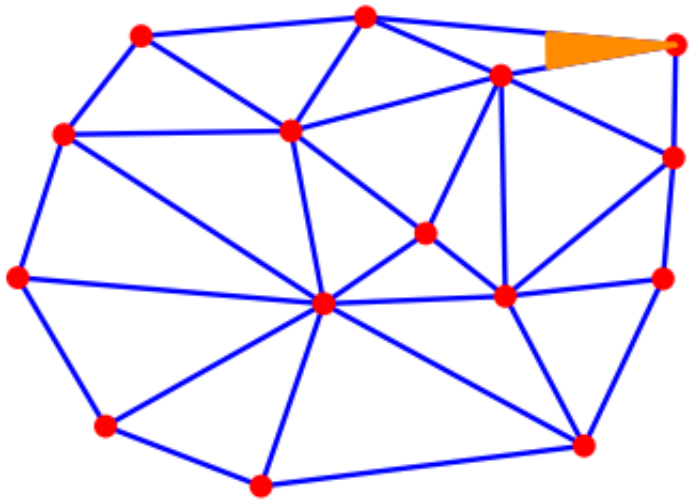
smallest angle

Triangulation 1 vs Delaunay Triangulation.

Properties - Quality triangulations [3]



Triangulation

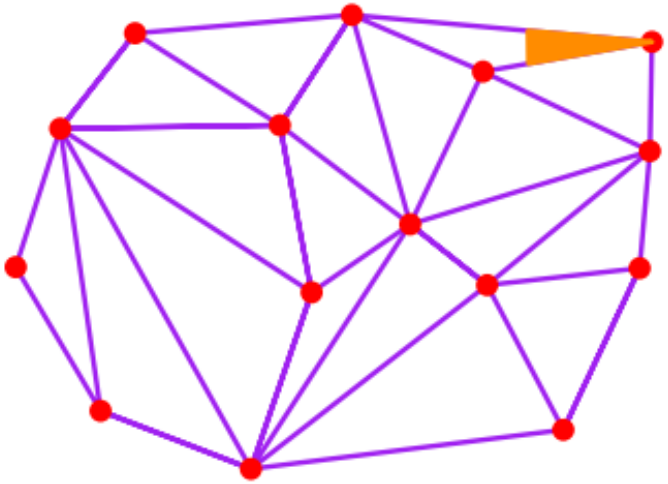


Delaunay

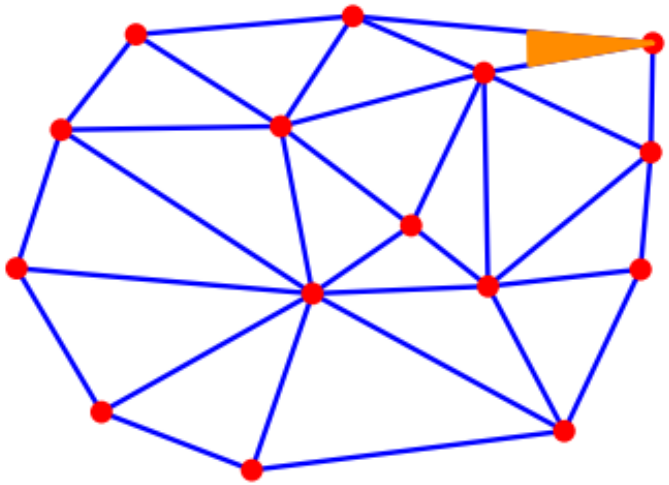
smallest angle

Triangulation 2 vs Delaunay Triangulation.

Properties - Quality triangulations [4]



Triangulation

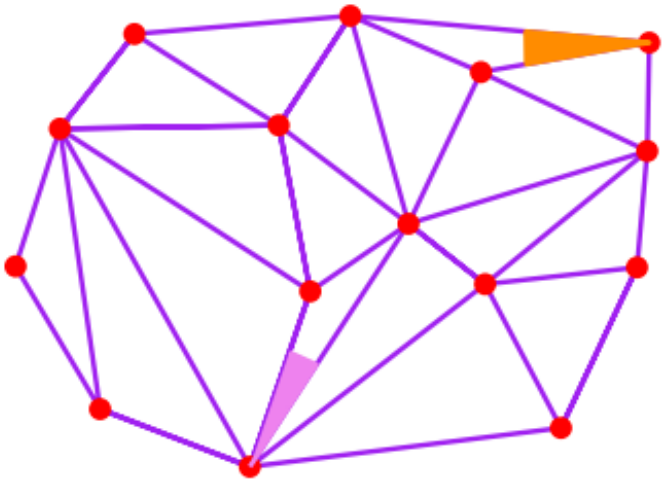


Delaunay

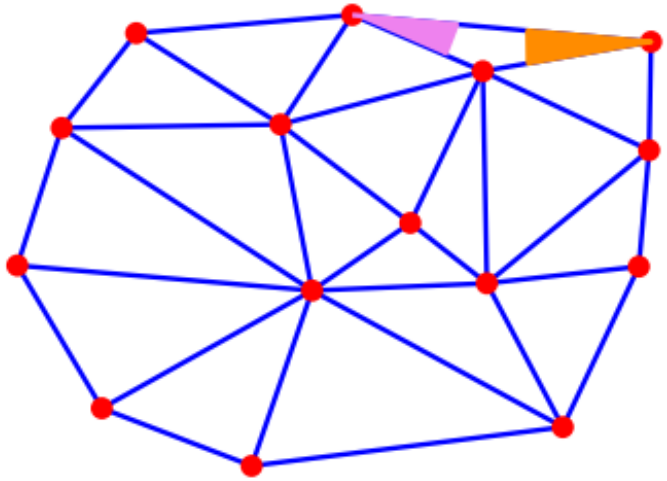
smallest angle

Triangulation 3 vs Delaunay Triangulation.

Properties - Quality triangulations [5]



Triangulation



Delaunay

smallest angle

second smallest angle

Since the smallest angles are the same, we check the next ones.

Algorithms

Computing the Delaunay triangulation

There are several ways to compute the Delaunay triangulation:

1. By iterative flipping from any triangulation
2. By plane sweep
3. By randomized incremental construction
4. By conversion from the Voronoi diagram

The last three run in $O(nlogn)$ time for n points in the plane

Randomized Incremental

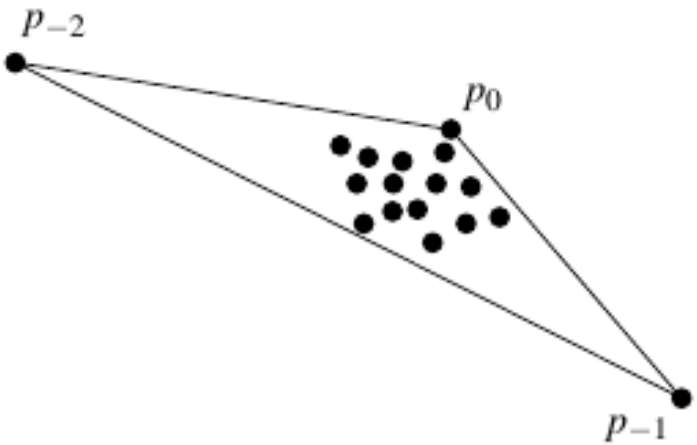
Algorithm DELAUNAYTRIANGULATION(P)
Input. A set P of $n+1$ points in the plane.
Output. A Delaunay triangulation of P.

1. Let p_0 be the lexicographically highest point of P, that is, the rightmost among the points with largest y-coordinate.
2. Let p_{-1} and p_{-2} be two points in $\mathbb{R}^2$ sufficiently far away and such that P is contained in the triangle $p_0p_{-1}p_{-2}$.
3. Initialize $\mathcal{T}$ as the triangulation consisting of the single triangle $p_0p_{-1}p_{-2}$.
4. Compute a random permutation $p_1, p_2, \ldots, p_n$ of $P \setminus \{p_0\}$.
5. **for** $r \leftarrow 1$ **to** n
6. **do** ($*$ Insert p_r into $\mathcal{T}$: $*$)
7. Find a triangle $p_ip_jp_k \in \mathcal{T}$ containing p_r.
8. **if** p_r lies in the interior of the triangle $p_ip_jp_k$
9. **then** Add edges from p_r to the three vertices of $p_ip_jp_k$, thereby splitting $p_ip_jp_k$ into three triangles.
10. LEGALIZEEDGE($p_r, \overline{p_ip_j}, \mathcal{T}$)
11. LEGALIZEEDGE($p_r, \overline{p_jp_k}, \mathcal{T}$)
12. LEGALIZEEDGE($p_r, \overline{p_kp_i}, \mathcal{T}$)
13. **else** ($*$ p_r lies on an edge of $p_ip_jp_k$, say the edge $\overline{p_ip_j}$ $*$)
14. Add edges from p_r to p_k and to the third vertex p_l of the other triangle that is incident to $\overline{p_ip_j}$, thereby splitting the two triangles incident to $\overline{p_ip_j}$ into four triangles.
15. LEGALIZEEDGE($p_r, \overline{p_ip_l}, \mathcal{T}$)
16. LEGALIZEEDGE($p_r, \overline{p_lp_j}, \mathcal{T}$)
17. LEGALIZEEDGE($p_r, \overline{p_jp_k}, \mathcal{T}$)
18. LEGALIZEEDGE($p_r, \overline{p_kp_i}, \mathcal{T}$)
19. Discard p_{-1} and p_{-2} with all their incident edges from $\mathcal{T}$.
20. **return** $\mathcal{T}$

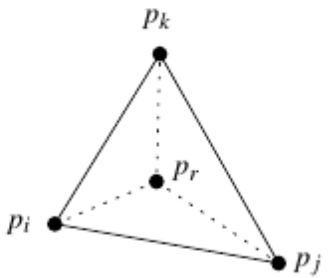
Randomized Incremental

LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)

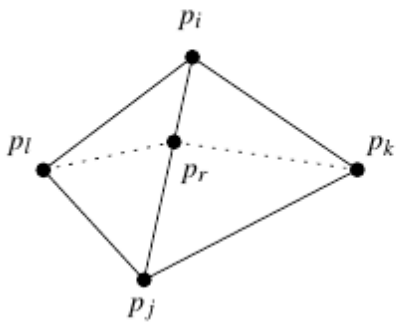
1. (∗ The point being inserted is p_r, and $\overline{p_i p_j}$ is the edge of $\mathcal{T}$ that may need to be flipped. ∗)
2. **if** $\overline{p_i p_j}$ is illegal
3. **then** Let $p_i p_j p_k$ be the triangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.
4. (∗ Flip $\overline{p_i p_j}$: ∗) Replace $\overline{p_i p_j}$ with $\overline{p_r p_k}$.
5. LEGALIZEEDGE($p_r, \overline{p_i p_k}, \mathcal{T}$)
6. LEGALIZEEDGE($p_r, \overline{p_k p_j}, \mathcal{T}$)



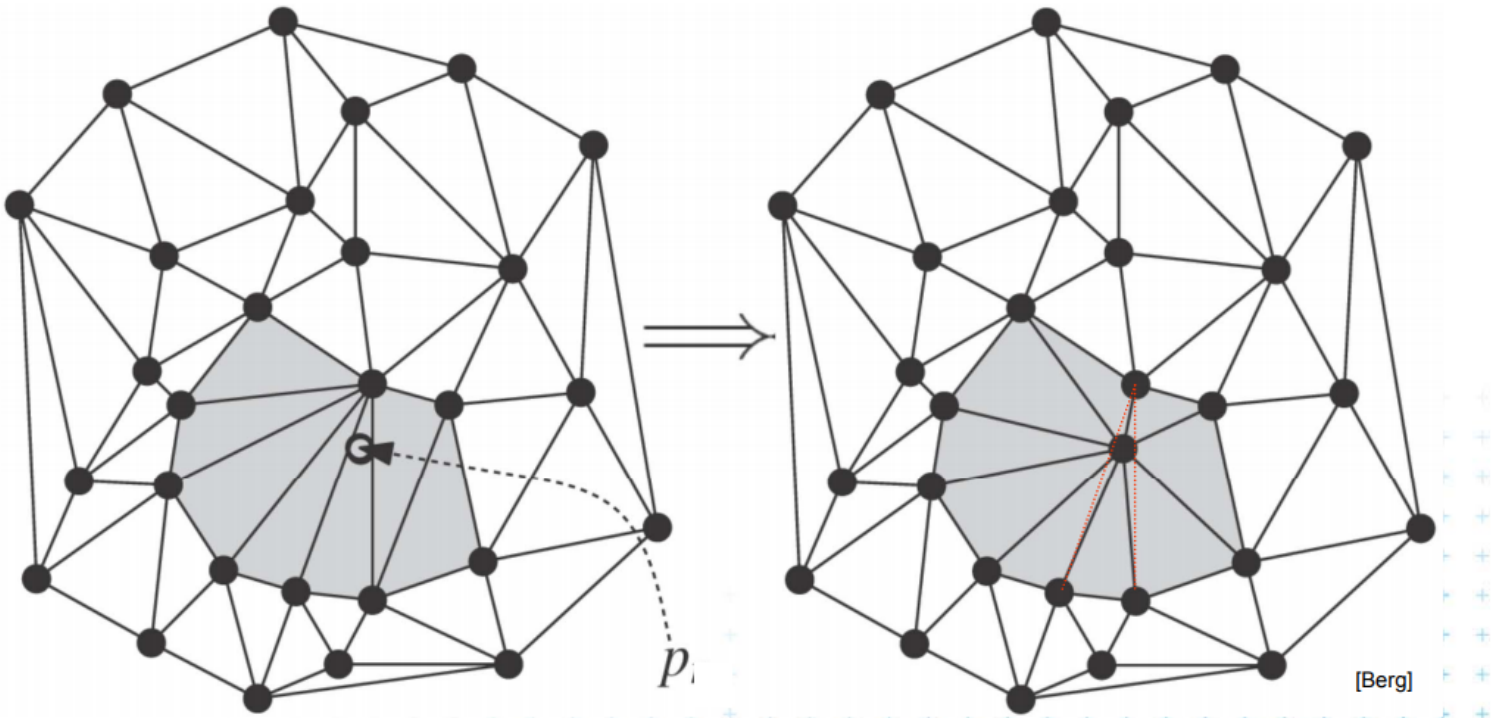
p_r lies in the interior of a triangle



p_r falls on an edge

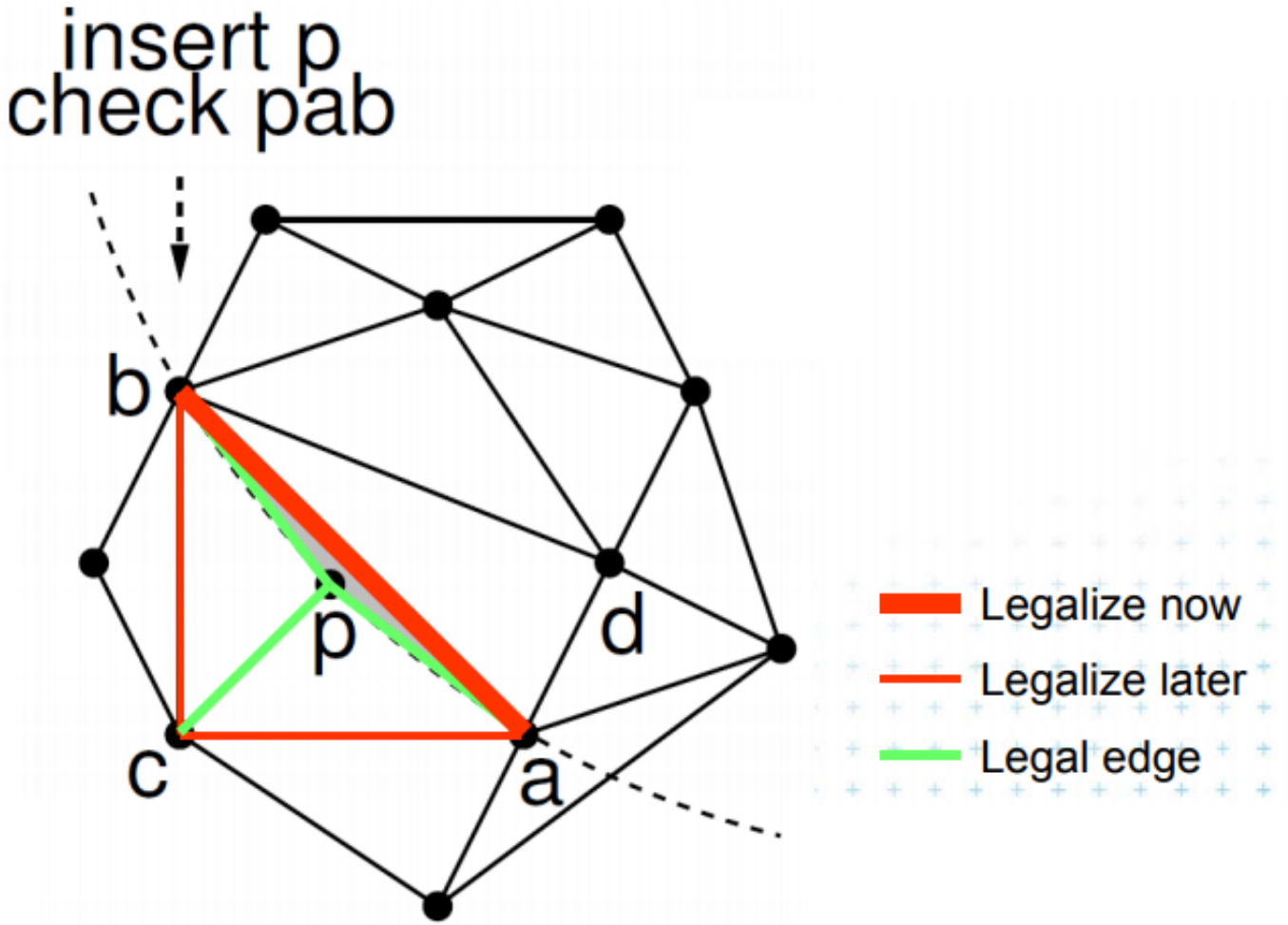


Randomized Incremental

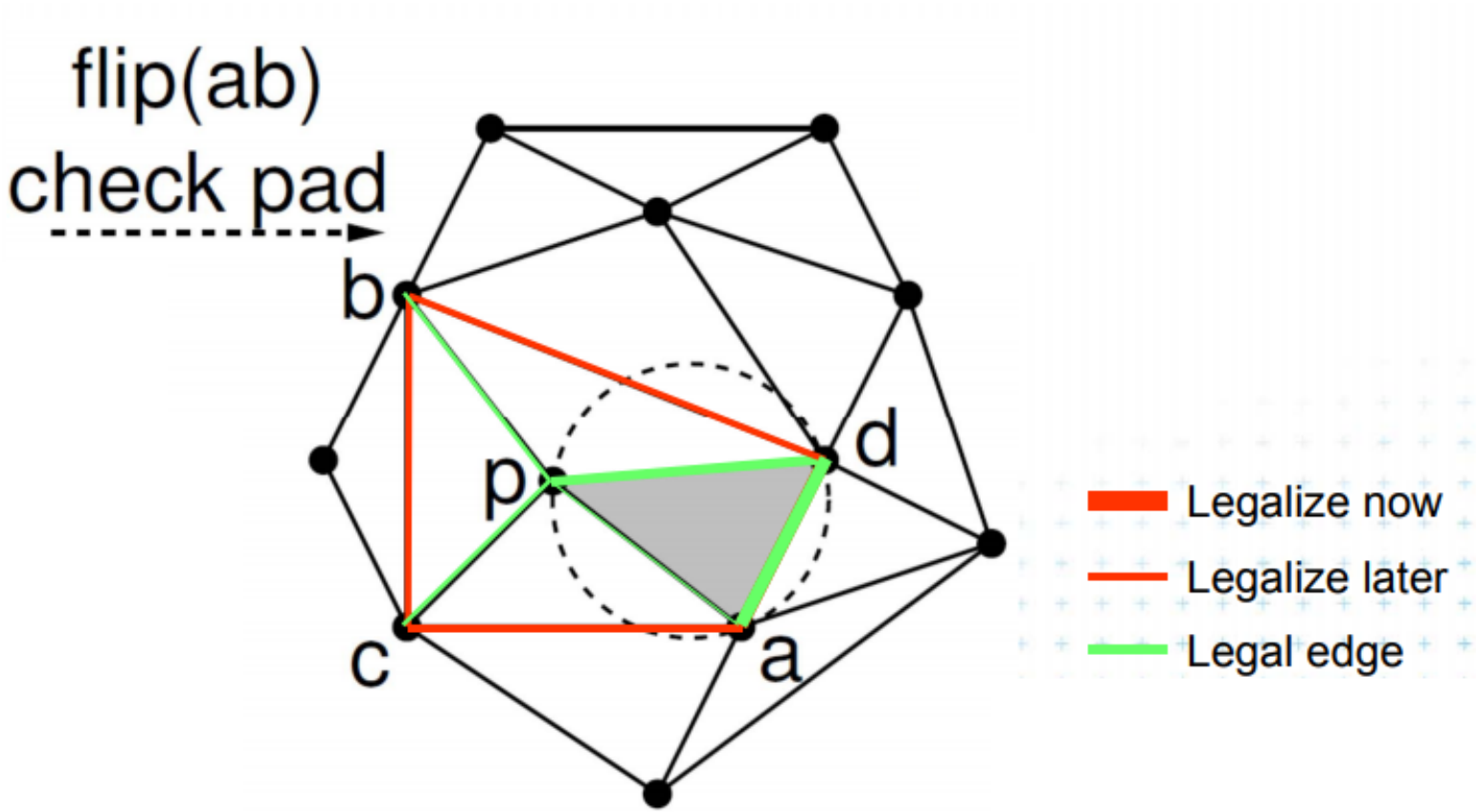


After the insertion of a new vertex, our graph needs to be modified in order to form a delaunay triangulation.

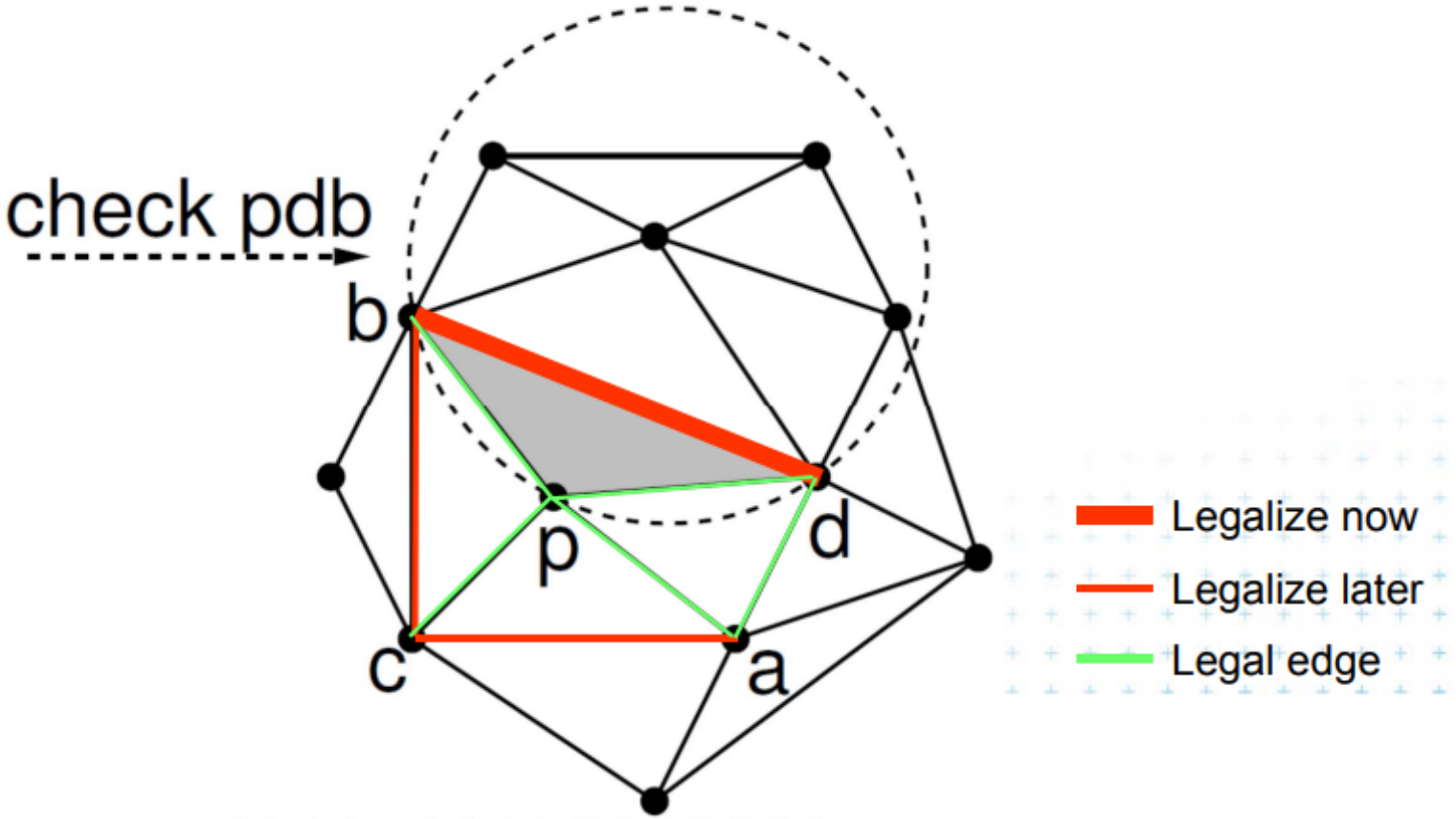
Step 1


insert p
check pab
b
p
d
c
a
Legalize now
Legalize later
Legal edge

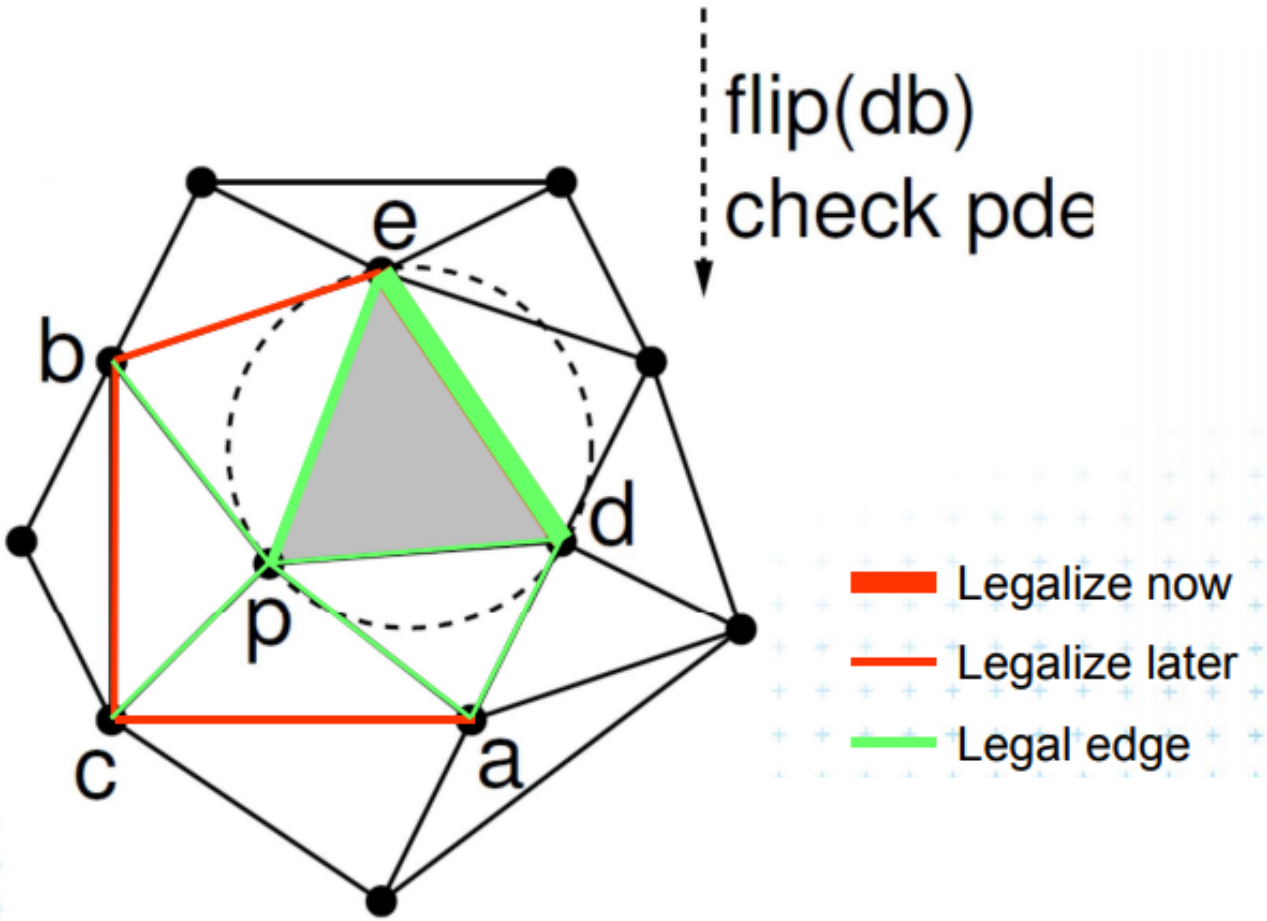
Step 2


flip(ab)
check pad
b
p
d
c
a
Legalize now
Legalize later
Legal edge

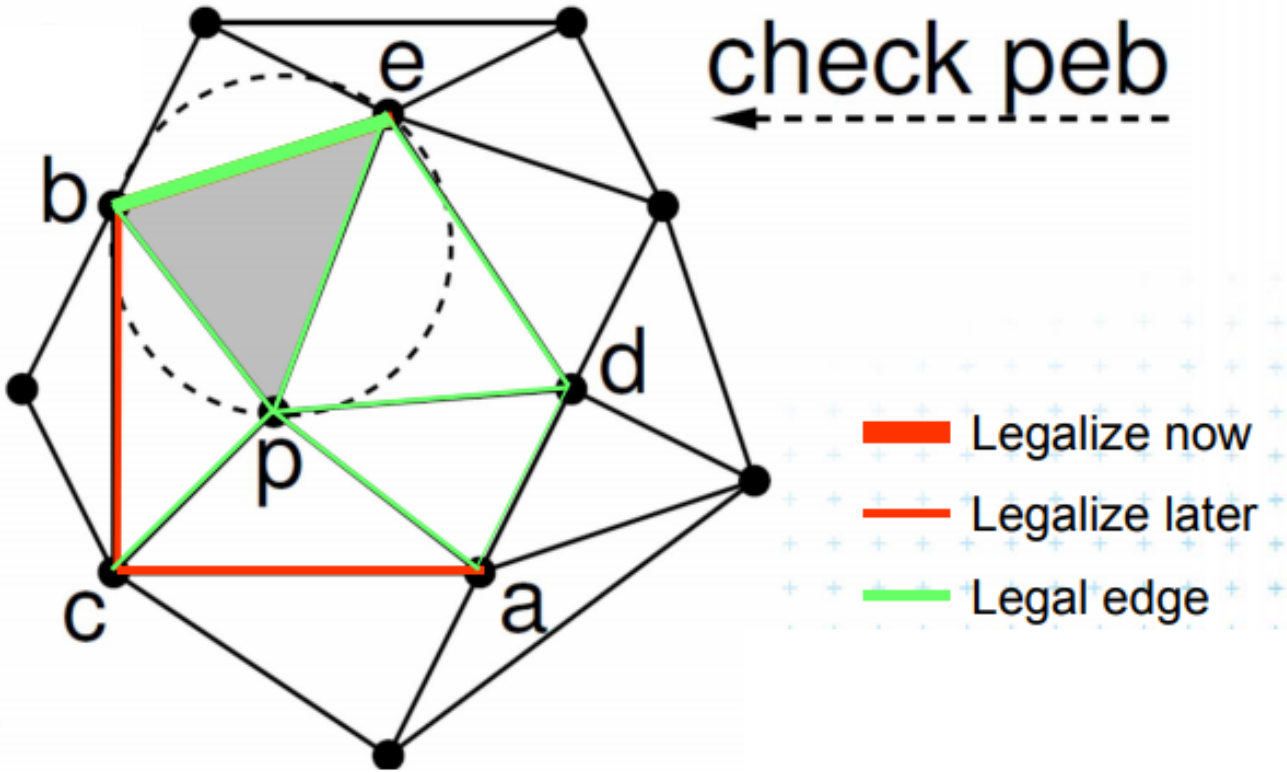
Step 3



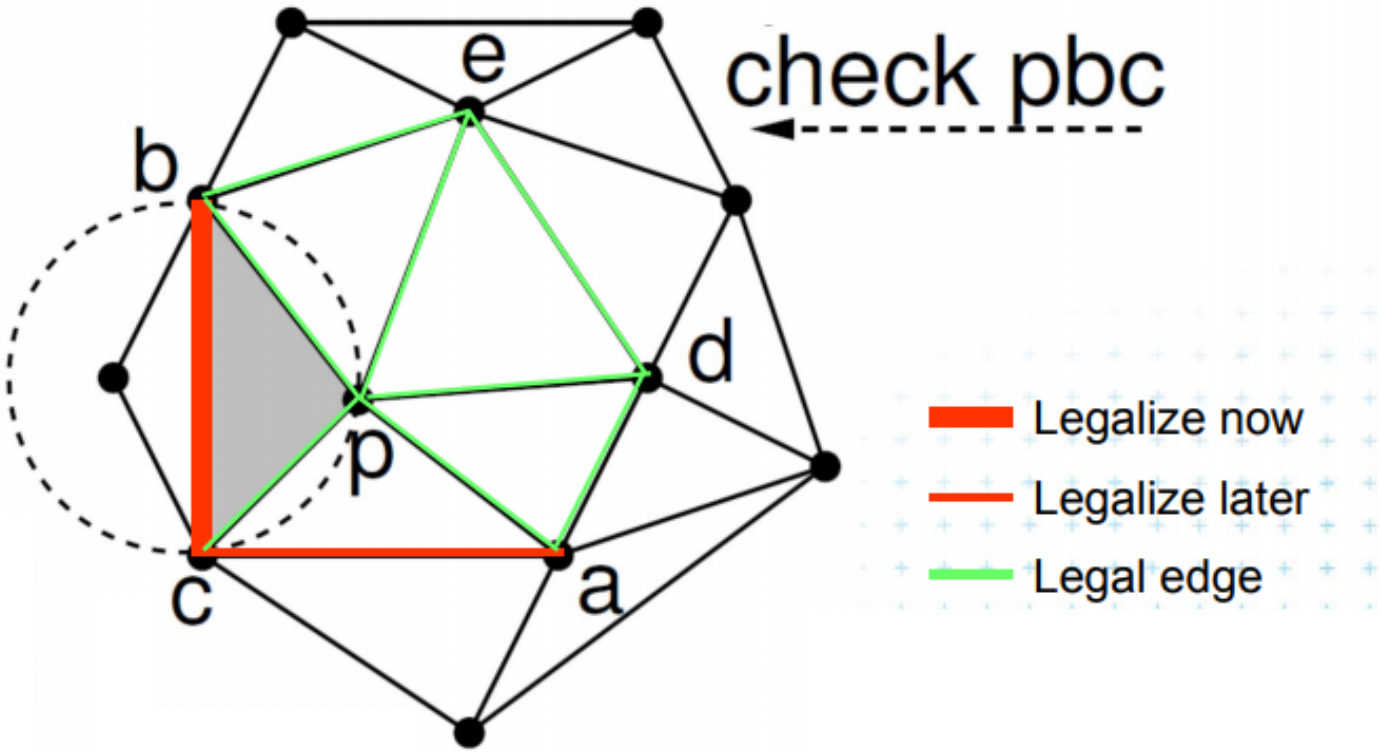
Step 4


flip(db)
check pde
e
b
d
p
c
a
Legalize now
Legalize later
Legal edge

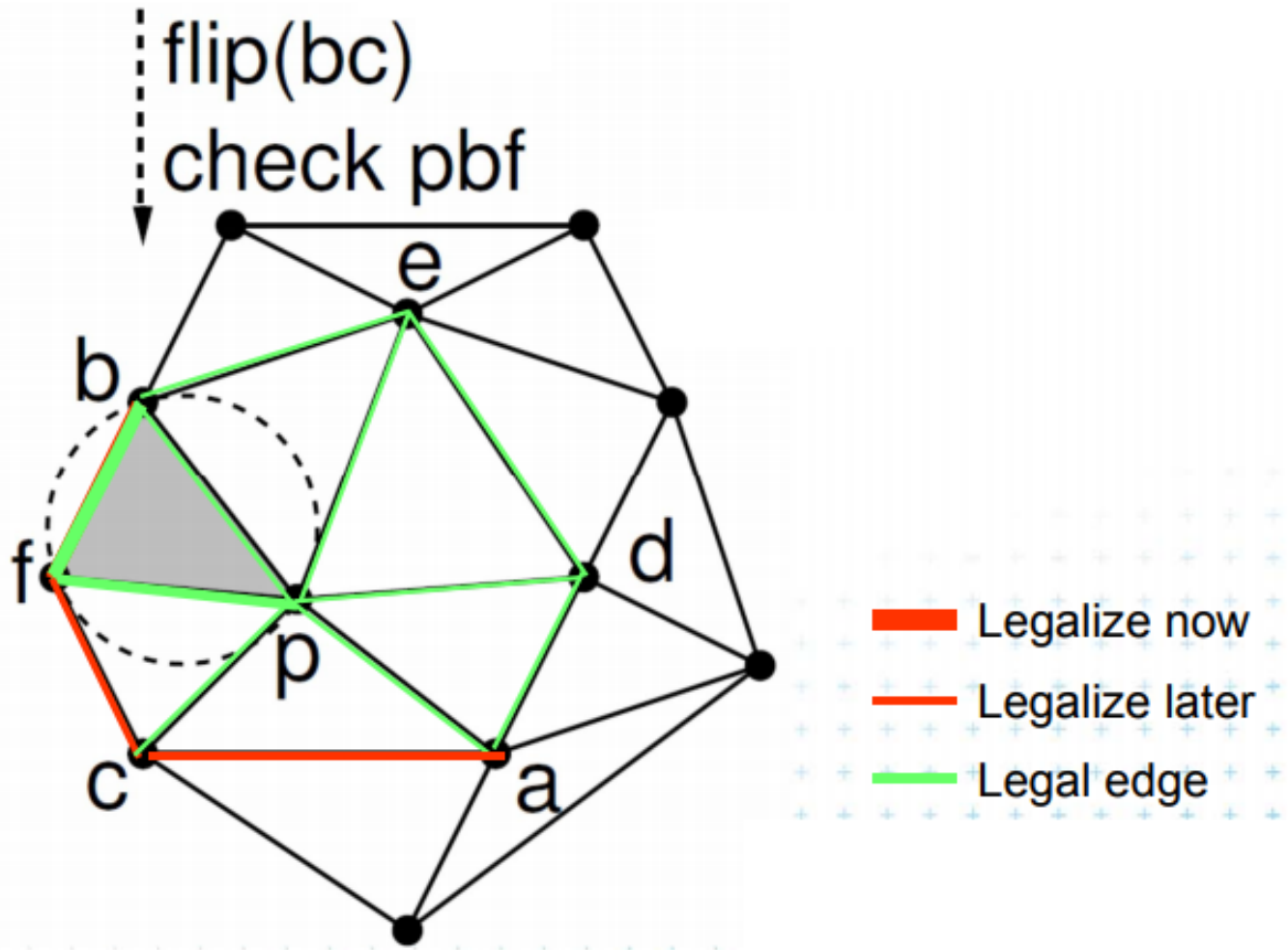
Step 5



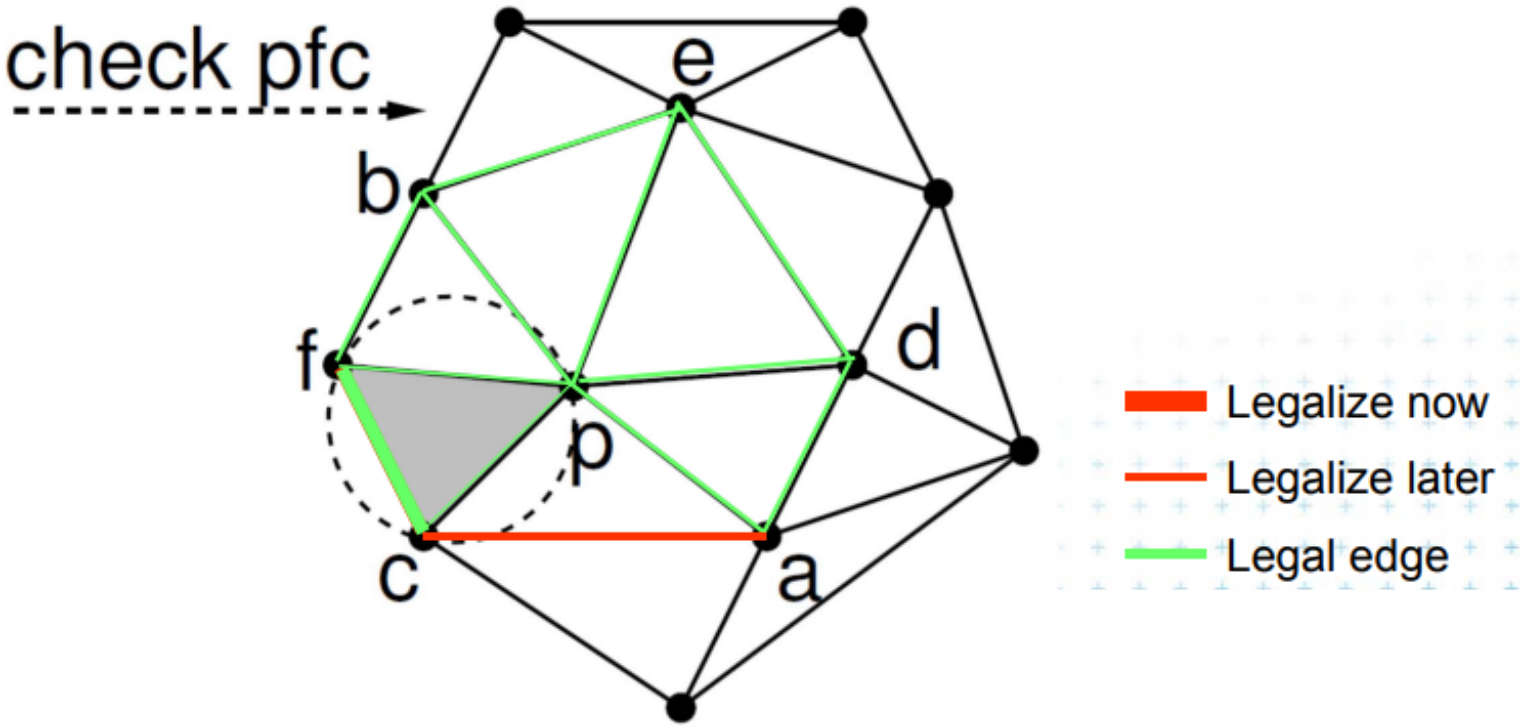
Step 6



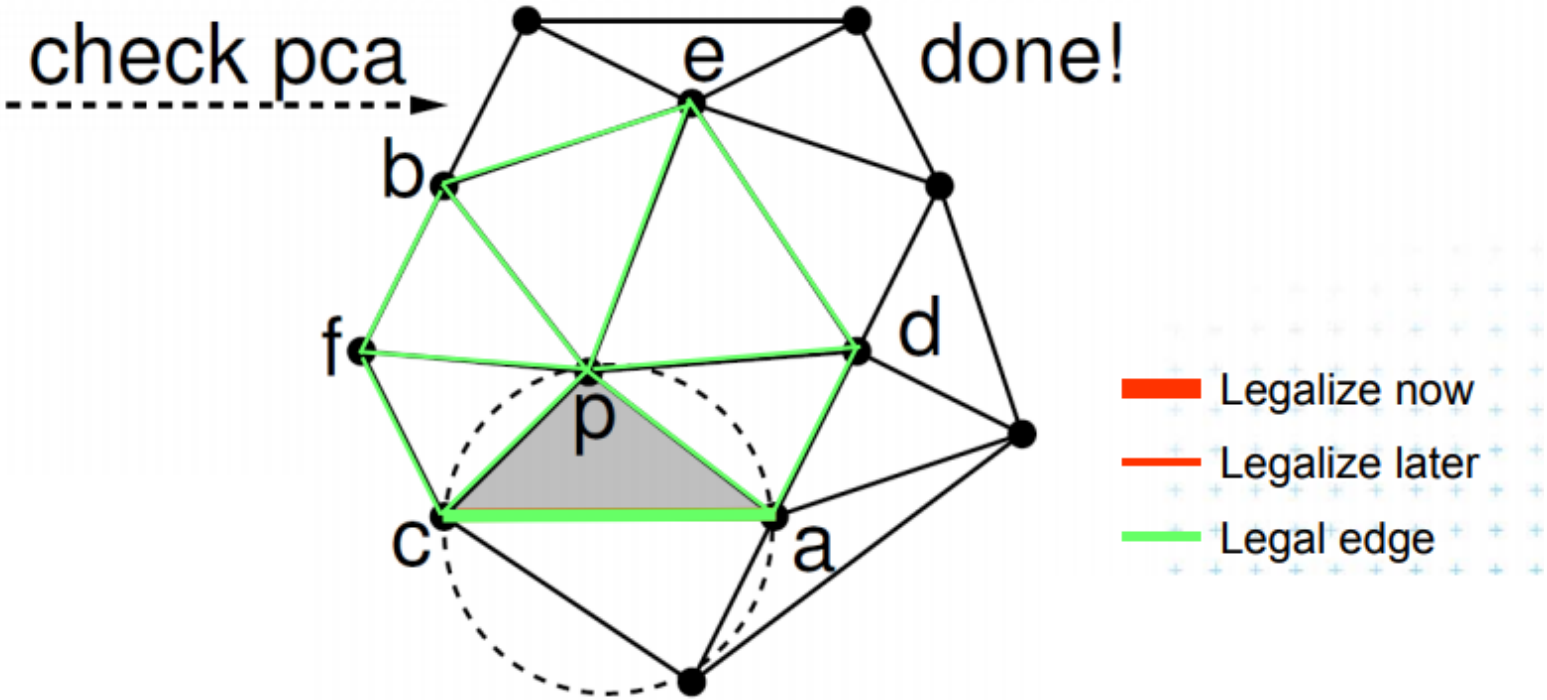
Step 7


flip(bc)
check pbf
e
b
f
d
p
c
a
Legalize now
Legalize later
Legal edge

Step 8



Step 9


check pca
e
done!
b
f
d
p
c
a
Legalize now
Legalize later
Legal edge

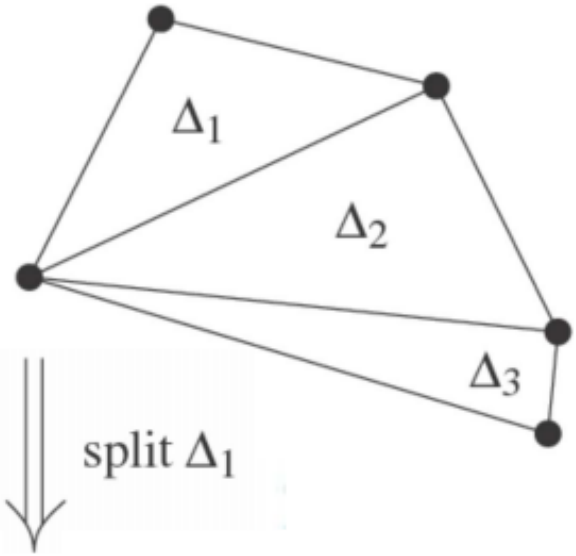
Point Location

Data Structures

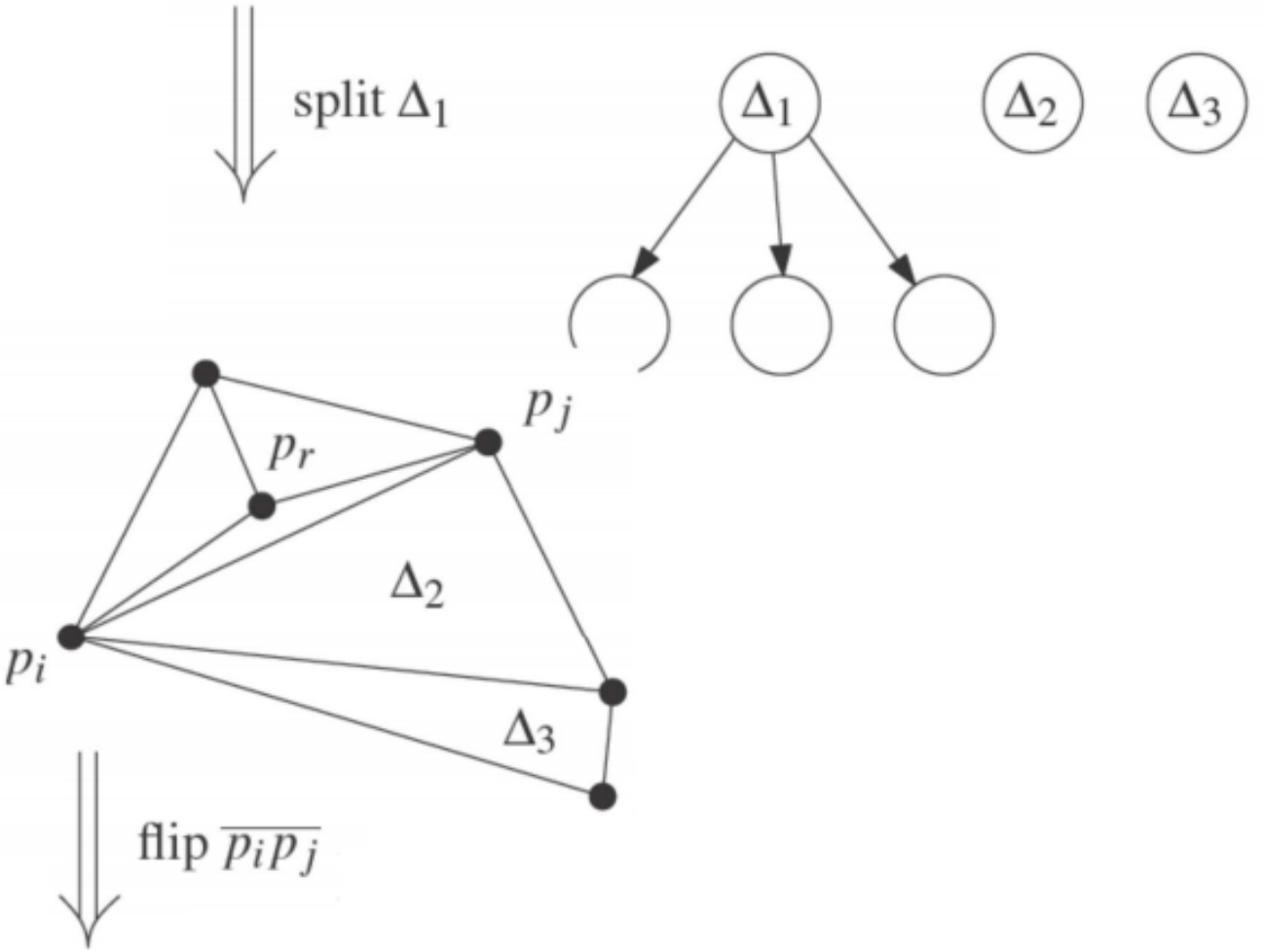
1. Standard: Build a history graph as in trapezoidal decomposition for ordinary point location.
2. Easier approach: Instead of building a history graph, we maintain a set of buckets.
   - For each triangle, the bucket – contains the set of points inside it that are not inserted yet.
   - Whenever edge flip is done, we do rebucketing.

History graph

Simplified
- it should also contain the root node

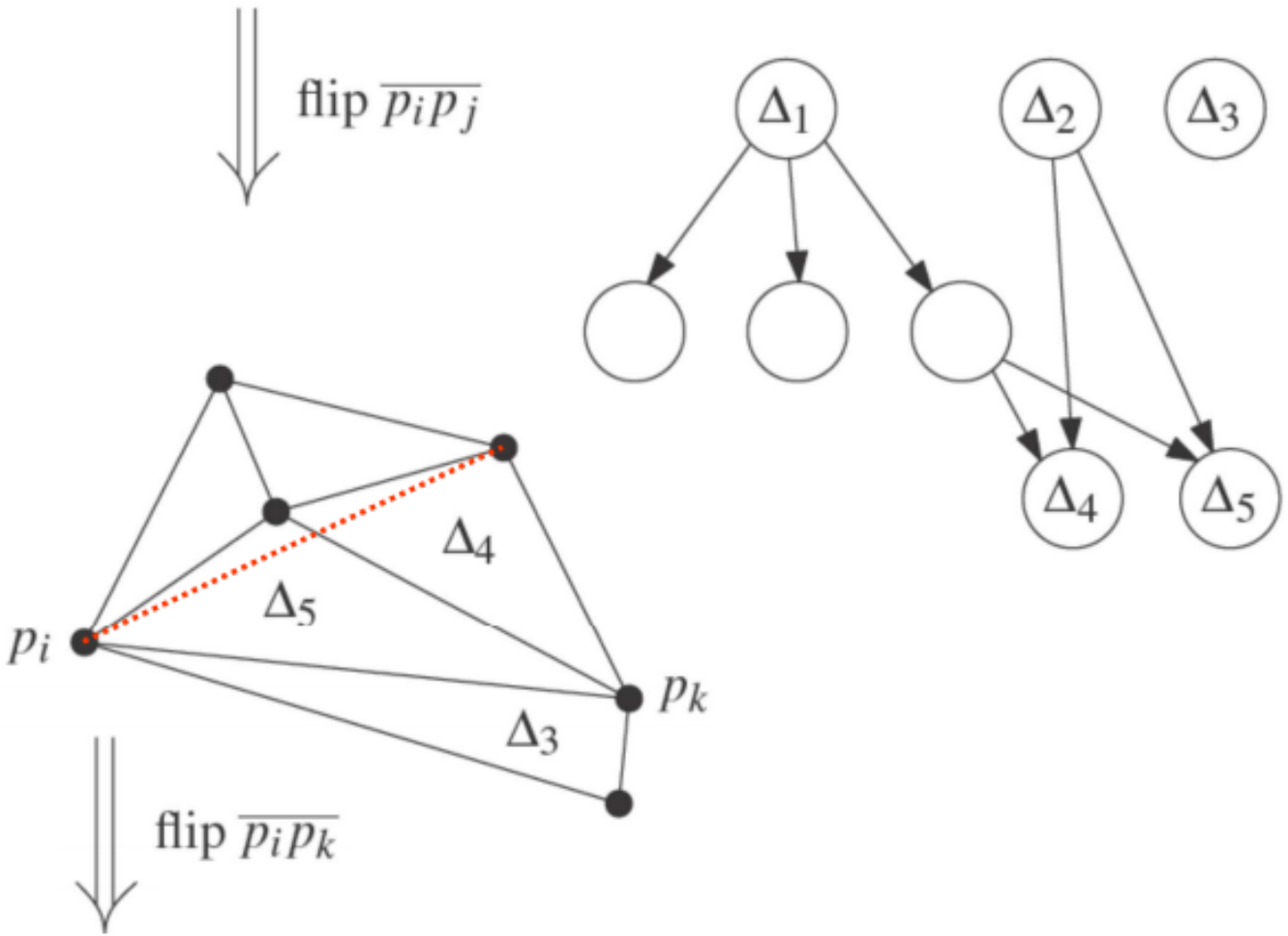


History graph



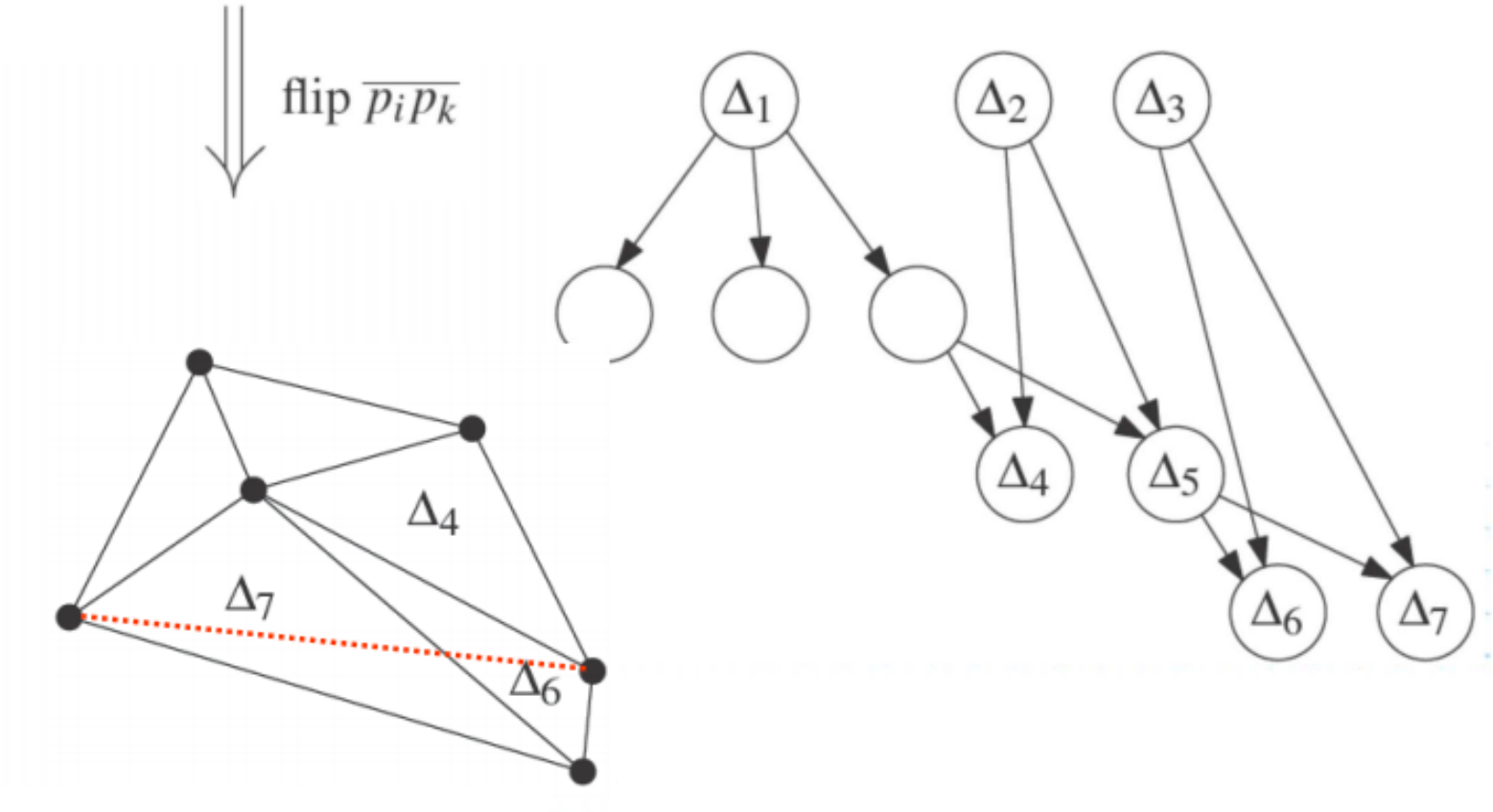
The triangle 1 is split into 3 new triagles by the time the new vertex is inserted. That's why we create 3 new vertices. The procedure does not stop here: some other triangles will also be modified as well.

History graph



The triangle 2 and the triangle on the right on the lowest level of our graph will be deleted and two new triangles will occur.

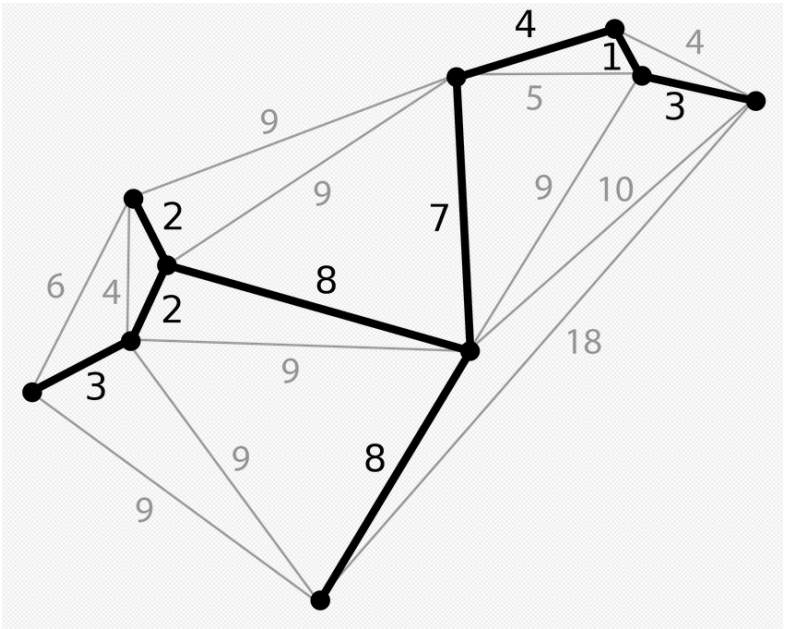
History graph



The same procedure will take place until we achieve a Delaunay triangulation. The last instance of our graph declares the triangles that are left.

Minimum Spanning Tree

- A minimum spanning tree of n points is a tree such that the total weight of edges is minimized.
- Weight of an edge = length of edge



The minimum spanning tree is a sub-graph of the Delaunay triangulation!

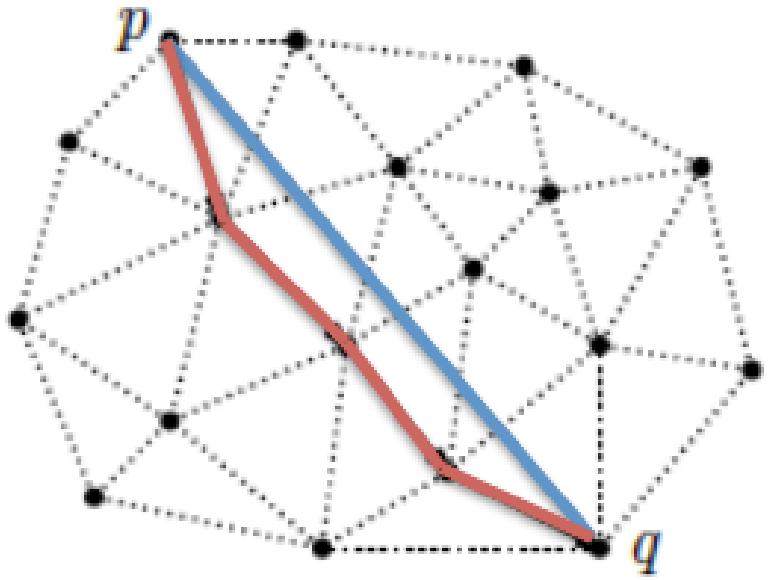
Minimum Spanning Tree

Computing MST

1. **In simple way**
   - Consider all possible edges $E = O(n^2)$
   - Apply Kruskal's algorithm to E
   - Total time $O(Elogn) = O(n^2 logn)$
2. **Using Delaunay triangulation**
   - Compute Delaunay triangulation in $O(nlogn)$ time
   - Apply Kruskal's algorithm to the Delaunay triangulation
   - Total time $O(nlogn)$

Spanner Properties

The length of the shortest path between two points on a planar Delaunay triangulation is **not** significantly longer than the straight-line distance between these points.

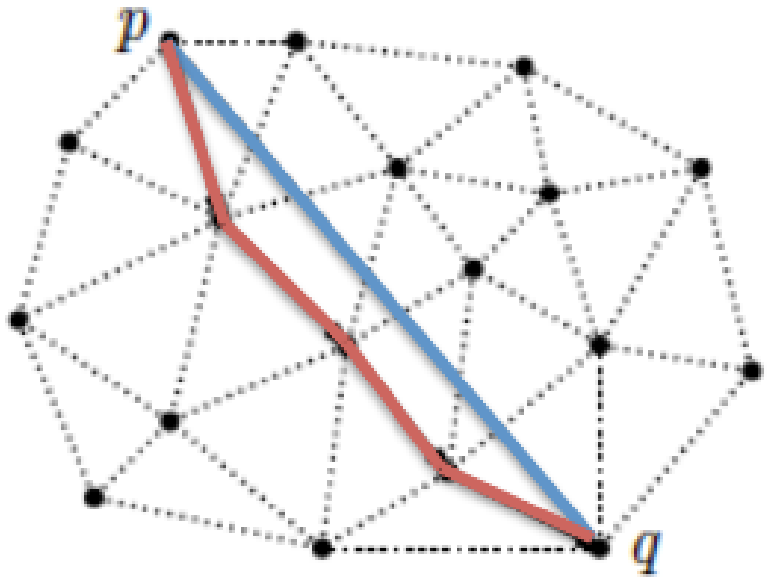


Spanner Properties

More Formally Given a set of points P and two points p and q. We say that a straight-line graph G is a t- spanner if

$$\delta_G(p, q) \leq t\|p * q\|$$

where $\delta_G(p, q)$ is the shortest path between p and q in graph G.



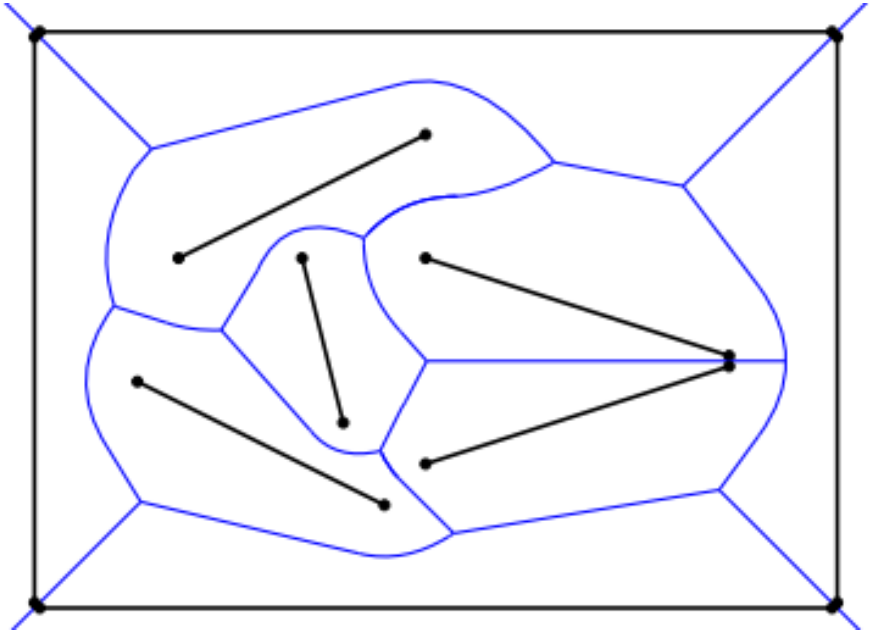
Path Plannning

Voronoi for Line Segments

The Voronoi diagram can also be defined for objects other than points. The distance from a point in the plane to an object is then measured to the closest point on the object.

Whereas the bisector of two points is simply a line, the bisector of two disjoint line segments has a more complex shape.

Voronoi for Line Segments

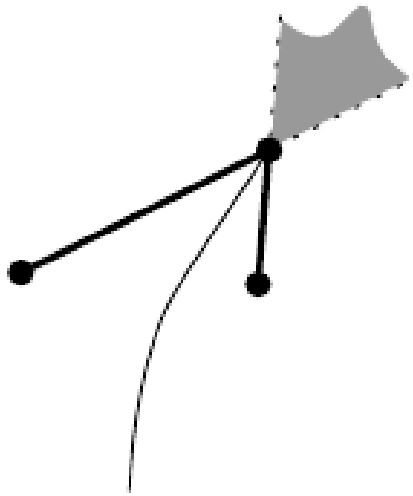


Observation

Parabolic arcs occur if the closest point of one line segment is an endpoint and the closest point of the other line segment is in its interior. In all other cases the bisector part is straight.

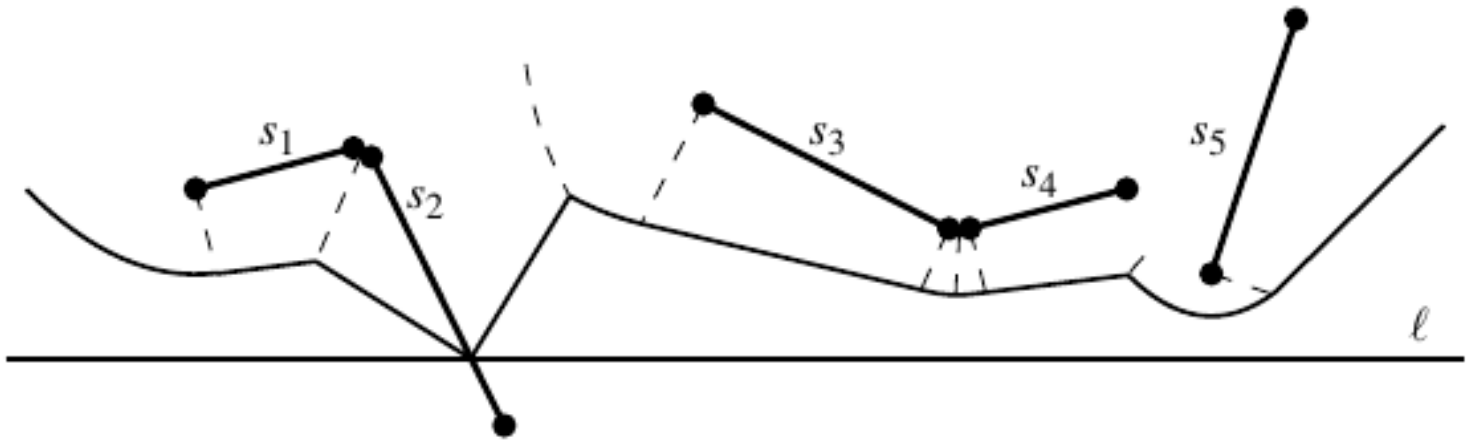
Voronoi for Line Segments

To avoid the complications that arise in defining and computing Voronoi diagrams of line segments that share endpoints, we will simply assume here that all line segments are strictly disjoint. In many applications we can simply shorten the line segments very slightly to obtain disjoint line segments.



Voronoi for Line Segments

The sweep line algorithm for points can be adapted to the case of line segment sites. Let $S = \{s_1, ..., s_n\}$ be a set of n disjoint line segments. We call the segments of S sites as before, and use the terms site endpoint and site interior in the following description.



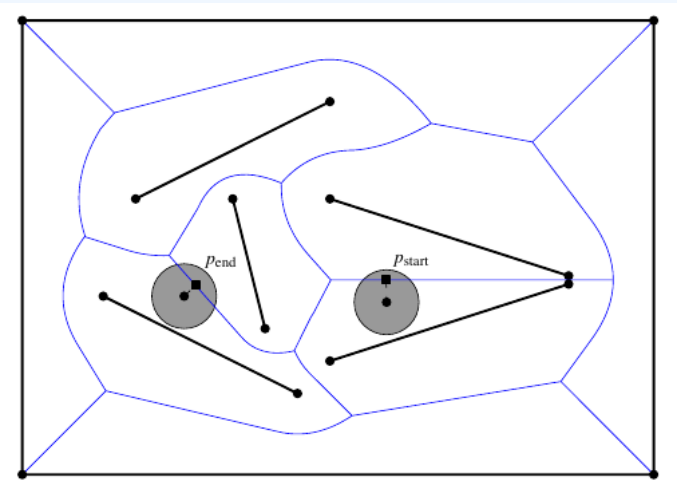
Complexity

Lemma. The Voronoi diagram of a set of n disjoint line segment sites can be computed in $O(n \log n)$ time using $O(n)$ storage.

Voronoi for Line Segments

Motion planning

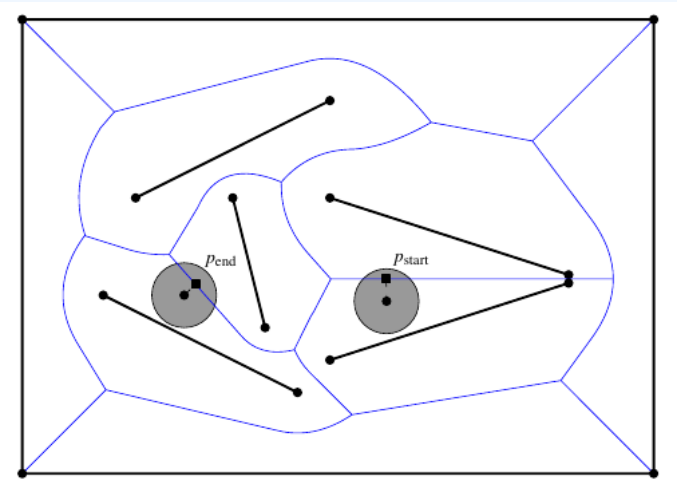
Assume that a set of obstacles is given, consisting of n line segments in total, and that we have a robot $\mathcal{R}$. We assume that the robot can move freely in all directions, and is approximated well by an enclosing disc D. Suppose that we wish to find a collision-free motion from one location of the robot to another, or to decide that none exists.



Voronoi for Line Segments

Motion planning

One motion-planning technique is called **retraction**. The idea of retraction is that the arcs of the Voronoi diagram define the middle between the line segments, and therefore define a path with the most clearance. So a path over the arcs of the Voronoi diagram is the best option for a collision-free path.



Voronoi for Line Segments

Algorithm RETRACTION$(S, q_{\text{start}}, q_{\text{end}}, r)$

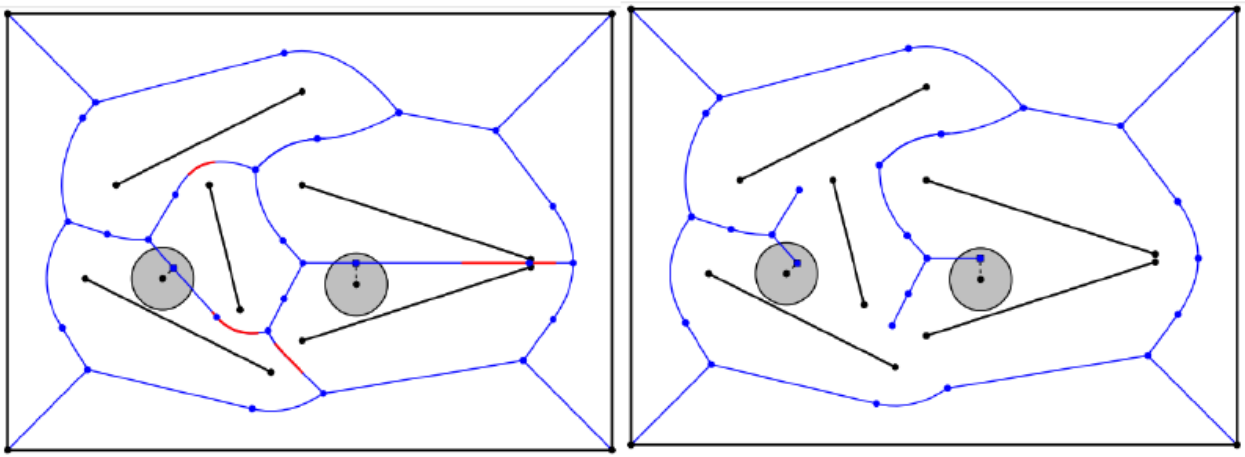
Input. A set $S := \{s_1, \ldots, s_n\}$ of disjoint line segments in the plane, and two discs D_{start} and D_{end} centered at q_{start} and q_{end} with radius r. The two disc positions do not intersect any line segment of S.

Output. A path that connects q_{start} to q_{end} such that no disc of radius r with its center on the path intersects any line segment of S. If no such path exists, this is reported.

1. Compute the Voronoi diagram $\text{Vor}(S)$ of S inside a sufficiently large bounding box.
2. Locate the cells of $\text{Vor}(P)$ that contain q_{start} and q_{end}.
3. Determine the point p_{start} on $\text{Vor}(S)$ by moving q_{start} away from the nearest line segment in S. Similarly, determine the point p_{end} on $\text{Vor}(S)$ by moving q_{end} away from the nearest line segment in S. Add p_{start} and p_{end} as vertices to $\text{Vor}(S)$, splitting the arcs on which they lie into two.
4. Let $\mathcal{G}$ be the graph corresponding to the vertices and edges of the Voronoi diagram. Remove all edges from $\mathcal{G}$ for which the smallest distance to the nearest sites is smaller than or equal to r.
5. Determine with depth-first search whether a path exists from p_{start} to p_{end} in $\mathcal{G}$. If so, report the line segment from q_{start} to p_{start}, the path in $\mathcal{G}$ from p_{start} to p_{end}, and the line segment from p_{end} to q_{end} as the path. Otherwise, report that no path exists.

Figure: Retraction pseudocode

Voronoi for Line Segments



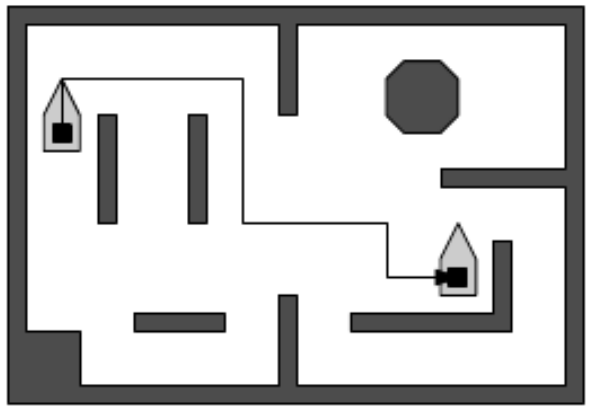
Voronoi for Line Segments

- **What if we are looking for the shortest path for our robot?**
  - **What if the robot itself will also be polygonal?**
    - **What if the obstacles are polygonal?**

Another Approach

Motion planning

The most drastic simplification is that we will look at a 2-dimensional motion planning problem. The environment will be a planar region with polygonal obstacles, and the robot itself will also be polygonal. We also assume that the environment is static and known to the robot.



Work space and Configuration space

Let $\mathcal{R}$ be a robot moving around in a 2-dimensional environment, or work space, consisting of a set S = $\mathcal{P}_1, ..., \mathcal{P}_t$ of obstacles.

A *placement*, or *configuration*, of the robot can now be specified by a translation vector.

- We denote the robot $\mathcal{R}$ translated over a vector (x,y) by $\mathcal{R}$(x,y).

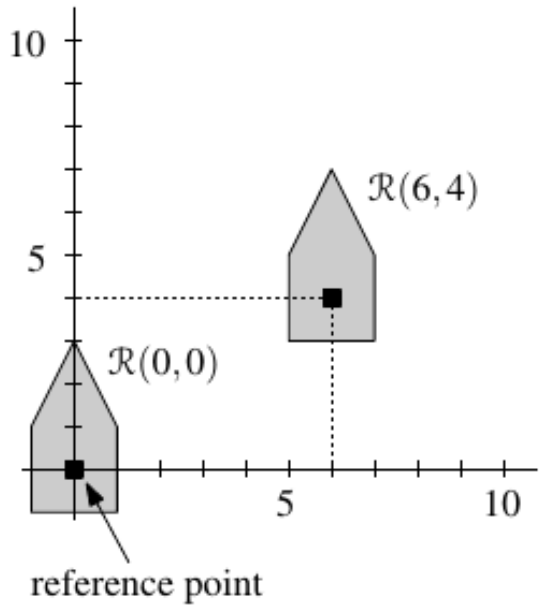
An alternative way to view this is in terms of a *reference point*. This is most intuitive if the origin (0,0) lies in the interior of $\mathcal{R}$(0,0).

Work space and Configuration space

Reference Point

We can specify a placement of $\mathcal{R}$ by simply stating the coordinates of the reference point if the robot is in the given placement.

- Thus $\mathcal{R}$(x,y) specifies that the robot is placed with its reference point at (x,y).
- By definition, this point is at the origin for $\mathcal{R}(0,0)$.



Work space and Configuration space

In general, a placement of a robot is specified by a number of parameters that corresponds to the number of *degrees of freedom* (DOF) of the robot.

- This number is two for planar robots that can only translate.
- It is three for planar robots that can rotate as well as translate.

The parameter space of a robot R is usually called its *configuration space*. It is denoted by $\mathcal{C}(\mathcal{R})$. A point p in this configuration space corresponds to a certain placement $\mathcal{R}$(p) of the robot in the work space.

Work space and Configuration space

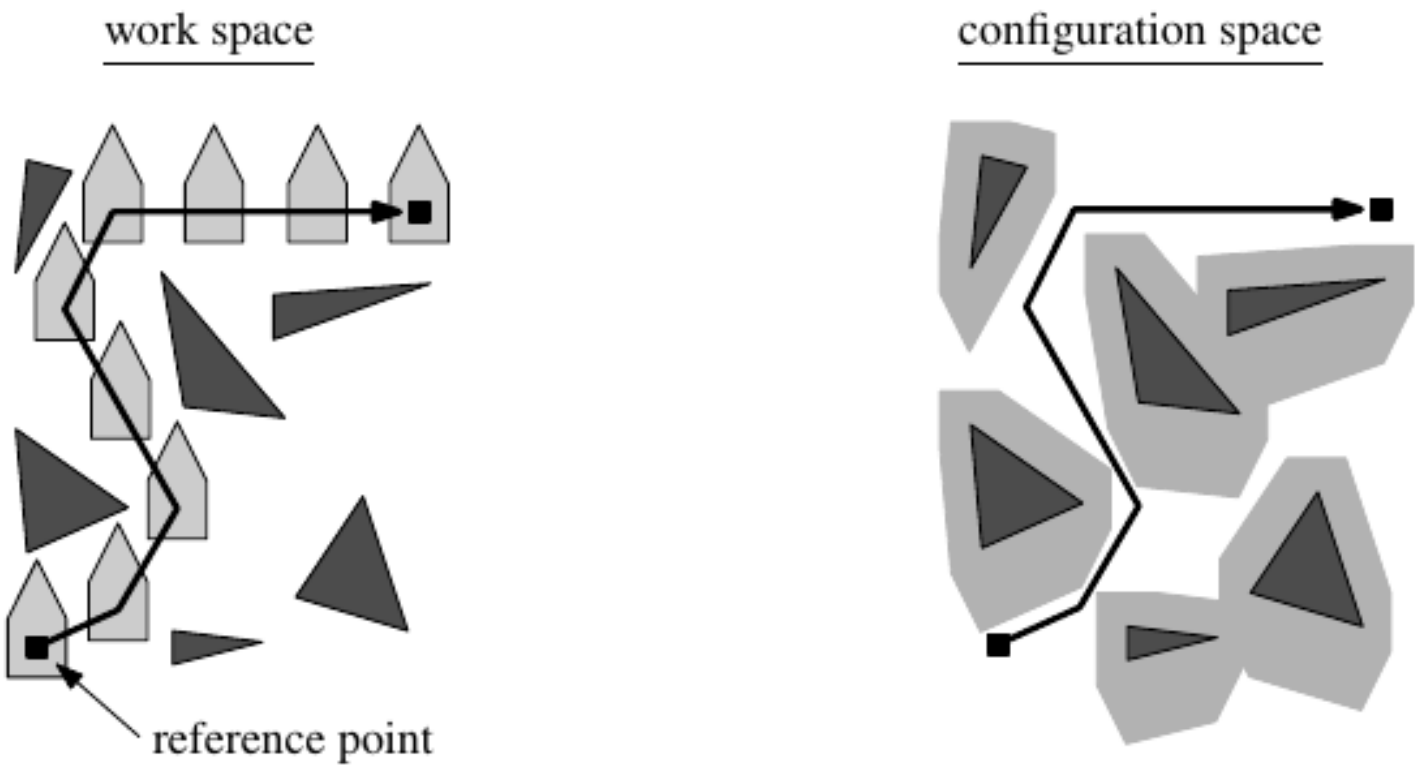


Figure: A path in the work space and the corresponding curve in the configuration space

Work space and Configuration space

t is useful to distinguish the two notions: the work space is the space where the robot actually moves around—the real world, so to speak—and the configuration space is the parameter space of the robot.

We now have a way to specify a placement of the robot, namely by specifying values for the parameters determining the placement or, in other words, by specifying a point in configuration space.

Work space and Configuration space

Clearly not all points in configuration space are possible; points corresponding to placements where the robot intersects one of the obstacles in S are forbidden.

We call the part of the configuration space consisting of these points the forbidden configuration space, or forbidden space for short. It is denoted by $\mathcal{C}_{forb}(\mathcal{R}, S)$.

The rest of the configuration space, which consists of the points corresponding to free placements—placements where the robot does not intersect any obstacle—is called the free configuration space, or free space, and it is denoted by $\mathcal{C}_{free}(\mathcal{R}, S)$.

Work space and Configuration space

We have seen how to map placements of the robot to points in the configuration space, and paths of the robot to curves in that space. Can we also map obstacles to configuration space?

The answer is yes: an obstacle P is mapped to the set of points p in configuration space such that $\mathcal{R}$(p) intersects $\mathcal{P}$.

The resulting set is called the configuration-space obstacle, or $\mathcal{C}$-obstacle for short, of $\mathcal{P}$.

A point robot

For a point robot, the work space and the configuration space are identical.

Rather than finding a path from a particular start position to a particular goal position we will construct a data structure storing a representation of the free space. This data structure can then be used to compute a path between any two given start and goal positions.

To simplify the description we restrict the motion of the robot to a large bounding box B that contains the set of polygons.

Free Space

Our goal is to compute a representation of the free space that allows us to find a path for any start and goal position.

- We will use the **trapezoidal map*** for this.

Algorithm COMPUTEFREESPACE(S)
Input. A set S of disjoint polygons.
Output. A trapezoidal map of $\mathcal{C}_{\text{free}}(\mathcal{R}, S)$ for a point robot $\mathcal{R}$.
1. Let E be the set of edges of the polygons in S.
2. Compute the trapezoidal map $\mathcal{T}(E)$ with algorithm TRAPEZOIDALMAP described in Chapter 6.
3. Remove the trapezoids that lie inside one of the polygons from $\mathcal{T}(E)$ and return the resulting subdivision.

Algorithm ComputeFreeSpace

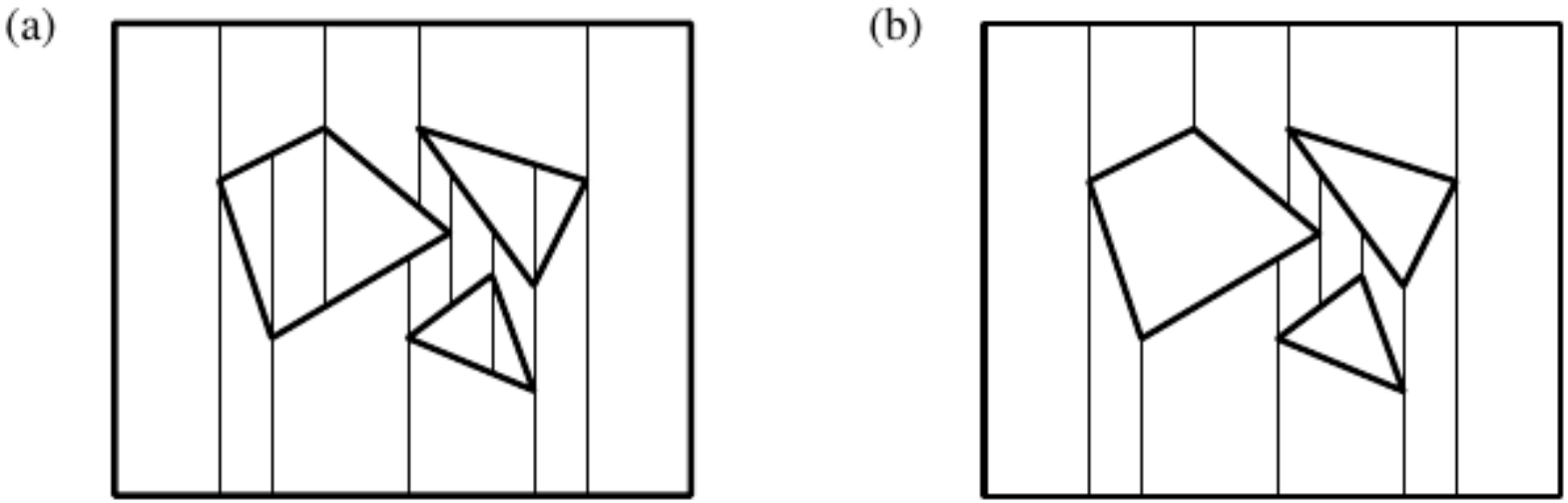


Figure: The algorithm is illustrated in this figure. Part (a) of the figure shows the trapezoidal map of the obstacle edges inside the bounding box; this is what is computed in line 2 of the algorithm. Part (b) shows the map after the trapezoids inside the obstacles have been removed in line 3.

Algorithm ComputeFreeSpace

Complexity

Lemma. A trapezoidal map of the free configuration space for a point robot moving among a set of disjoint polygonal obstacles with n edges in total can be computed by a randomized algorithm in $O(n \log n)$ expected time.

Intuitive approach of trapezoidal map

- **Goal:** preprocess an arrangement of segments, so that we can answer the following query fast:
  find in which face of the arrangement a given input point belongs to.
- **Solution:** trapezoidal map via randomized incremental construction

trapezoidal map - Point location data structure

- A directed acyclic graph (DAG)
- leaf: node with zero out-degree (leafs are in 1–1 correspondence with trapezoids, using pointers)
- internal nodes (non-zero out-degree) are of two types:

  - x-node: points to a segment endpoint vertex
    - left subtree: points left of vertex
    - right subtree: points right of vertex
  - y-node: points to a segment
    - left subtree: points above segment
    - right subtree: points below segment (or vice versa)

trapezoidal map - Example

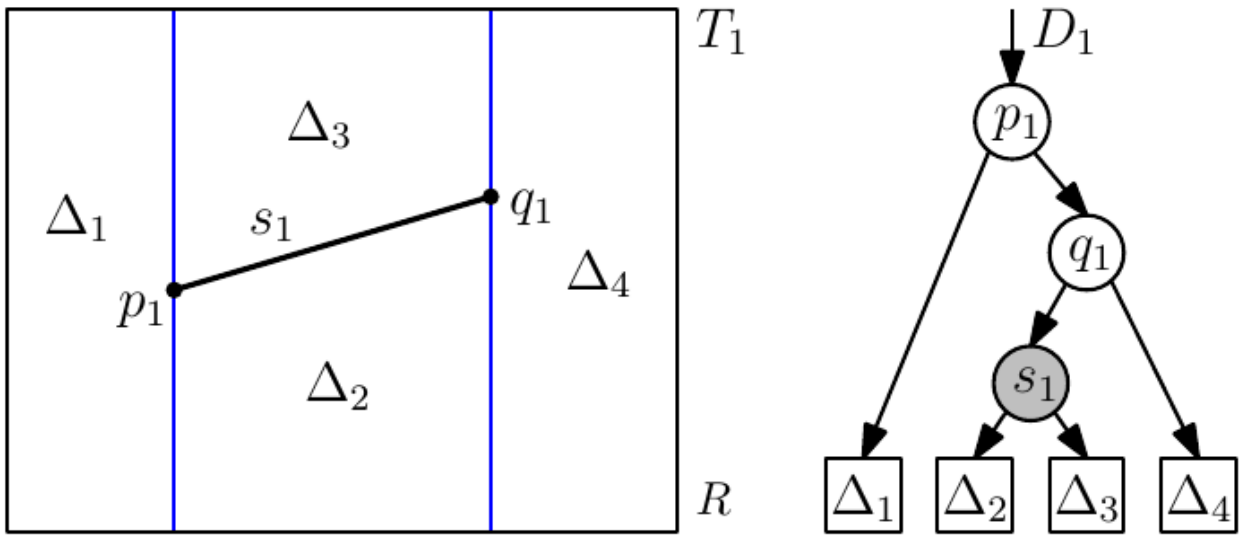


Figure: Initial state

trapezoidal map - Example

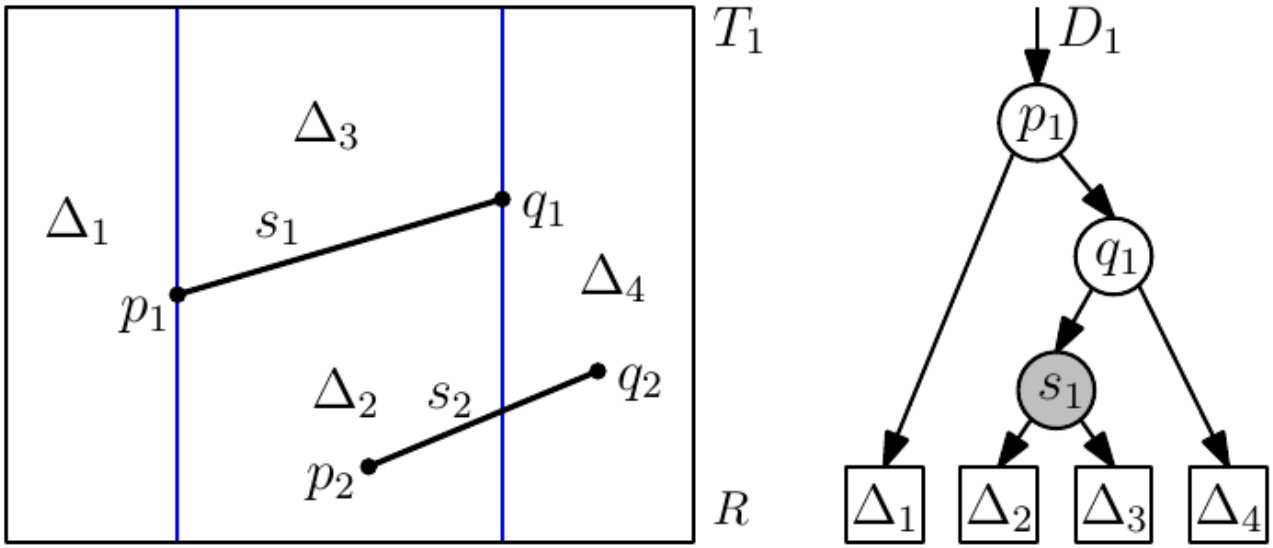


Figure: Insert line segment S_2

trapezoidal map - Example

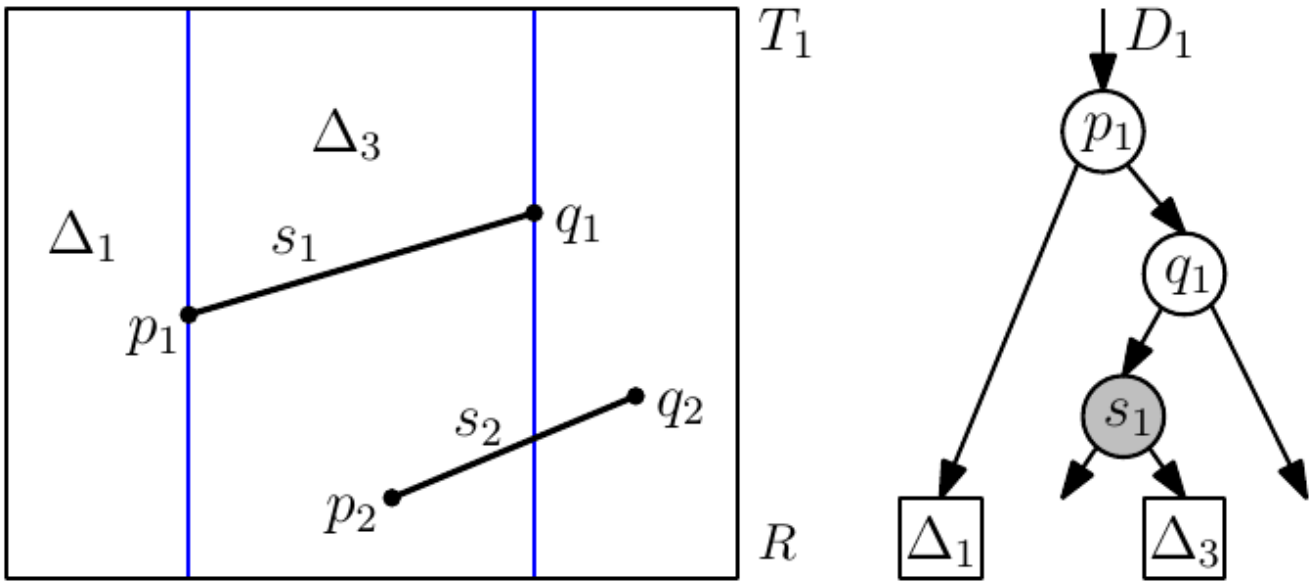


Figure: Remove old Trapezoidals Δ

trapezoidal map - Example

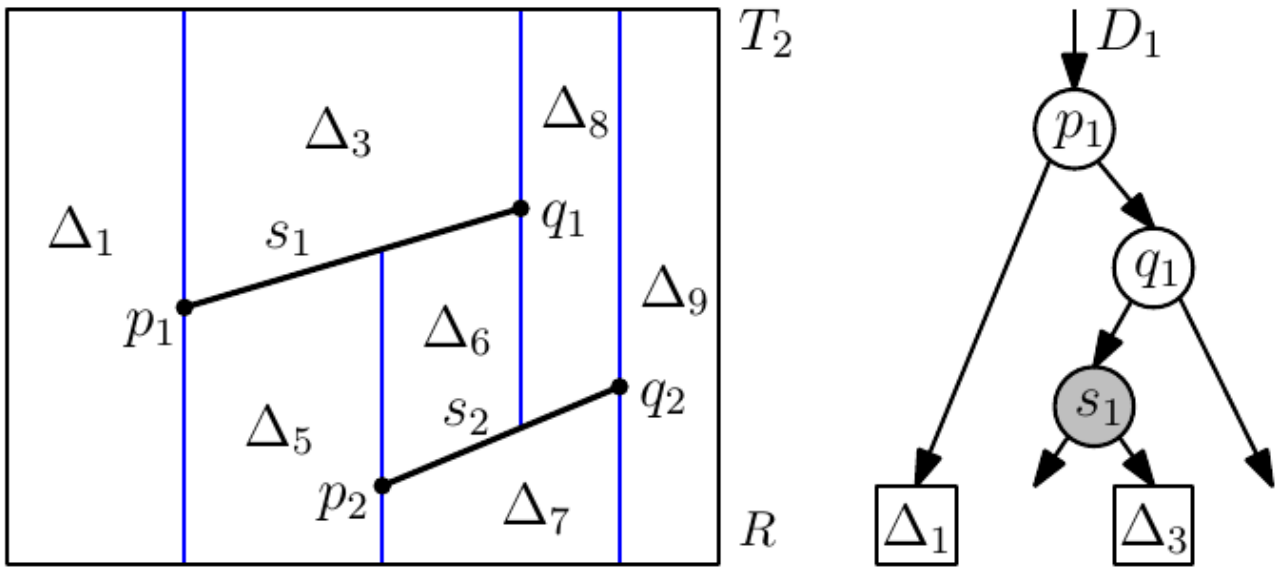


Figure: update DAG-1

trapezoidal map - Example

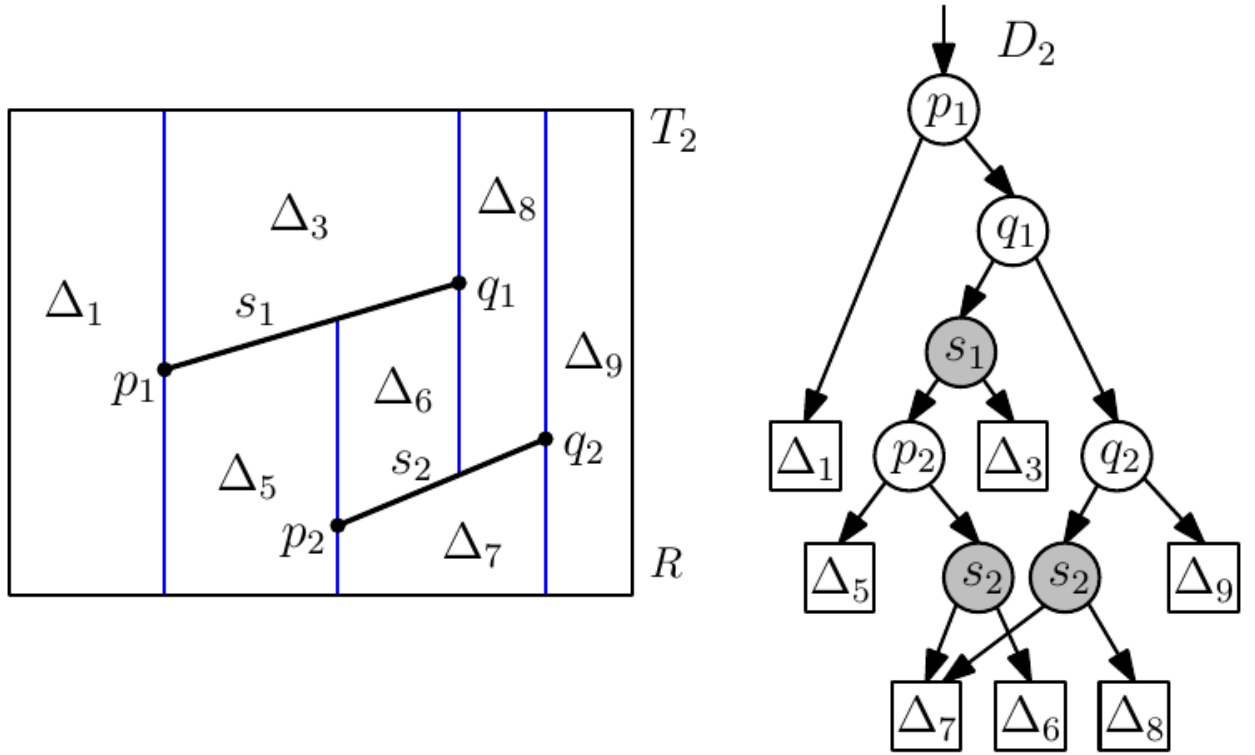


Figure: update DAG-2

Algorithm ComputeFreeSpace

How to

How do we use $\mathcal{T}(\mathcal{C}_{free})$ to find a path from a start position p_{start} to a goal position p_{goal}?

- If p_{start} and p_{goal} are in the same trapezoid of the map, this is easy: the robot can simply move to its goal in a straight line.
- If the start and goal position are in different trapezoids,to guide the motion across trapezoids we construct a *road map*, $\mathcal{G}_{road}$, through the free space.

Except for an initial and final portion, paths will always follow the road map.

Road Map

How to

We place one node in the center of each trapezoid, and we place one node in the middle of each vertical extension.There is an arc between two nodes if and only if one node is in the center of a trapezoid and the other node is on the boundary of that same trapezoid.

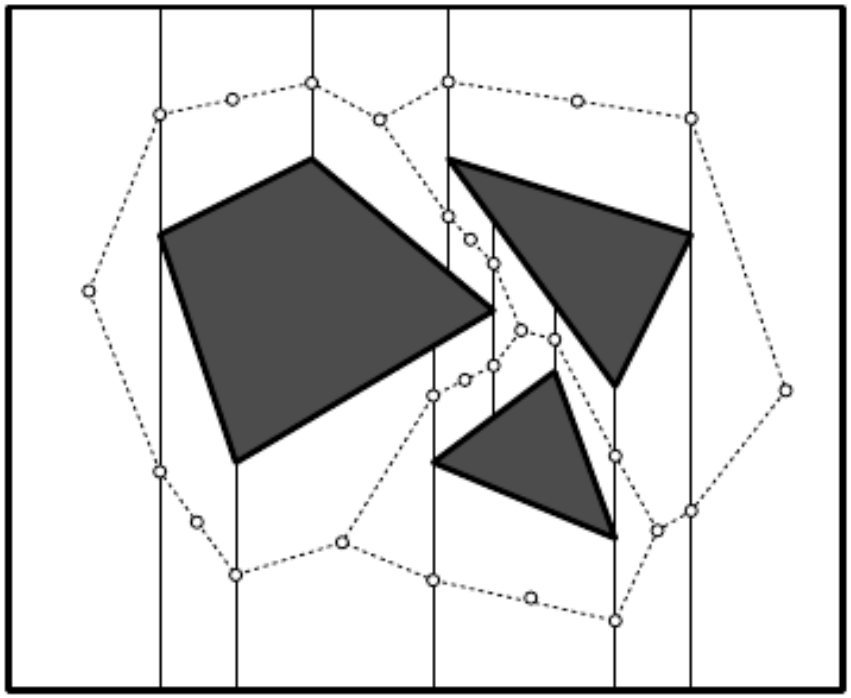


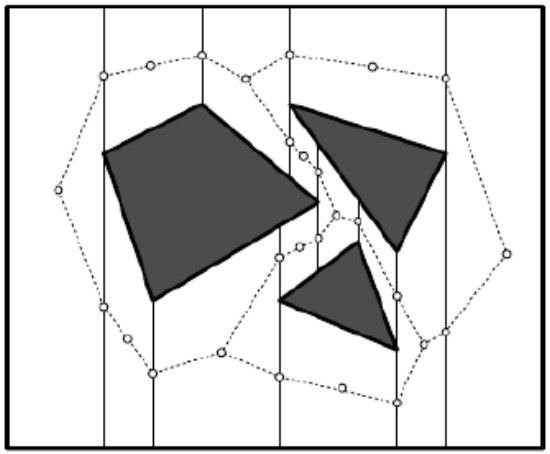
Figure: A road map

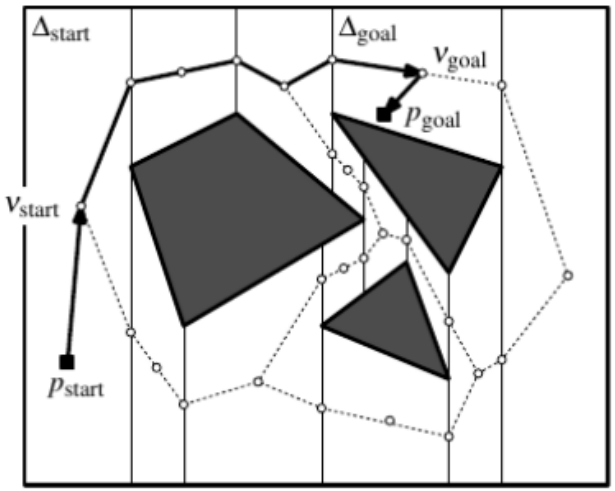
Algorithm

Algorithm COMPUTEPATH($\mathcal{T}(\mathcal{C}_{\text{free}}), \mathcal{G}_{\text{road}}, p_{\text{start}}, p_{\text{goal}}$)
Input. The trapezoidal map $\mathcal{T}(\mathcal{C}_{\text{free}})$ of the free space, the road map $\mathcal{G}_{\text{road}}$, a start position p_{start}, and goal position p_{goal}.
Output. A path from p_{start} to p_{goal} if it exists. If a path does not exist, this fact is reported.

1. Find the trapezoid Δ_{start} containing p_{start} and the trapezoid Δ_{goal} containing p_{goal}.
2. **if** Δ_{start} or Δ_{goal} does not exist
3. **then** Report that the start or goal position is in the forbidden space.
4. **else** Let v_{start} be the node of $\mathcal{G}_{\text{road}}$ in the center of Δ_{start}.
5. Let v_{goal} be the node of $\mathcal{G}_{\text{road}}$ in the center of Δ_{goal}.
6. Compute a path in $\mathcal{G}_{\text{road}}$ from v_{start} to v_{goal} using breadth-first search.
7. **if** there is no such path
8. **then** Report that there is no path from p_{start} to p_{goal}.
9. **else** Report the path consisting of a straight-line motion from p_{start} to v_{start}, the path found in $\mathcal{G}_{\text{road}}$, and a straight-line motion from v_{goal} to p_{goal}.

Voronoi for Line Segments





Algorithm

Complexity

Let $\mathcal{R}$ be a point robot moving among a set S of polygonal obstacles with n edges in total. We can preprocess S in O($nlogn$) expected time, such that between any start and goal position a collision-free path for $\mathcal{R}$ can be computed in O(n) time, if it exists.

Minkowski sums

The same approach can be used if the robot is a polygon.

There is one difference that makes dealing with a polygonal robot more difficult: the configuration-space obstacles are no longer the same as the obstacles in work space.

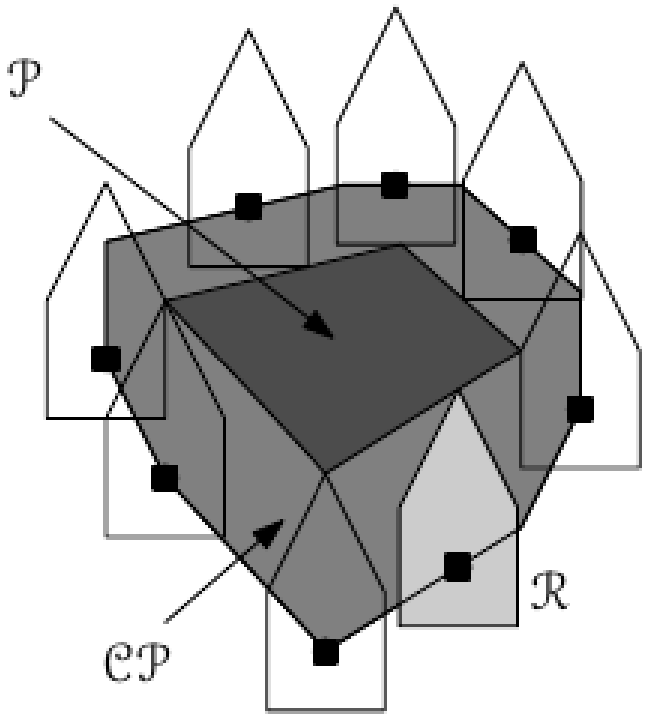
Obstacle

If we denote the $\mathcal{C}$-obstacle of P by $\mathcal{CP}$, then we have

$$\mathcal{CP} := \{(x, y) : \mathcal{R}(x, y) \cap P \neq 0\}.$$

Minkowski sums

We can visualize the shape of $\mathcal{CP}$ by sliding $\mathcal{R}$ along the boundary of $\mathcal{P}$; the curve traced by the reference point of $\mathcal{R}$ is the boundary of $\mathcal{CP}$.



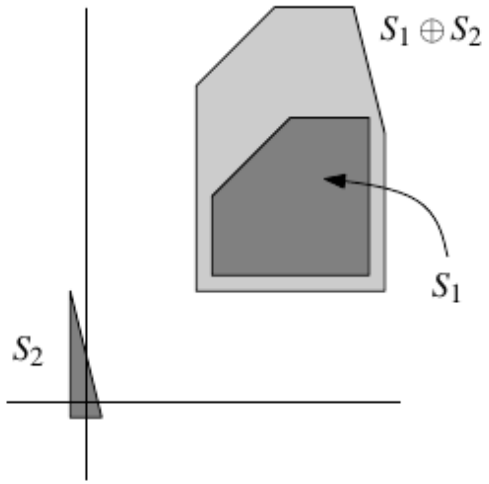
Minkowski sums

We can describe this in a different way using the notion of Minkowski sums. The Minkowski sum of two sets $S_1 \subset \mathbb{R}^2$ and $S_2 \subset \mathbb{R}^2$, denoted by $S_1 \oplus S_2$, is defined as

$$S_1 \oplus S_2 := \{p + q : p \in S_1, q \in S_2\},$$

where $p + q$ denotes the vector sum of the vectors p and q, that is, if $p = (p_x, p_y)$ and $q = (q_x, q_y)$ then we have

$$p + q := (p_x + q_x, p_y + q_y).$$



Minkowski sums

- For a point $p = (p_x, p_y)$ we define $-p := (-p_x, -p_y)$, and for a set S we define $-S := \{-p : p \in S\}$

Theorem. Let $\mathcal{R}$ be a planar, translating robot and let $\mathcal{P}$ be an obstacle. Then the $\mathcal{C}$-obstacle of $\mathcal{P}$ is $\mathcal{P} \oplus (-\mathcal{R}(0,0))$.

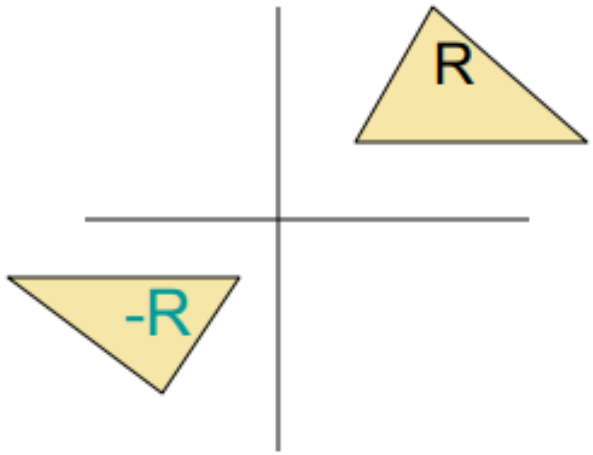
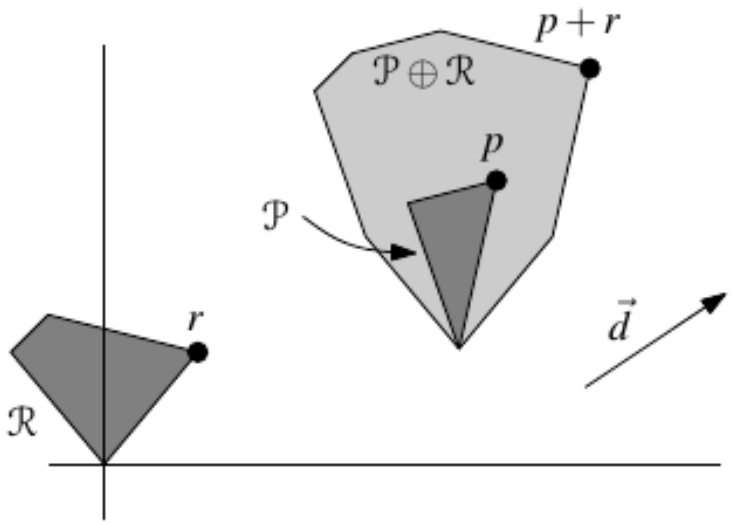


Figure: R vs (-R)

Minkowski sums

Observation. Let $\mathcal{P}$ and $\mathcal{R}$ be two objects in the plane, and let $\mathcal{CP} := \mathcal{P} \oplus \mathcal{R}$. An extreme point in direction $\overrightarrow{\mathrm{d}}$ on CP is the sum of extreme points in direction $\overrightarrow{\mathrm{d}}$ on $\mathcal{P}$ and $\mathcal{R}$.



Minkowski sums

Theorem. Let $\mathcal{P}$ and $\mathcal{R}$ be two convex polygons with n and m edges, respectively. Then the Minkowski sum $\mathcal{P} \oplus \mathcal{R}$ is a convex polygon with at most $n + m$ edges.

We now need an algorithm that computes the Minkowski sum of two convex polygons $\mathcal{P}$ and $\mathcal{R}$.

- For each pair v, w of vertices, with $v \in \mathcal{P}$ and $w \in \mathcal{R}$, compute $v + w$. Next, compute the convex hull of all these sums.
- Another faster way is described by the following algorithm.

Algorithm for the Minkowski sum

Algorithm MINKOWSKISUM($\mathcal{P}, \mathcal{R}$)
Input. A convex polygon $\mathcal{P}$ with vertices $v_1, \ldots, v_n$, and a convex polygon $\mathcal{R}$ with vertices $w_1, \ldots, w_m$. The lists of vertices are assumed to be in counterclockwise order, with v_1 and w_1 being the vertices with smallest y-coordinate (and smallest x-coordinate in case of ties).
Output. The Minkowski sum $\mathcal{P} \oplus \mathcal{R}$.

1. $i \leftarrow 1$; $j \leftarrow 1$
2. $v_{n+1} \leftarrow v_1$; $v_{n+2} \leftarrow v_2$; $w_{m+1} \leftarrow w_1$; $w_{m+2} \leftarrow w_2$
3. **repeat**
4. Add $v_i + w_j$ as a vertex to $\mathcal{P} \oplus \mathcal{R}$.
5. **if** $angle(v_i v_{i+1}) < angle(w_j w_{j+1})$
6. **then** $i \leftarrow (i+1)$
7. **else if** $angle(v_i v_{i+1}) > angle(w_j w_{j+1})$
8. **then** $j \leftarrow (j+1)$
9. **else** $i \leftarrow (i+1)$; $j \leftarrow (j+1)$
10. **until** $i = n+1$ **and** $j = m+1$

Complexity

Theorem. The Minkowski sum of two convex polygons with n and m vertices, respectively, can be computed in $O(n+m)$ time.

Minkowski sums

What happens if one or both of the polygons are not convex? In that case we can use the following equality three sets S_1, S_2 and S_3:

$$S_1 \oplus (S_2 \cup S_3) = (S_1 \oplus S_2) \cup (S_1 \oplus S_3).$$

Now consider the Minkowski sum of a non-convex polygon $\mathcal{P}$ and a convex polygon $\mathcal{R}$ with n and m vertices respectively. What is the complexity of $\mathcal{P} \oplus \mathcal{R}$?

We know that the polygon P can be triangulated into $n-2$ triangles $t_1, \ldots, t_{n-2}$, where n is its number of vertices. From the equality above we can conclude that

$$\mathcal{P} \oplus \mathcal{R} = \bigcup_{t=1}^{n-2} t_i \oplus \mathcal{R}$$

The complexity of $\mathcal{P} \oplus \mathcal{R}$ is O(nm). If both $\mathcal{P}$, $\mathcal{R}$ are non-convex, the complexity of the Minkowski sum is O(n^2m^2)

Translational Motion Planning

Theorem. Let $\mathcal{R}$ be a convex robot of constant complexity, translating among a set S of non-intersecting polygonal obstacles with a total of n edges. Then the complexity of the free configuration space $\mathcal{C}_{free}(\mathcal{R}, S)$ is $O(n)$.

It remains to find an algorithm to construct the free space. Rather than computing the free space $\mathcal{C}_{free}$, we shall compute the forbidden space $\mathcal{C}_{forb}$. The free space is simply its complement.

Algorithm FORBIDDENSPACE($\mathcal{CP}_1, \ldots, \mathcal{CP}_n$)
Input. A collection of $\mathcal{C}$-obstacles.
Output. The forbidden space $\mathcal{C}_{\text{forb}} = \bigcup_{i=1}^{n} \mathcal{CP}_i$.
1. **if** $n = 1$
2. **then return** $\mathcal{CP}_1$
3. **else** $\mathcal{C}^1_{\text{forb}} \leftarrow$FORBIDDENSPACE($\mathcal{P}_1, \ldots, \mathcal{P}_{\lceil n/2 \rceil}$)
4. $\mathcal{C}^2_{\text{forb}} \leftarrow$FORBIDDENSPACE($\mathcal{P}_{\lceil n/2 \rceil + 1}, \ldots, \mathcal{P}_n$)
5. Compute $\mathcal{C}_{\text{forb}} = \mathcal{C}^1_{\text{forb}} \cup \mathcal{C}^2_{\text{forb}}$.
6. **return** $\mathcal{C}_{\text{forb}}$

Work space and Configuration space

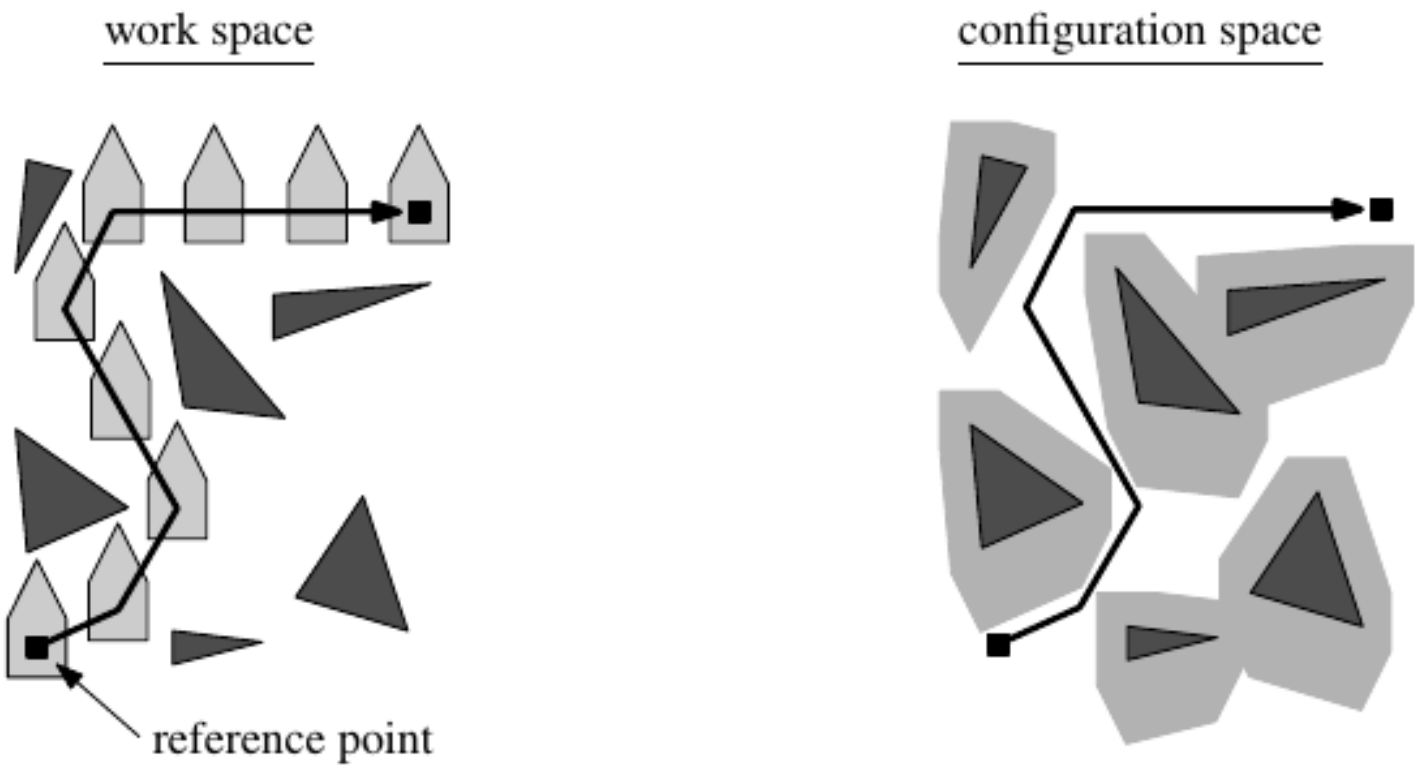
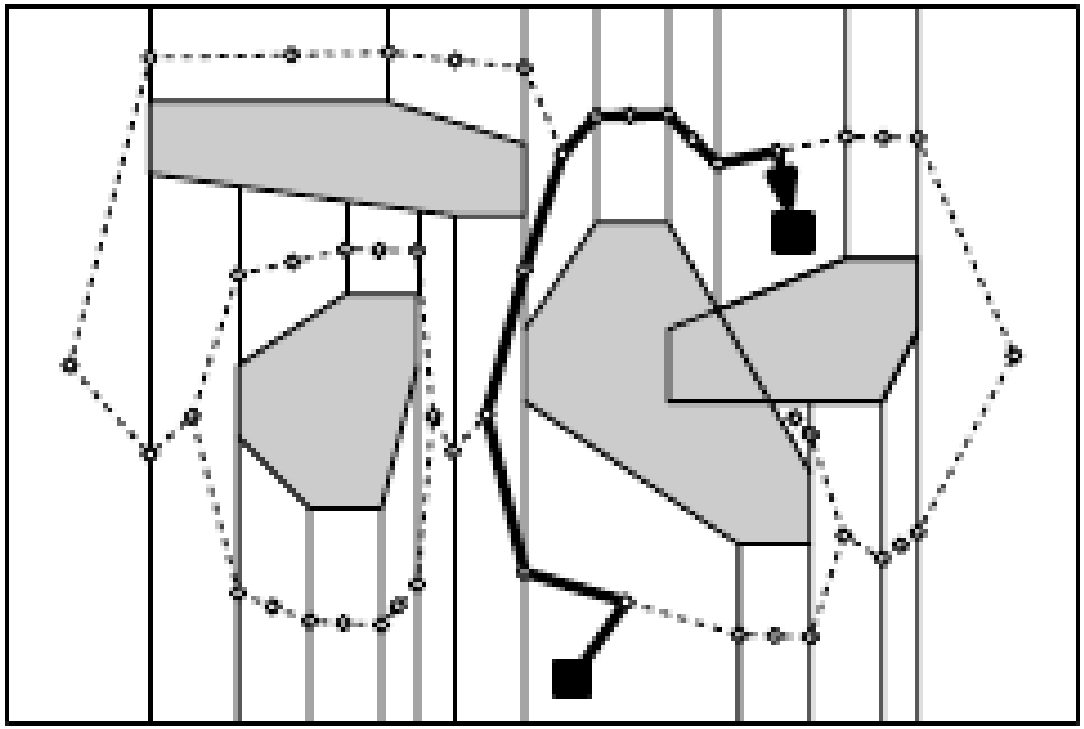


Figure: A path in the work space and the corresponding curve in the configuration space

Translational Motion Planning

Now that we have computed the free space, we can continue in exactly the same way as before.



Translational Motion Planning

- But what if we want to calculate the shortest path?

Finding the Shortest Route

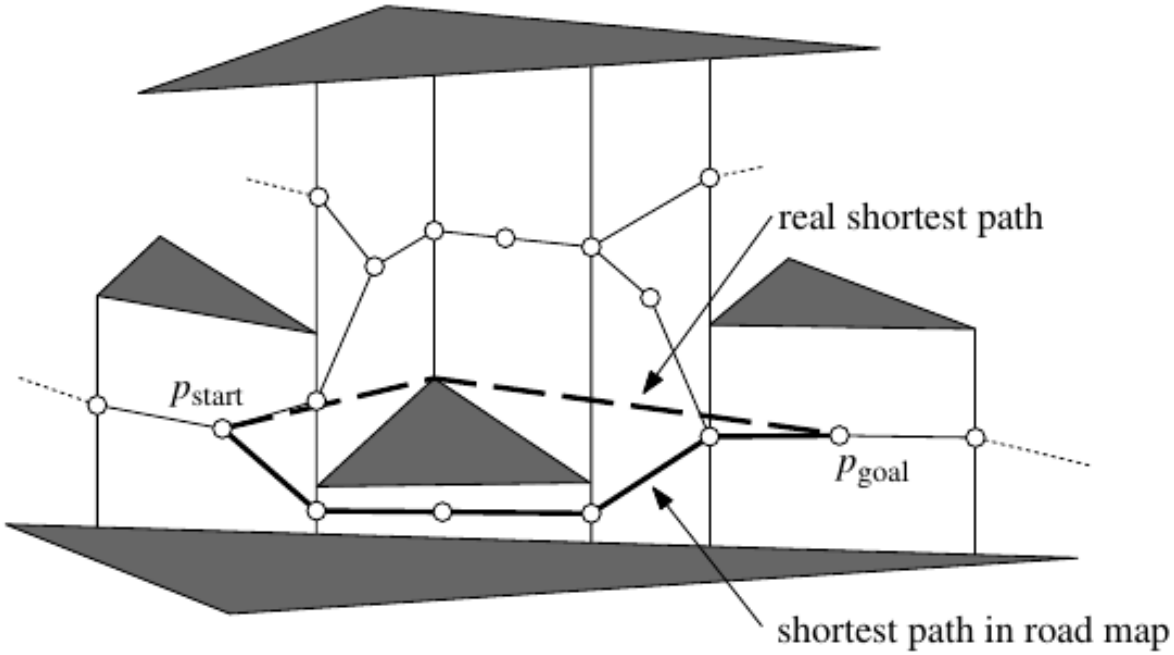


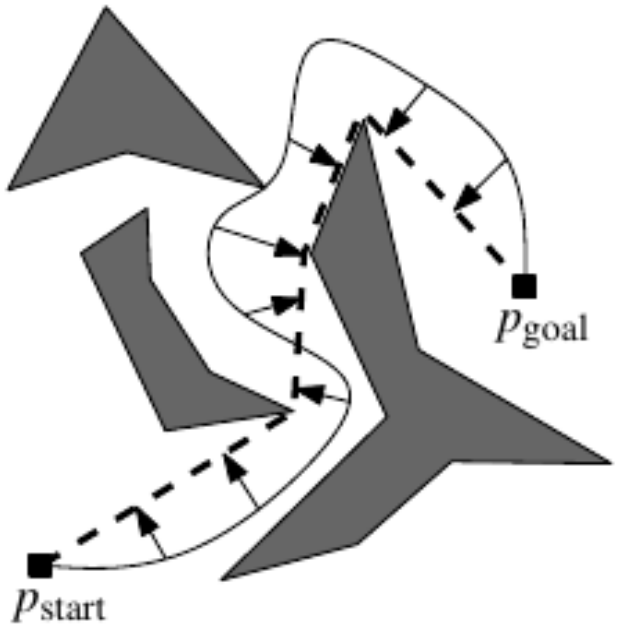
Figure: The shortest path does not follow the road map

Finding the Shortest Route

What can we say about the shape of a shortest path? Consider some path from p_{start} to p_{goal}.

We think of this path as an elastic rubber band, whose endpoints we fix at the start and goal position and which we force to take the shape of the path.

At the moment we release the rubber band, it will try to contract and become as short as possible, but it will be stopped by the obstacles.



Finding the Shortest Route

Lemma. Any shortest path between p_{start} and p_{goal} among a set S of disjoint polygonal obstacles is a polygonal path whose inner vertices are vertices of S.

With this characterization of the shortest path, we can construct a road map that allows us to find the shortest path. This road map is the visibility graph of S, which we denote by $\mathcal{G}_{vis}(S)$.

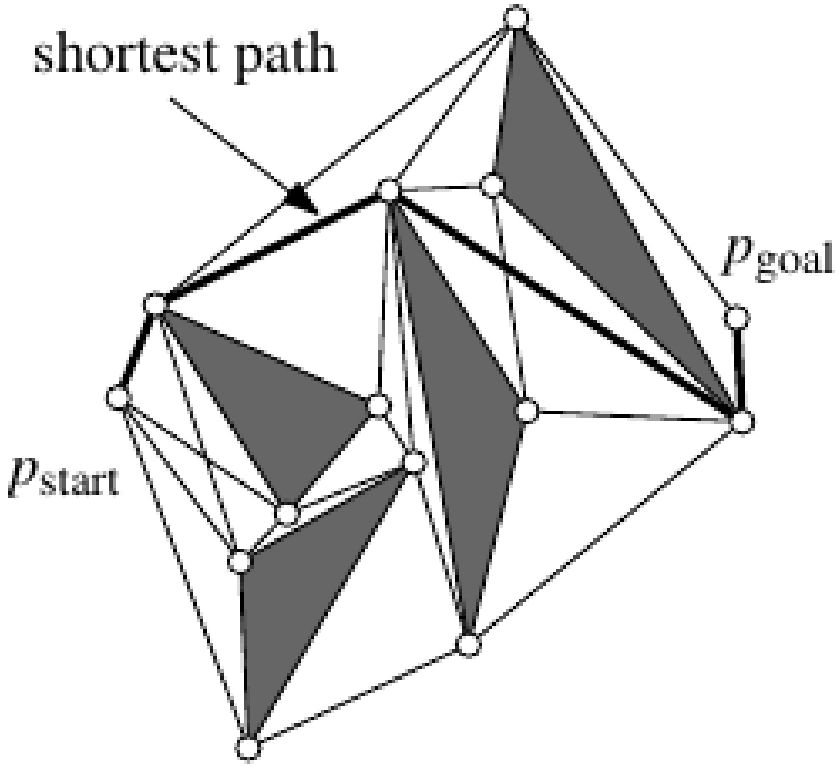
Its nodes are the vertices of S, and there is an arc between vertices v and w if they can see each other, that is, if the segment $\overline{vw}$ does not intersect the interior of any obstacle in S.

Finding the Shortest Route

Algorithm SHORTESTPATH($S, p_{\text{start}}, p_{\text{goal}}$)
Input. A set S of disjoint polygonal obstacles, and two points p_{start} and p_{goal} in the free space.
Output. The shortest collision-free path connecting p_{start} and p_{goal}.

1. $\mathcal{G}_{\text{vis}} \leftarrow$ VISIBILITYGRAPH($S \cup \{p_{\text{start}}, p_{\text{goal}}\}$)
2. Assign each arc (v, w) in $\mathcal{G}_{\text{vis}}$ a weight, which is the Euclidean length of the segment $\overline{vw}$.
3. Use Dijkstra's algorithm to compute a shortest path between p_{start} and p_{goal} in $\mathcal{G}_{\text{vis}}$.

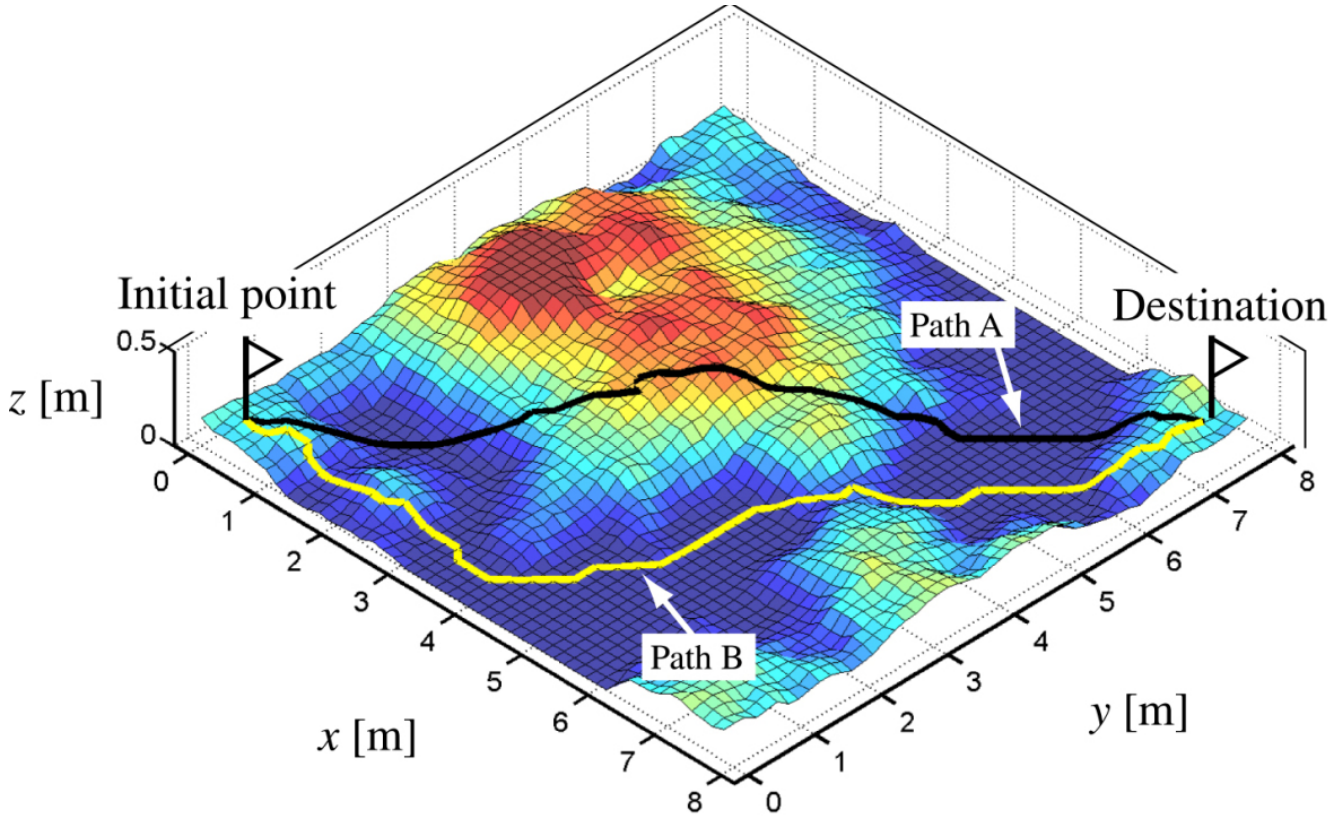
Finding the Shortest Route



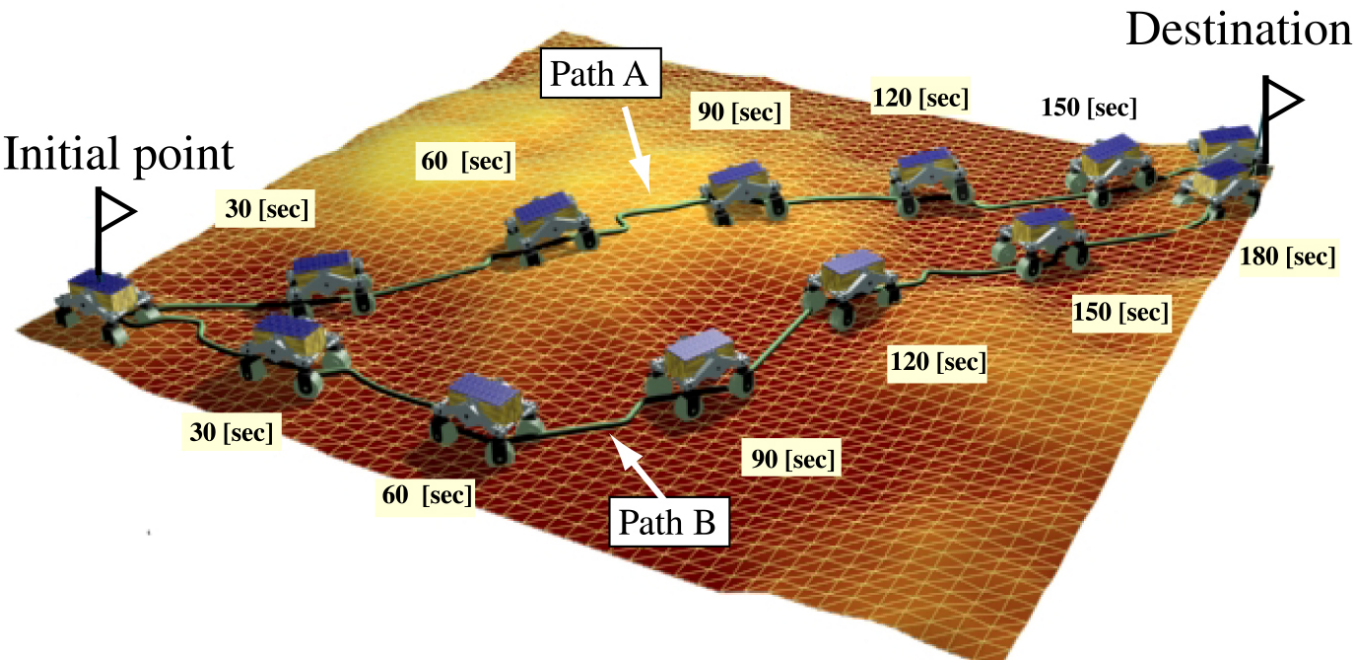
Complexity

A shortest path between two points among a set of polygonal obstacles with n edges in total can be computed in $O(n^2 logn)$ time.

Finding the Shortest Route



Finding the Shortest Route



References

Steven S Skiena. **The algorithm design manual: Text**. Vol. 1. Springer Science & Business Media, 1998.

Marc Van Kreveld et al. **Computational geometry algorithms and applications**. Springer, 2000.

https://en.wikipedia.org/wiki/Voronoi_diagram

https://en.wikipedia.org/wiki/Fortune%27s_algorithm

https://en.wikipedia.org/wiki/Delaunay_triangulation

https://cw.fel.cvut.cz/b181/_media/courses/cg/lectures/08-triang.pdf?fbclid=IwAR3wmVSbUeAncYxGTI5JRYBYh38_X2Ee2FV9XclEtlhvDqnGvP6HiSnCFLg

https://members.loria.fr/MTeillaud/collab/Astonishing/2017_workshop_slides/Olivier_Devillers.pdf

https://www2.cs.arizona.edu/classes/cs437/fall11/lec11.pdf

The end

Thank You!