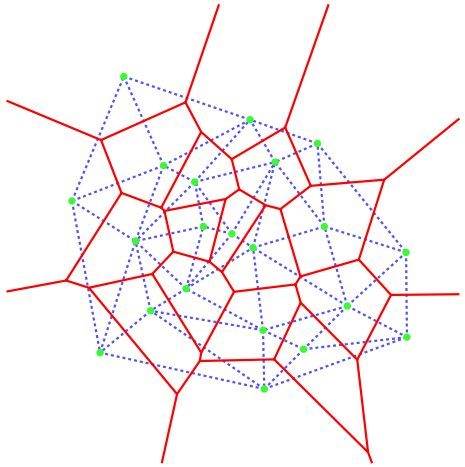
Algorithmische Geometrie

Voronoi Diagram

Martin Held
FB Computerwissenschaften
Universität Salzburg
A–5020 Salzburg, Austria

May 28, 2008



Acknowledgments

These slides are partially based on notes and slides transcribed by various students — most notably Elias Pschernig, Christian Spielberger, Werner Weiser and Franz Wilhelmstötter — for previous courses on “Algorithmische Geometrie”. Some figures were derived from figures originally prepared by students of my lecture “Wissenschaftliche Arbeitstechniken und Präsentation”. I would like to express my thankfulness to all of them for their help. This revision and extension was carried out by myself, and I am responsible for any errors.

I am also happy to acknowledge that we benefited from material published by colleagues on diverse topics that are partially covered in this lecture. While some of the material used for this lecture was originally presented in traditional-style publications (such as textbooks), some other material has its roots in non-standard publication outlets (such as online documentations, electronic course notes, or user manuals).

Salzburg, January 2008 Martin Held

Voronoi Diagram

- Proximity Problems and Lower Bounds,
- Voronoi Diagram: Definition and Basic Facts,
- Voronoi Diagram: Properties,
- Delaunay Triangulation: Definition and Basic Facts,
- Algorithms for Constructing Voronoi Diagrams,
- Generalized Voronoi Diagram,
- Applications of Voronoi Diagrams.

A Set of Proximity Problems

- Consider a set $S := \{p_1, p_2, \ldots, p_n\}$ of n points in $\mathbb{E}^2$, i.e., in $\mathbb{R}^2$ under the Euclidean metric.

- CLOSESTPAIR: Determine two points of S whose mutual distance is smallest.

- ALLNEARESTNEIGHBORS: Determine the "nearest neighbor" (point of minimum distance within S) for each point in S.

- EUCLIDEANMINIMUMSPANNINGTREE (EMST): Construct a tree of minimum total (Euclidean) length whose vertices are the points of S. (No Steiner points allowed.)

- MAXIMUMEMPTYCIRCLE: Find a circle with largest radius which does not contain a point of S in its interior and whose center lies within $CH(S)$.

- TRIANGULATION: Join the points in S by non-intersecting straight-line segments so that every region internal to the convex hull of S is a triangle.

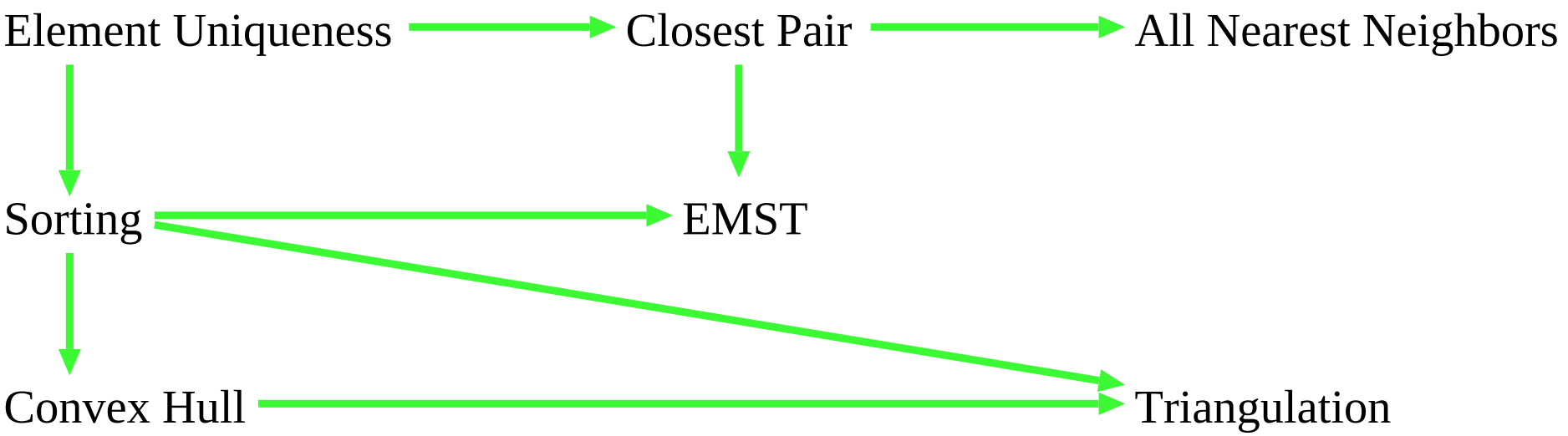
- NEARESTNEIGHBORSEARCH: Given a query point q, which point $p \in S$ is a nearest neighbor of q?

Lower Bounds

- NEARESTNEIGHBORSEARCH: standard argument yields $\Omega(\log n)$ comparisons.
- CLOSESTPAIR has an $\Omega(n \log n)$ lower bound since ELEMENTUNIQUENESS is linearly transformable to CLOSESTPAIR.
- ALLNEARESTNEIGHBORS has an $\Omega(n \log n)$ lower bound since CLOSESTPAIR is linearly transformable to ALLNEARESTNEIGHBORS.
- EMST: an EMST contains a closest pair, which establishes the $\Omega(n \log n)$ lower bound. (Also, SORTING can be transformed linearly to EMST.)
- MAXIMUMEMPTYCIRCLE in 1D solves MAXGAP, which establishes the $\Omega(n \log n)$ lower bound.
- TRIANGULATION has an $\Omega(n \log n)$ lower bound since CONVEXHULL is linearly transformable to TRIANGULATION. (Also, SORTING can be transformed linearly to TRIANGULATION.)

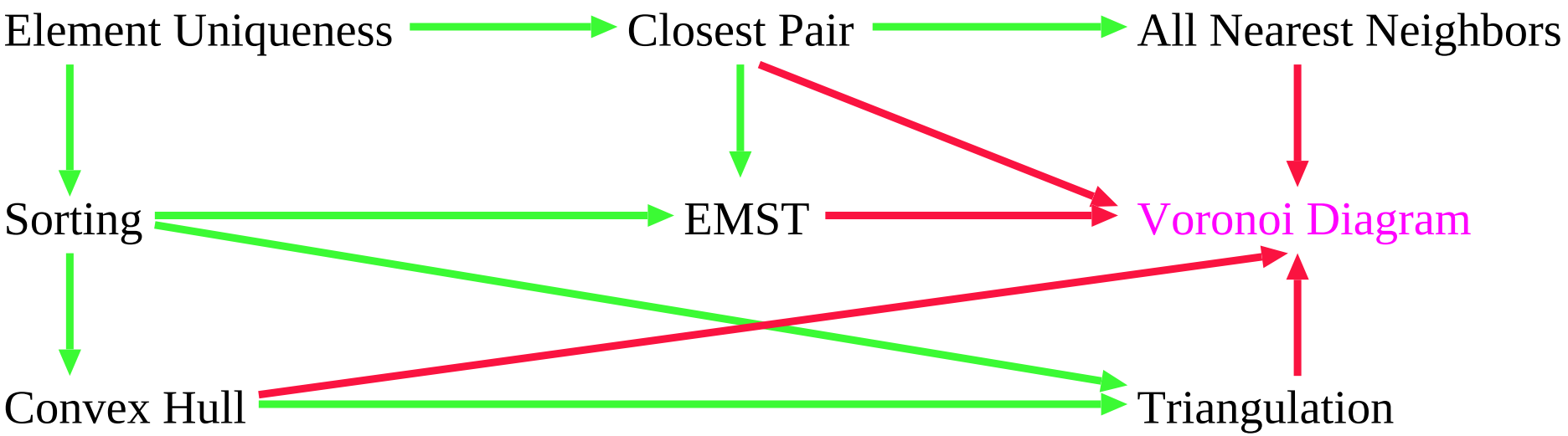
Lower Bounds: Summary of Reductions

- We have $\Omega(n \log n)$ lower bounds due to a variety of reductions.



Voronoi Diagram: Definition and Basic Facts

- If Voronoi diagram is available then proximity problems can be solved in $O(n)$ time!

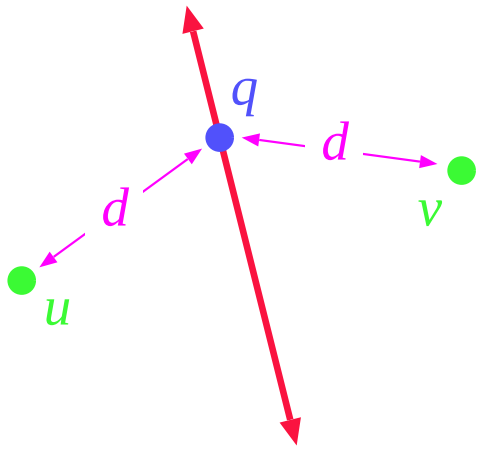


Voronoi Diagram: Definition and Basic Facts (cont'd)

- Consider a set $S := \{p_1, \cdots, p_n\}$ of n distinct points in $\mathbb{R}^2$.
- General position assumed: no four points are co-circular!
- Def.: The *bisector* of two points $u, v \in \mathbb{R}^2$ is the set of points of $\mathbb{R}^2$ which are equidistant to u and v:

$$b(u,v) := \{q \in \mathbb{R}^2 : d(u,q) = d(v,q)\},$$

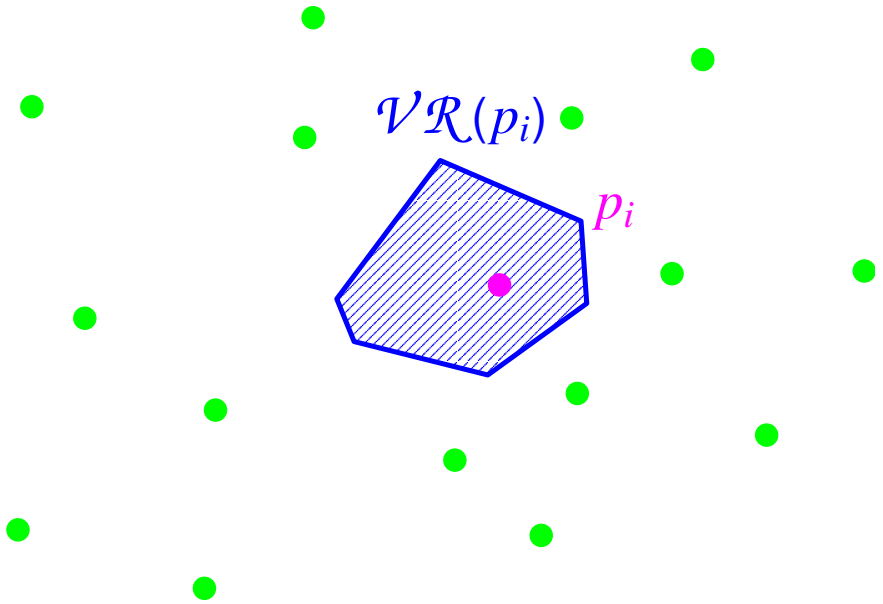
where $d(\cdot,\cdot)$ denotes the Euclidean distance.



Voronoi Diagram: Definition and Basic Facts (cont'd)

- Def.: The *Voronoi region* (VR, aka "Voronoi cell") of a point $p_i \in S$ is the locus of points of $\mathbb{R}^2$ whose distance to p_i is not greater than the distance to any other point of S:

$$\mathcal{VR}(p_i) := \{q \in \mathbb{R}^2 : d(p_i, q) \leq d(S \setminus \{p_i\}, q)\}.$$



Voronoi Diagram: Definition and Basic Facts (cont'd)

- Def.: A *Voronoi polygon* (VP) is defined as

$$\mathcal{VP}(p_i) := \{q \in \mathbb{R}^2 : d(p_i, q) = d(S \setminus \{p_i\}, q)\}.$$

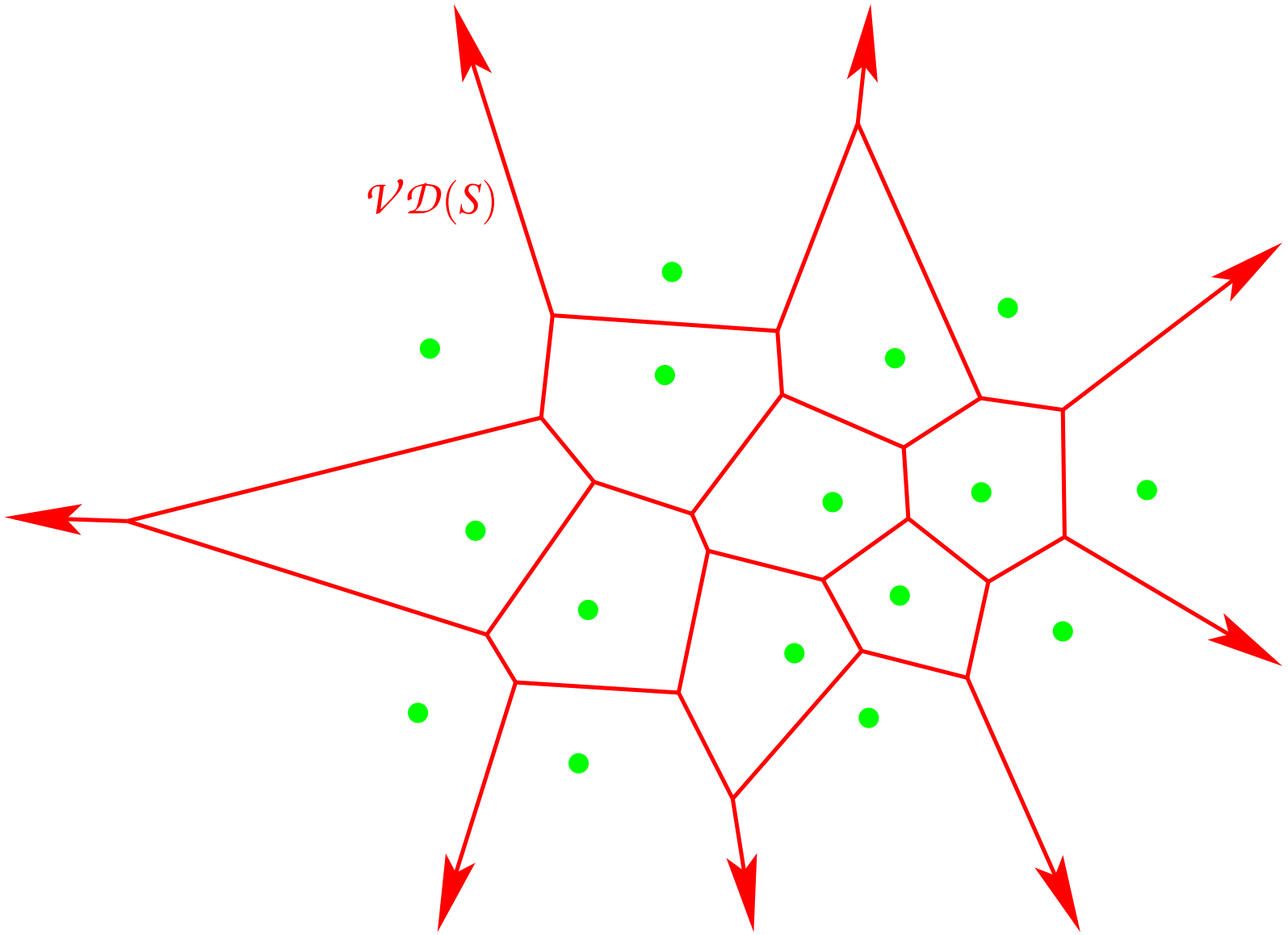
- A Voronoi polygon forms the (polygonal) boundary of a Voronoi region.

- The segments of a Voronoi polygon are called *Voronoi edges*.

- Def.: The *Voronoi diagram* (VD) of S is defined as

$$\mathcal{VD}(S) := \bigcup_{1 \le i \le n} \mathcal{VP}(p_i).$$

Voronoi Diagram: Definition and Basic Facts (cont'd)

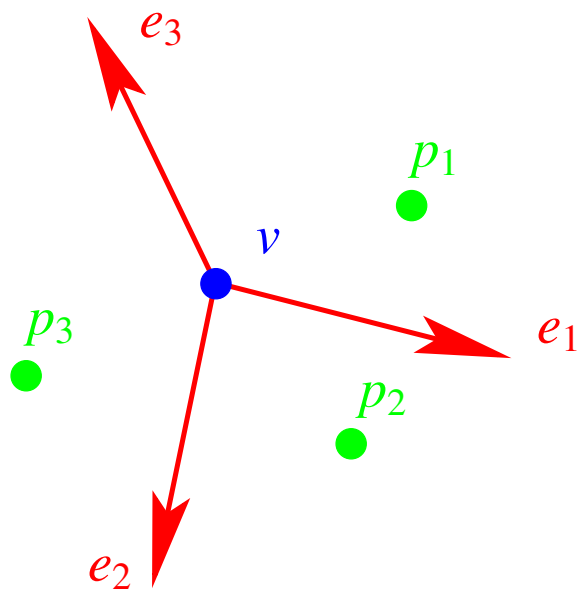


The Voronoi diagram $\mathcal{VD}(S)$ of the point set S.

Voronoi Diagram: Definition and Basic Facts (cont'd)

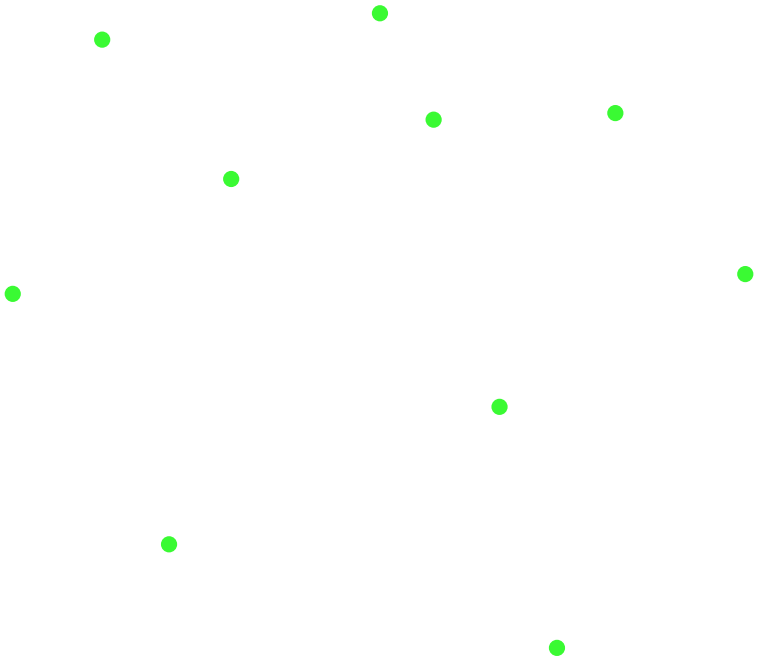
- A Voronoi edge always lies on a bisector. Thus, points on a Voronoi edge are equidistant to two points of S.

- Intersections of Voronoi edges are called *Voronoi nodes*.



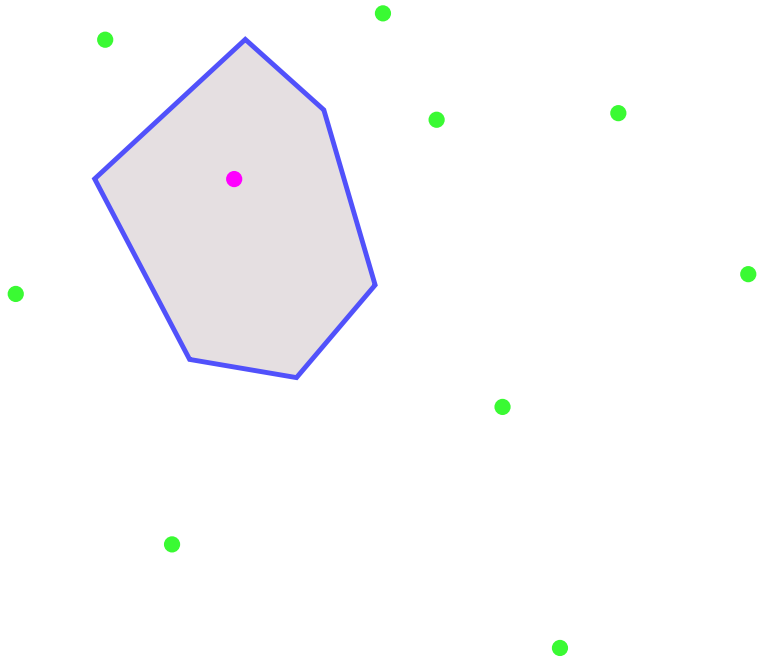
Sample Voronoi Diagram

- Input S.



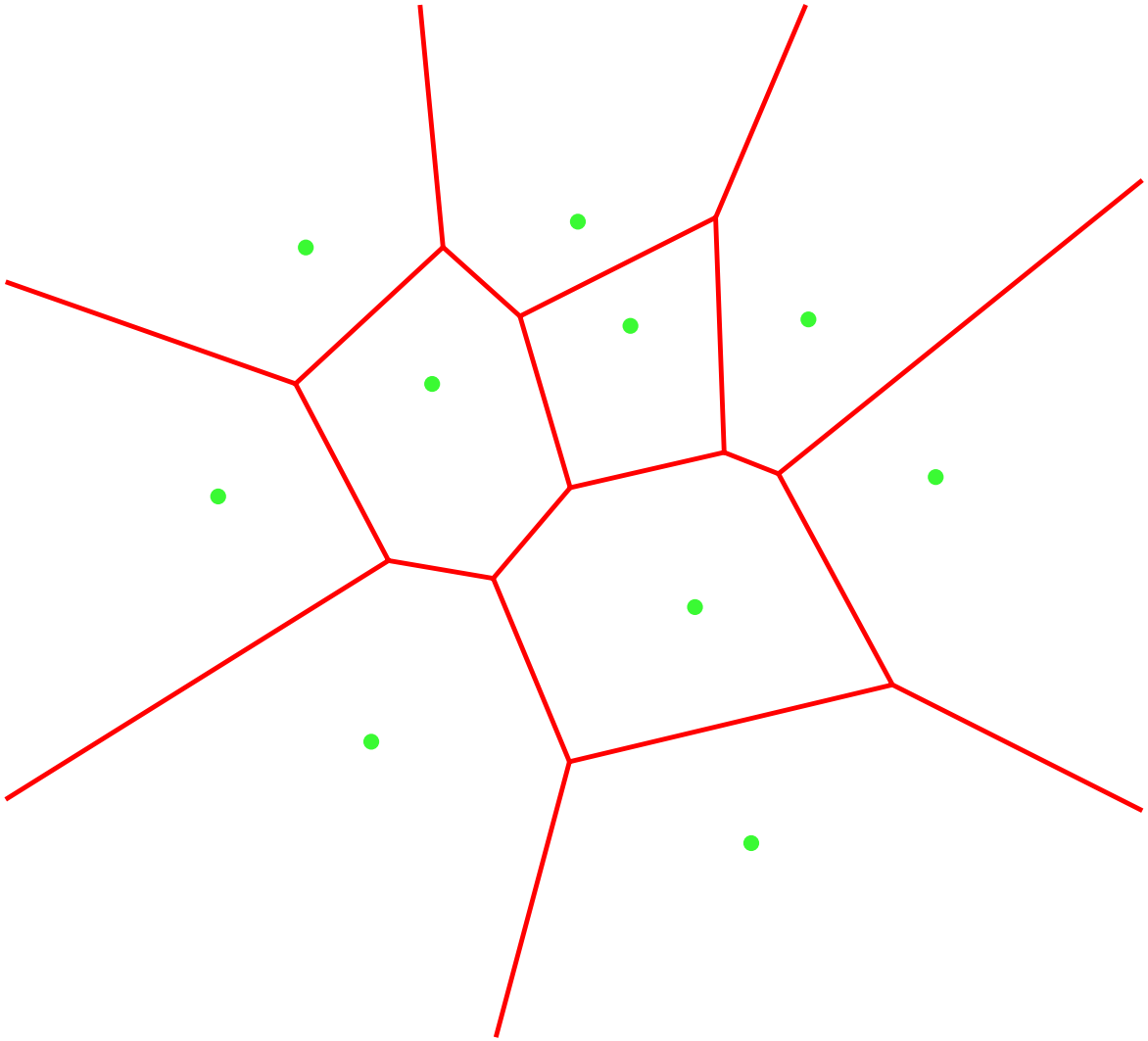
Sample Voronoi Diagram

- Input S, Voronoi region.



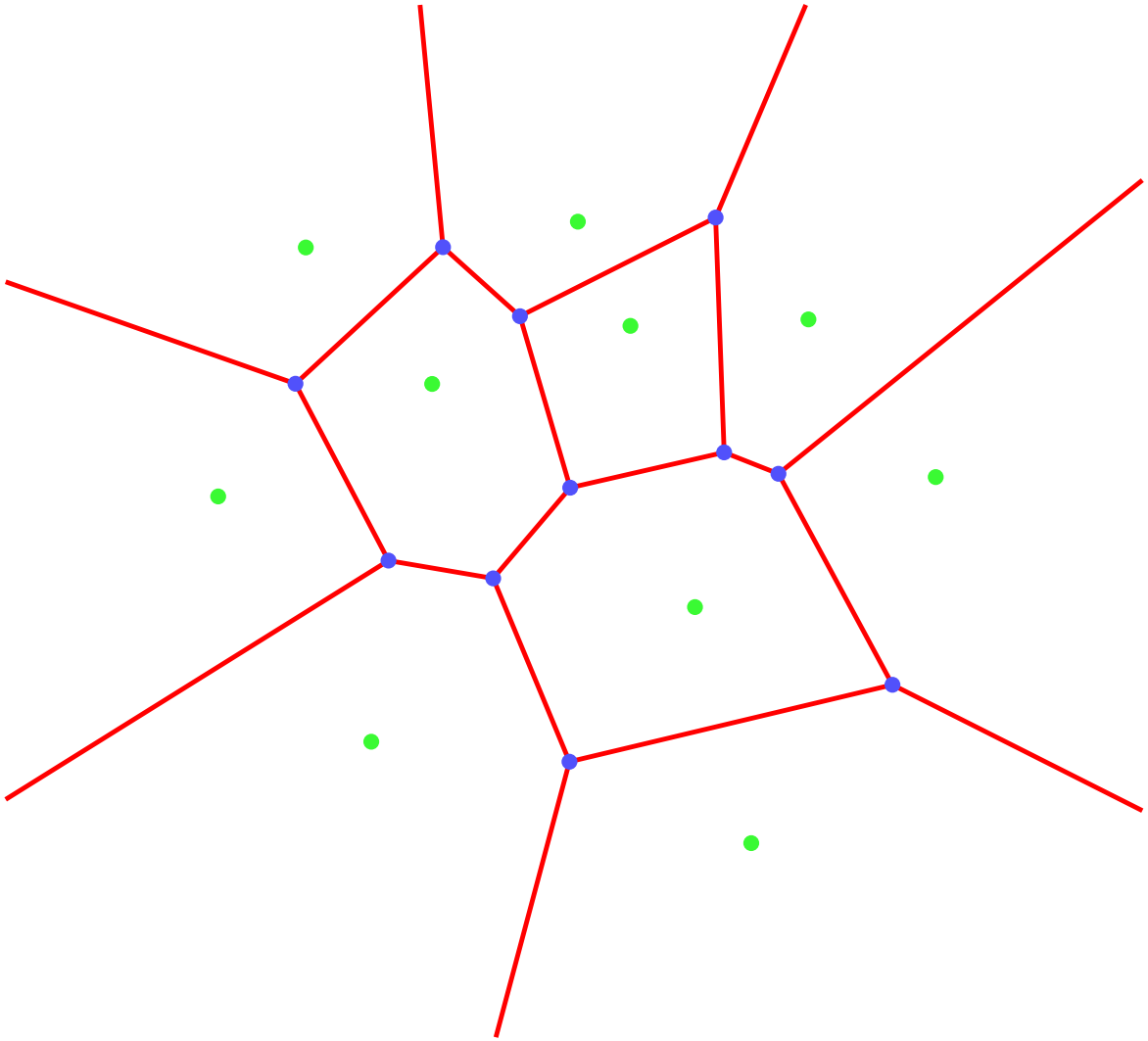
Sample Voronoi Diagram

- Input S, Voronoi region, Voronoi diagram.



Sample Voronoi Diagram

- Input S, Voronoi region, Voronoi diagram, Voronoi nodes.



Voronoi Diagram: Definition and Basic Facts (cont'd)

- Lemma: The Voronoi region $\mathcal{VR}(p_i)$ is the intersection of half-planes defined by bisectors between p_i and the other points of S:

$$\mathcal{VR}(p_i) = \bigcap_{\substack{1 \le j \le n \\ j \neq i}} H(p_i, p_j),$$

  where $H(p_i, p_j)$ is the half-space that contains p_i.

- Thus, a Voronoi region is a convex polygonal area.

- Lemma: Every point of S has its own Voronoi region that is not empty.

- Lemma: The (topological) interiors of Voronoi regions of distinct points of S are disjoint.

Voronoi Diagram: Properties

- Lemma: A Voronoi node is the common intersection of exactly three Voronoi edges. It is equidistant to the three points of S which lie in the Voronoi regions it belongs to.

- Proof:
  - ⋆ Let a Voronoi node v be the intersection of k edges $e_1, e_2, \ldots, e_k$, with $k \geq 2$, which are ordered clockwise around v. Then e_1 is equidistant to some points p_1 and p_2, e_2 is equidistant to p_2 and p_3, and so on, and e_k is equidistant to p_k and p_1.
  - ⋆ Thus, v is equidistant to $p_1, p_2, \ldots, p_k$.
  - ⋆ Since a Voronoi region is convex, all points $p_1, p_2, \ldots, p_k$ are distinct.
  - ⋆ Based on our assumption that no more than three points are co-circular we conclude $k \leq 3$.
  - ⋆ However, $k = 2$ would mean e_1 is equidistant to p_1 and p_2, and e_2 is equidistant to p_2 and p_1. Therefore, they would lie on the same bisector, and could not intersect in v.

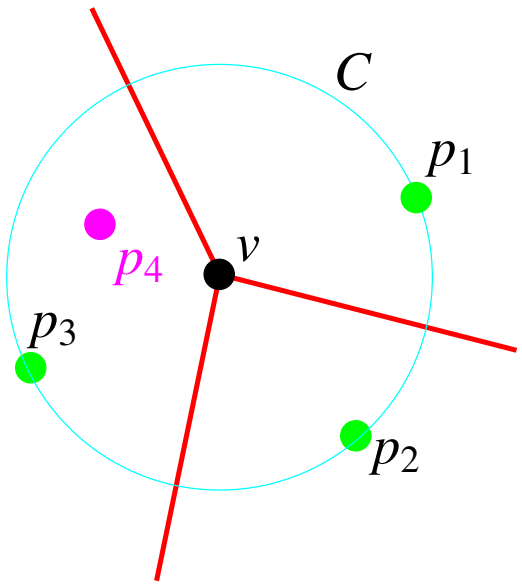
- This means that a Voronoi Diagram is a 3-regular (planar) graph.

- Note that without the general-position assumption the degree of a Voronoi node can get as high as n.

Voronoi Diagram: Properties (cont'd)

- Lemma: The circle C centered at a Voronoi node v that passes through the node's three equidistant points $p_1, p_2, p_3 \in S$ contains no other points of S in its interior.

- Proof:
  - ⋆ Assume that C contains another point $p_4 \in S$ in its interior.
  - ⋆ Then v would be closer to p_4 than to any of p_1, p_2, p_3. Therefore, v would lie in the Voronoi region of p_4.
  - ⋆ This is a contradiction because v lies in the Voronoi regions of p_1, p_2, p_3.

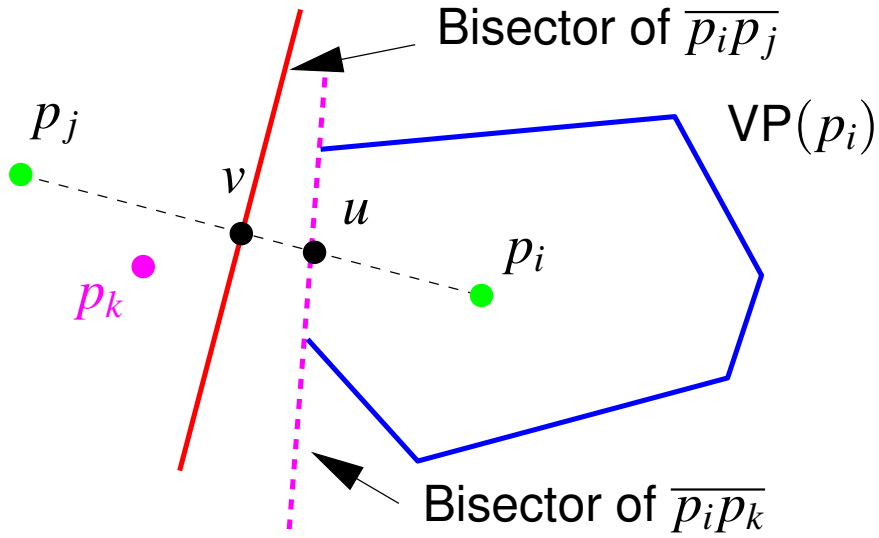


Voronoi Diagram: Properties (cont'd)

- Lemma: For $p_i \in S$, every nearest neighbor of p_i defines an edge of $\mathcal{VP}(p_i)$.

- Proof:

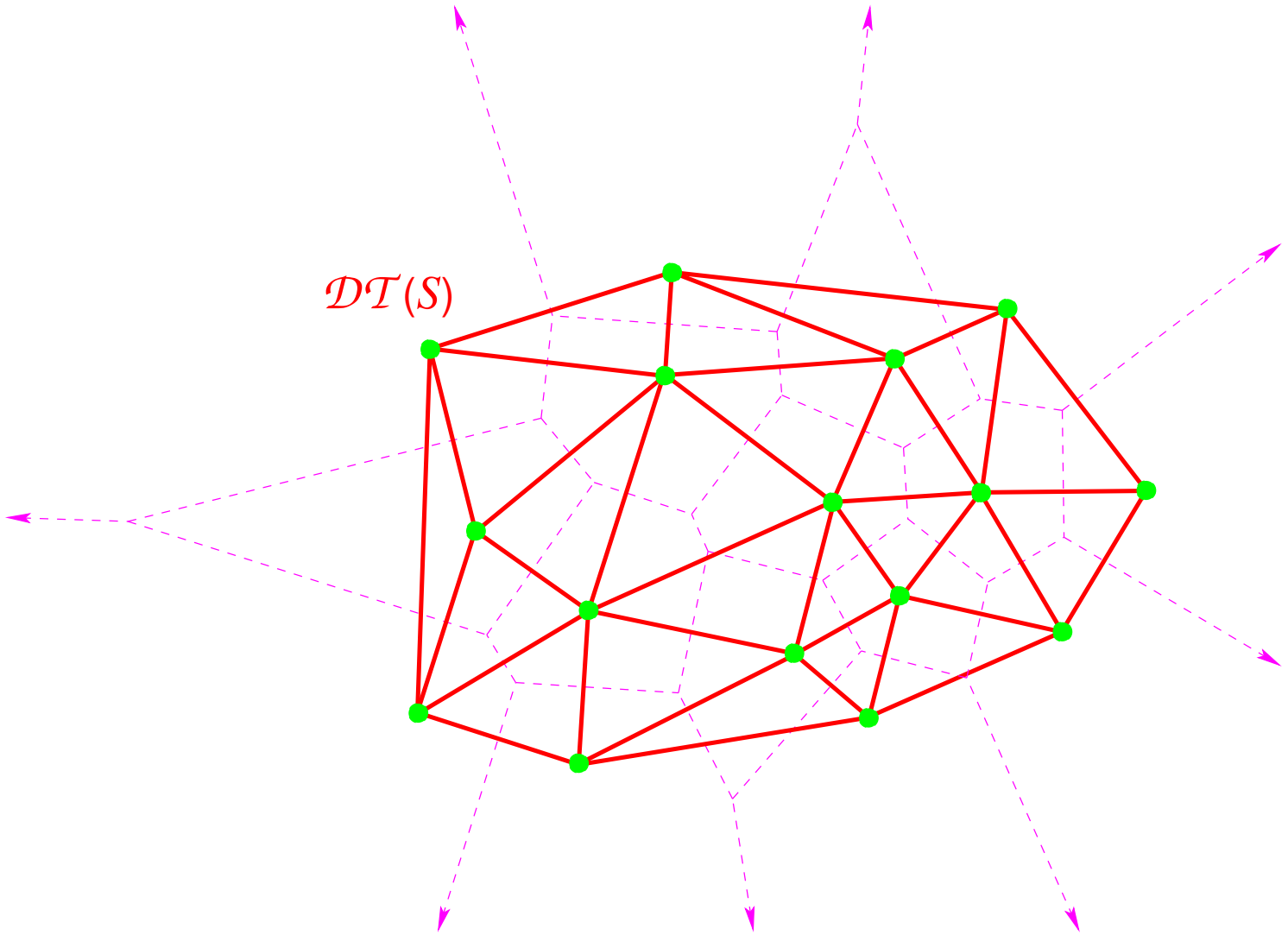
  ⋆ Let $p_j \in S$ be a nearest neighbor of p_i, and let v be their midpoint.
  ⋆ Suppose that v does not lie on the boundary of $\mathcal{VP}(p_i)$.
  ⋆ Then the line segment $\overline{p_i v}$ would intersect some edge of $\mathcal{VP}(p_i)$. Assume that it intersects the bisector of $\overline{p_i p_k}$ in the point u. Now $|\overline{p_i u}| < |\overline{p_i v}|$, and therefore $|\overline{p_i p_k}| \leq 2|\overline{p_i u}| < 2|\overline{p_i v}| = |\overline{p_i p_j}|$, and we would have p_k closer to p_i than p_j, which is a contradiction.



Delaunay Triangulation: Definition and Basic Facts

- Def.: A *Delaunay triangulation* (DT) of S is a graph that is *dual* to the Voronoi diagram of S:
  - ⋆ The nodes of the graph are given by the points of S.
  - ⋆ Two points are connected by a line segment, and form an edge of $\mathcal{DT}(S)$, exactly if they share a Voronoi edge of $\mathcal{VD}(S)$.
- Thus, the interior faces of $\mathcal{DT}(S)$ are defined by triples of S which correspond to nodes of $\mathcal{VD}(S)$.
- $\mathcal{DT}(S)$ does indeed form a triangulation of S. (Proof: See Preparata&Shamos.)

Delaunay Triangulation: Definition and Basic Facts (cont'd)



Delaunay Triangulation $\mathcal{DT}(S)$, with the underlying Voronoi diagram.

Delaunay Triangulation: Definition and Basic Facts (cont'd)

- By definition, every edge of the Delaunay triangulation has a corresponding edge in the Voronoi diagram.
- $\mathcal{DT}(S)$ is called the *straight-line dual* of $\mathcal{VD}(S)$.
- Note: An edge of $\mathcal{DT}(S)$ need not intersect its dual Voronoi edge.
- If no four points of a point set are co-circular then its Delaunay triangulation is unique.

Complexity of Voronoi Diagram and Delaunay Triangulation

- Recall that a Delaunay triangulation forms a very special planar graph on n nodes, to which Euler's formula

$$V - E + F = 2$$

can be applied. This implies that we have

  - ⋆ at least $E \geq F * 3/2$ edges,
  - ⋆ at most $F \leq E * 2/3$ faces.

We conclude that

$\mathcal{DT}$:	$\leq 3n-6$ edges	and thus	$\mathcal{VD}$:	$\leq 3n-6$ edges,
$\mathcal{DT}$:	$\leq 2n-4$ faces	and thus	$\mathcal{VD}$:	$\leq 2n-5$ nodes.

- Note: one Voronoi polygon may have $n-1$ edges but the entire Voronoi diagram of n points can have only $3n-6$ edges in total. Thus, $\mathcal{VD}(S)$ and $\mathcal{DT}(S)$ can be stored in $O(n)$ space!

- Since every edge belongs to two Voronoi polygons, a Voronoi polygon has only six edges on average.

Proximity Problems Solved by Voronoi Diagrams

- The fact that the Voronoi polygons of nearest neighbors always have a Voronoi edge in common implies that it is sufficient to check all points in adjacent Voronoi regions to find a nearest neighbor of a point p_i.

- Thus, knowledge of the Voronoi diagram helps to solve CLOSESTPAIR and ALL-NEARESTNEIGHBORS in $O(n)$ time.

- Lemma: The Voronoi polygon of p_i is unbounded if and only if p_i is a point of the convex hull of the set S. (Proof: See Preparata&Shamos.) This means that the vertices of $CH(S)$ are those points of S which have unbounded Voronoi polygons.

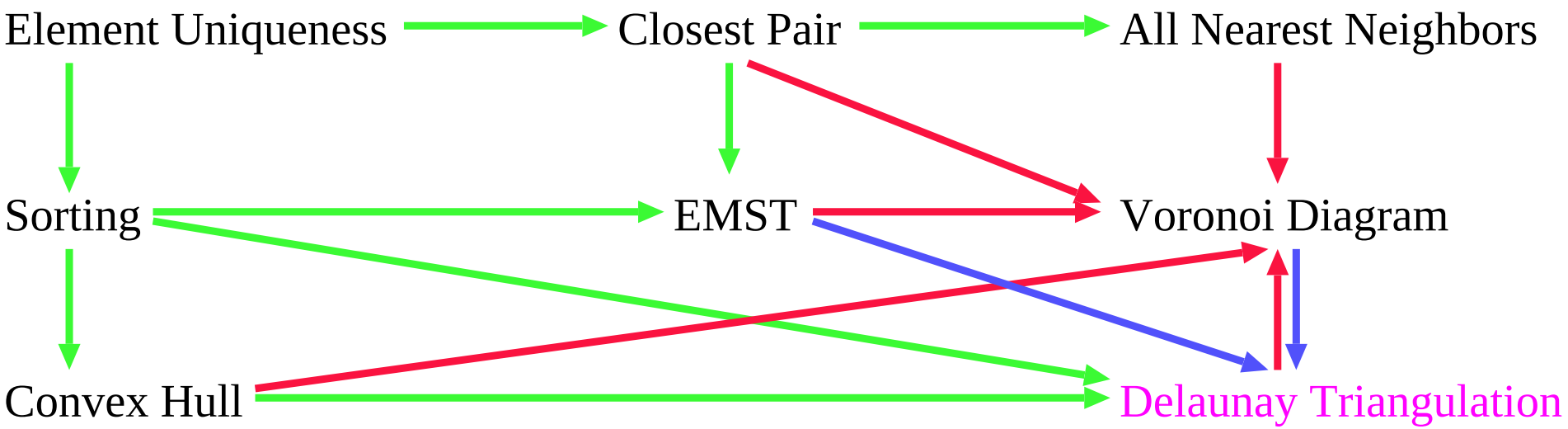
- Thus, knowledge of the Voronoi diagram allows to solve CONVEXHULL in $O(n)$ time.

- A MAXIMUMEMPTYCIRCLE can be found in $O(n)$ time by scanning all nodes of the Voronoi diagram; see later.

- After $O(n \log n)$ preprocessing for building a search data structure of size $O(n)$ on top of the Voronoi diagram, NEARESTNEIGHBORSEARCH queries can be handled in $O(\log n)$ time.

Reductions Among Proximity Problems

- Lemma: The Voronoi diagram can be obtained in $O(n)$ time from the Delaunay triangulation, and the Delaunay triangulation can be obtained in $O(n)$ time from the Voronoi diagram.



Algorithms for Constructing Voronoi Diagrams

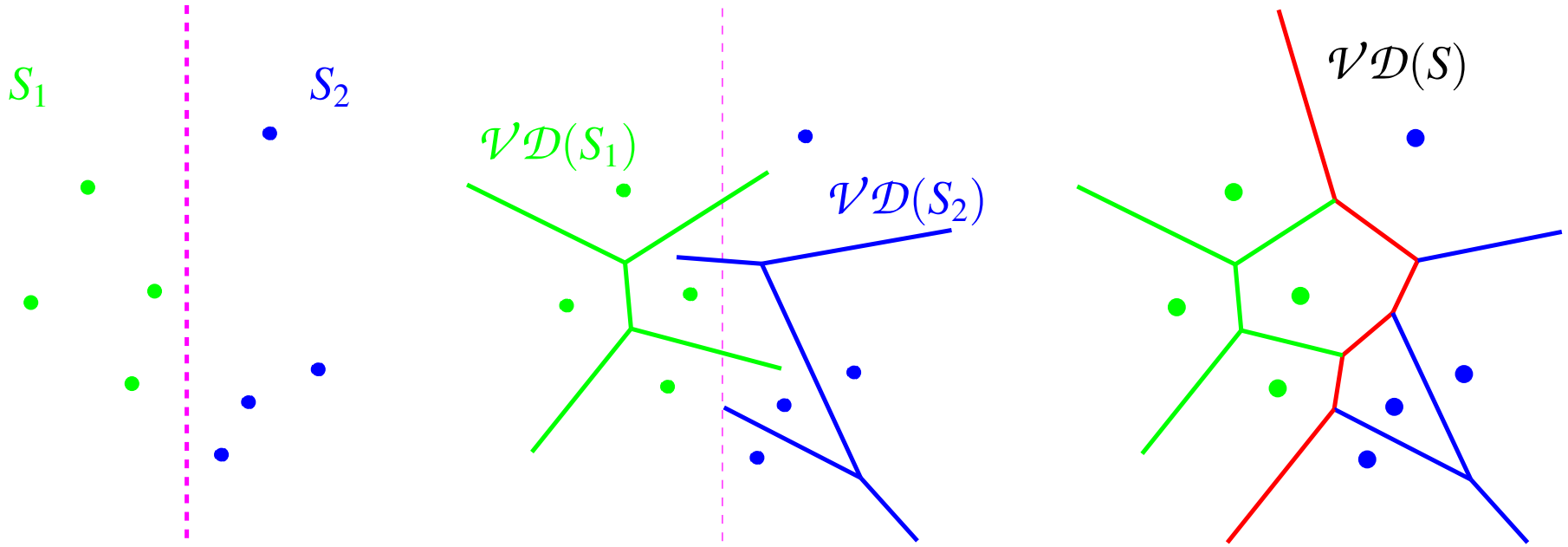
- Divide&Conquer Algorithm,
- Incremental Construction,
- Sweep-Line Algorithm,
- Construction via Lifting to 3D,
- Approximate Voronoi Diagram by Means of Graphics Hardware.

Divide&Conquer Algorithm

- Preprocessing: sort the points of S by x-coordinates. This takes $O(n \log n)$ time.
- Divide-Step:
  - ⋆ Divide S into two subsets S_1 and S_2 of roughly equal size such that the points in S_1 lie to the left and the points in S_2 lie to the right of a vertical line.
  - ⋆ This step can be carried out in $O(n)$ time.
- Conquer-Step (aka “Merge”):
  - ⋆ Assume that $\mathcal{VD}(S_1)$ and $\mathcal{VD}(S_2)$ are known.
  - ⋆ Clip those parts of $\mathcal{VD}(S_1)$ that lie to the “right” of a so-called *dividing chain*.
  - ⋆ Analogously for $\mathcal{VD}(S_2)$.

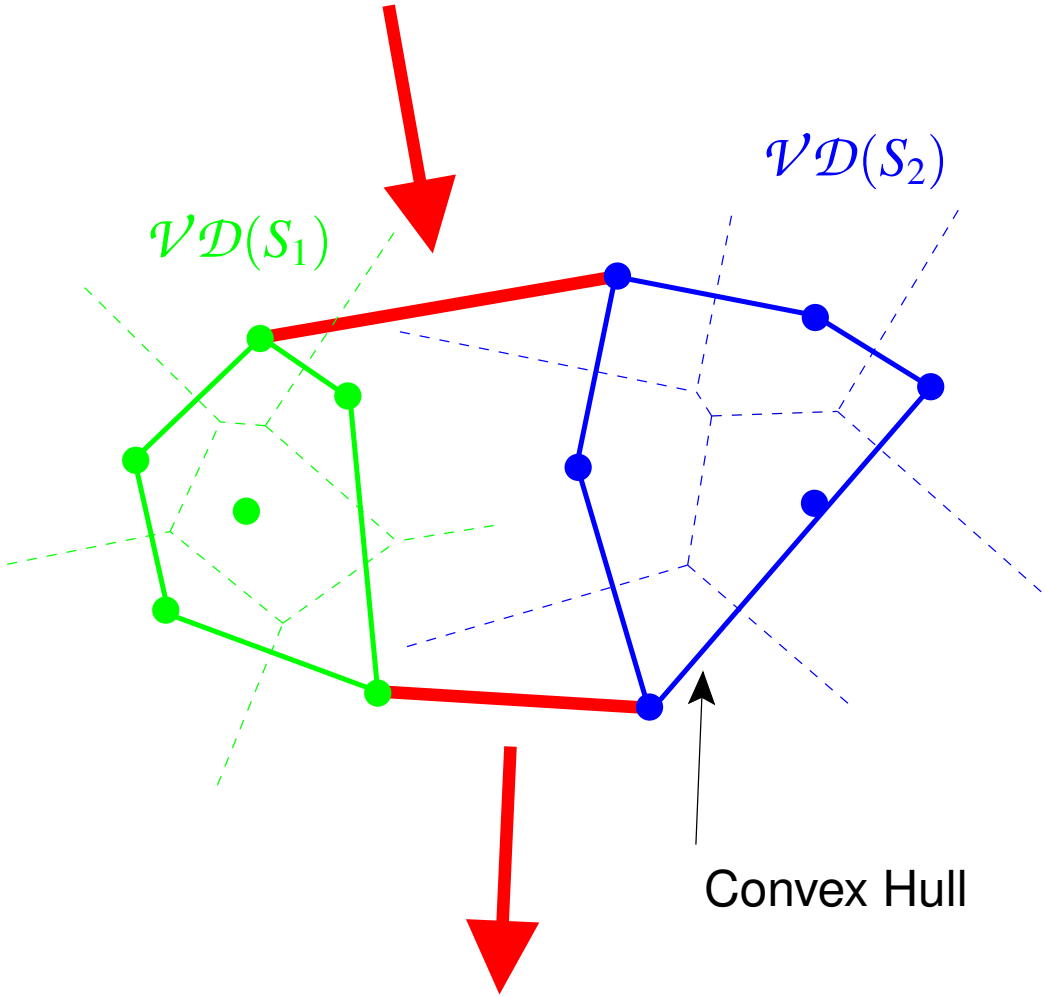
Divide&Conquer Algorithm (cont'd)

- The merge step is carried out by incrementally generating the *dividing chain*, and by clipping $\mathcal{VD}(S_1)$ and $\mathcal{VD}(S_2)$ appropriately.



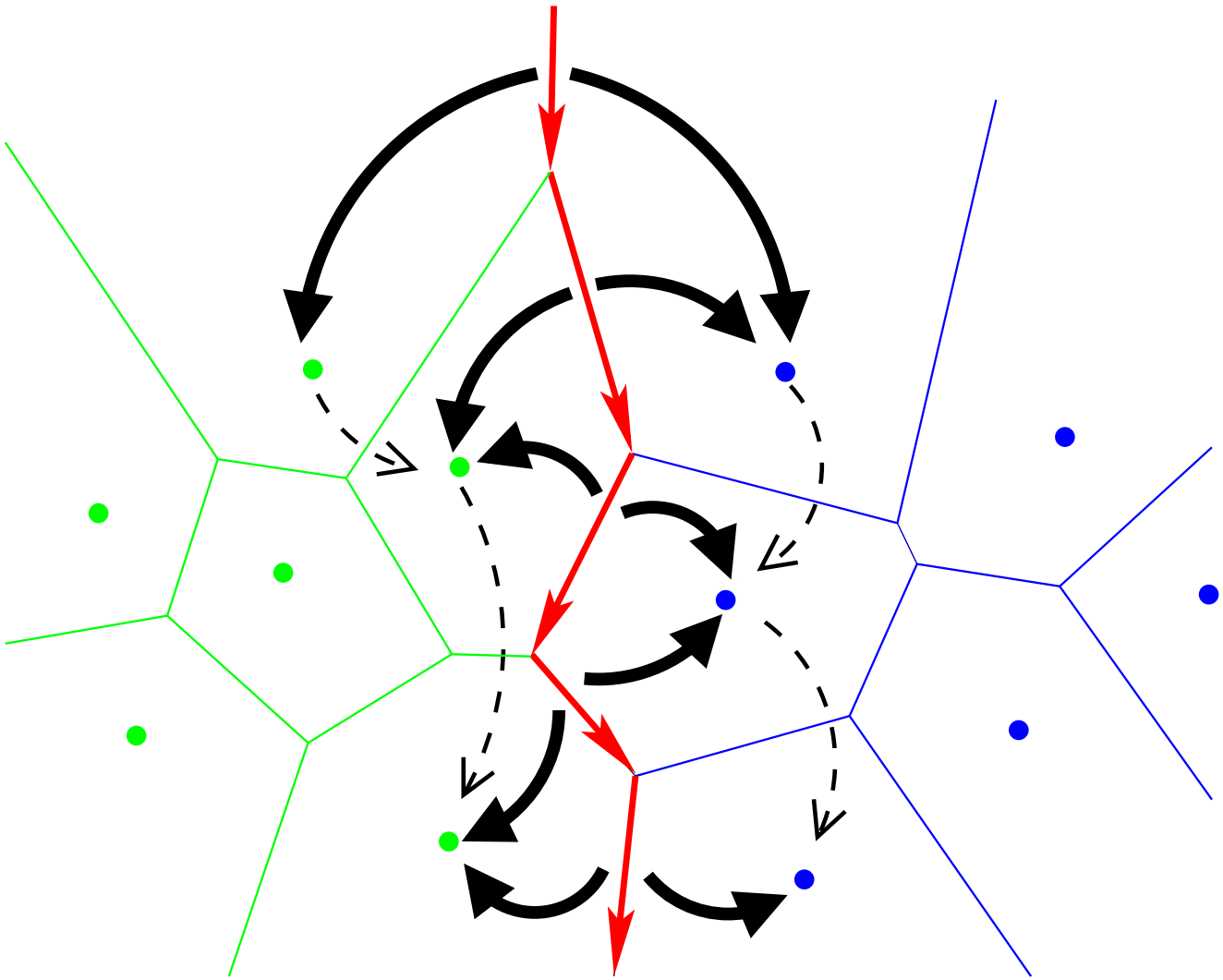
Divide&Conquer Algorithm: Merge

- First, find upper and lower bridges of the convex hull.
  - ⋆ Bisector (ray) defined by upper bridge of convex hull is part of the dividing chain.
  - ⋆ Bisector (ray) defined by lower bridge of convex hull is part of the dividing chain.



Divide&Conquer Algorithm: Merge (cont'd)

- Build dividing chain from top to bottom:
  - ⋆ Start by walking down along the upper ray.
  - ⋆ Intersect the ray with $\mathcal{VD}(S_1)$ and $\mathcal{VD}(S_2)$.
  - ⋆ Pick the first intersection as new Voronoi node.
  - ⋆ The next ray is the new bisector originating at this node.
  - ⋆ Continue this jagged walk until the lower ray is reached.



Divide&Conquer Algorithm: Complexity Analysis

- The merge can be carried out in $O(n)$ time, based on the Shamos-Hoey scanning scheme that prevents Voronoi edges from being searched for an intersection for more than a constant number of times.

- If the merge is carried out in linear time then we get a familiar recurrence relation for the time T:

$$T(n) = 2T(\frac{n}{2}) + O(n), \qquad \text{and thus } T(n) = O(n \log n).$$

- We conclude that a divide&conquer algorithm can compute $\mathcal{VD}(S)$ in optimal $O(n \log n)$ time.

Incremental Construction

- We compute the Voronoi diagram $\mathcal{VD}(S)$ of a set $S := \{p_1, p_2, \ldots, p_n\}$ of n points by inserting the i-th point p_i into $\mathcal{VD}(\{p_1, p_2, \ldots, p_{i-1}\})$, for $1 \leq i \leq n$.

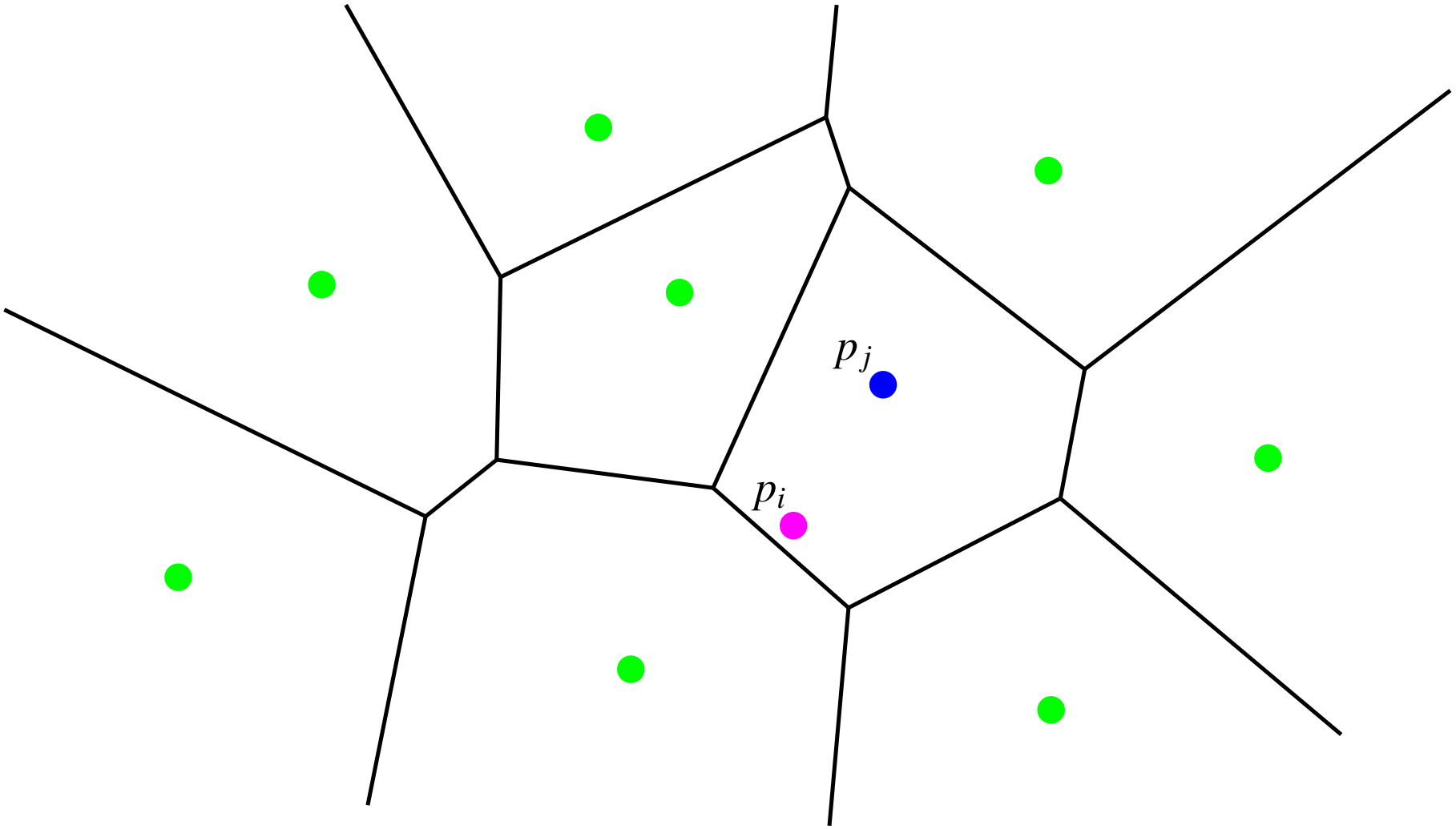
- If we could achieve constant complexity per insertion then a linear algorithm would result:
  $\longrightarrow$ Best case: $O(n)$.

- An insertion could, however, affect all other sites:
  $\longrightarrow$ Worst case: $O(n^2)$, or even worse.

- Since, on average, every Voronoi region is bounded by six Voronoi edges there is reason to hope that a close-to-linear time complexity can be achieved.

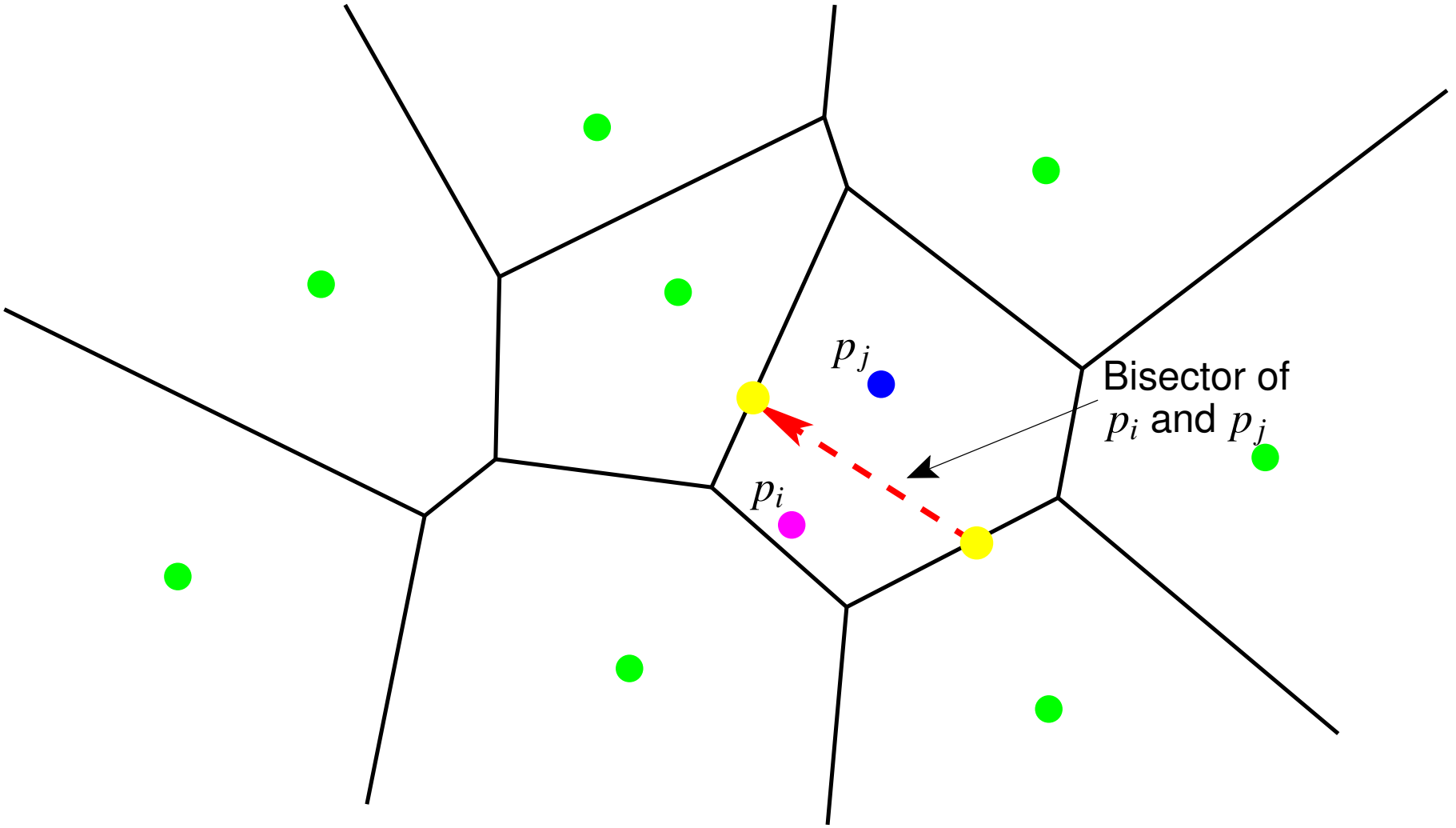
Incremental Construction: Basic Algorithm

1. Nearest-neighbor search among $\{p_1, p_2, \ldots, p_{i-1}\}$: Determine $1 \leq j < i$ such that the new point p_i lies in $\mathcal{VR}(p_j)$.



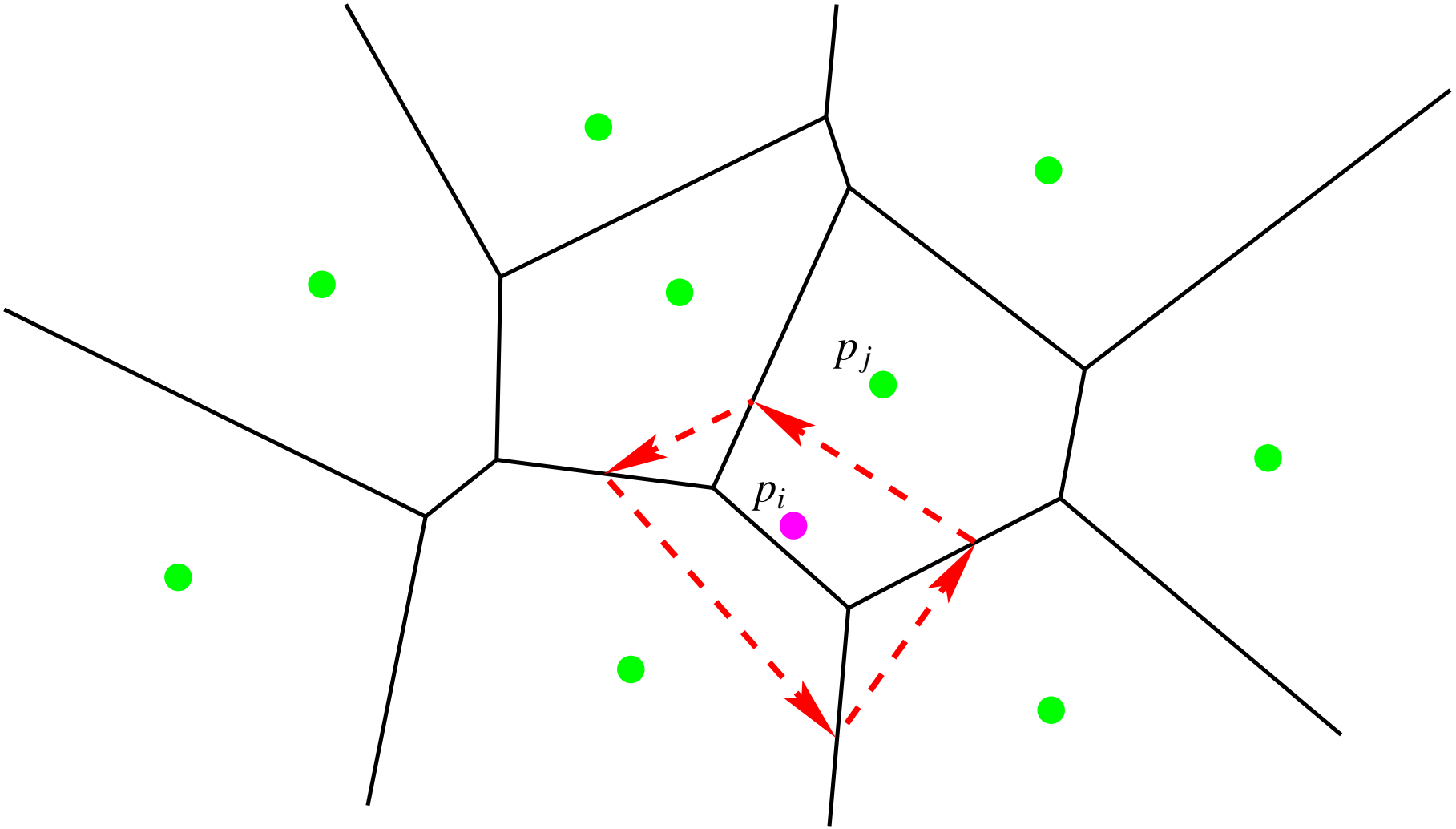
Incremental Construction: Basic Algorithm (cont'd)

2. Construct the bisector $b(p_i, p_j)$ between p_i and p_j, intersect it with $\mathcal{VP}(p_j)$, and clip that portion of $\mathcal{VP}(p_j)$ which is closer to p_i than to p_j.



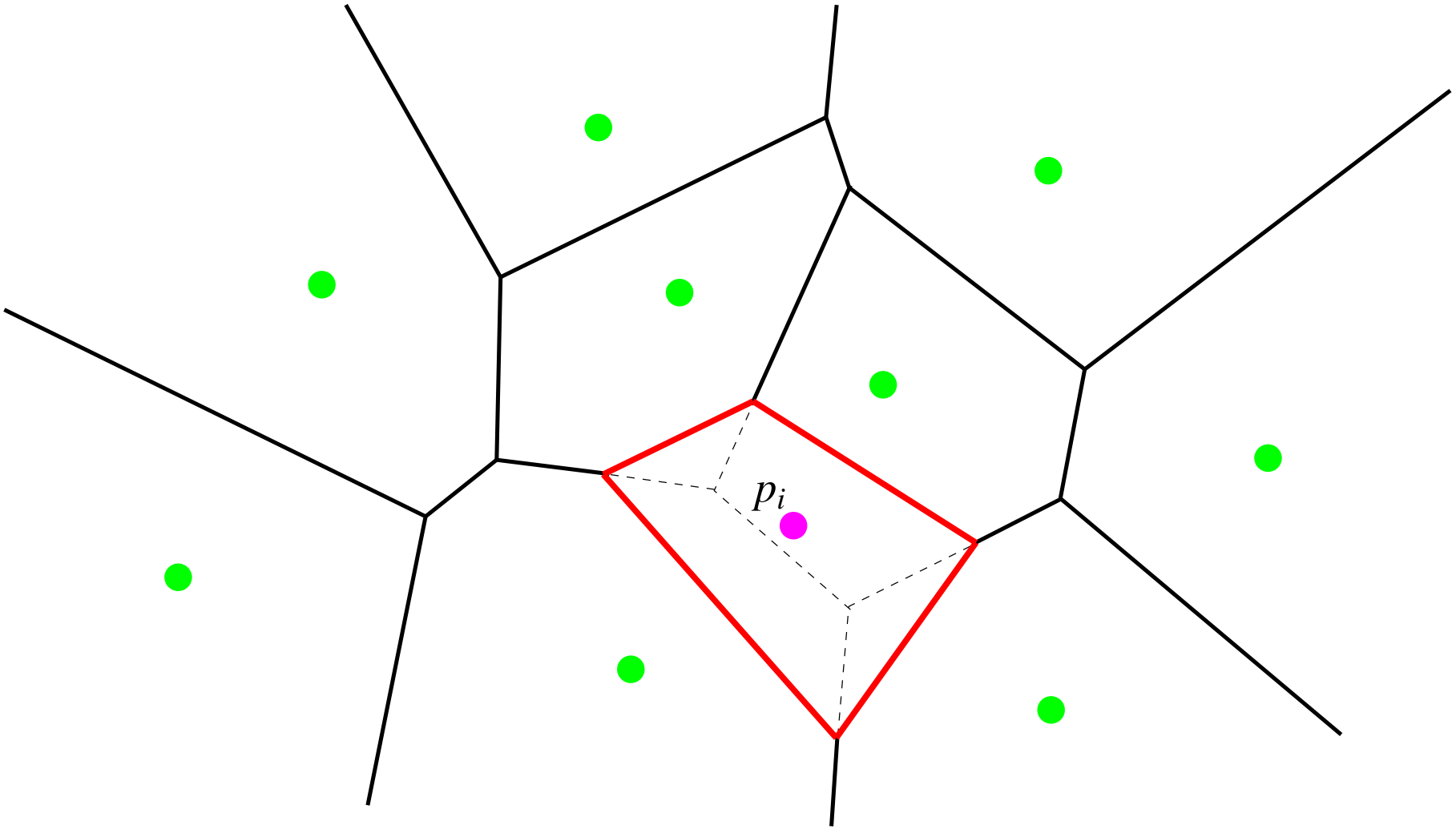
Incremental Construction: Basic Algorithm (cont'd)

3. Generate $\mathcal{VP}(p_i)$ by a circular scan around p_i, similar to the construction of the dividing chain in the divide&conquer algorithm.



Incremental Construction: Basic Algorithm (cont'd)

- The scan is finished once it returns to $\mathcal{VR}(p_j)$. What is the complexity of one insertion?



Incremental Construction: Complexity of Nearest-Neighbor Search

- The complexity mainly depends on the complexity of the nearest-neighbor search and on the number of edges generated/deleted during the scan.

- Nearest-neighbor search by brute force:
  - ⋆ Compute all possible distances.
  - ⋆ Not very practical: $O(n)$ per nearest-neighbor query.

- Nearest-neighbor search akin to “steepest ascent”:
  - ⋆ Guess an initial candidate for p_j.
  - ⋆ Compare $d(p_i, p_j)$ to the distances between p_i and the neighbors of p_j.
  - ⋆ Select the closest neighbor point as new candidate p_j.
  - ⋆ Continue with the new point, until all neighbors have a distance from p_i that is larger than $d(p_i, p_j)$.
  - ⋆ This approach heavily depends on the availability of a good initial guess.

Incremental Construction: Complexity of Nearest-Neighbor Search (cont'd)

- Nearest-neighbor search based on *geometric hashing*:
  - ⋆ Register p_1, p_2, p_{i-1} in a uniform grid.
  - ⋆ Locate the grid cell that contains p_i, and find nearest neighbor.
  - ⋆ This approach works best for a uniform distribution of the points.

- Nearest-neighbor search based on a *history DAG* and *randomized incremental insertion*:
  - ⋆ Assume that the points are inserted in random order.
  - ⋆ One can prove (using backwards analysis): the history DAG has $O(n)$ expected size, and supports nearest-neighbor queries in $O(\log n)$ expected time.
  - ⋆ See the slides on triangulation for more on randomized incremental construction.

Incremental Construction: Complexity of Incremental Merge

- Randomized incremental insertion:
  - ⋆ Again assume that the points are inserted in random order.
  - ⋆ One can prove (using backwards analysis): the update necessary for inserting one point can be done in $O(1)$ expected time.
  - ⋆ Note: This is independent of the point distribution, as long as the order is random!
- Worst-case insertion order:
  - ⋆ It is fairly easy to pick n points and number them "appropriately" such that the insertion of the i-th point requires the generation of $O(i)$ Voronoi edges!

Incremental Construction: Overall Complexity

- Overall cost for computing the Voronoi diagram incrementally: we get $O(n \log n)$ expected time if randomized insertion is used, independent of the distribution of the points.

- If points are uniformly distributed then geometric hashing tends to answer a nearest-neighbor query in $O(1)$ time.

- Thus, randomized incremental insertion of n uniformly distributed points can be assumed to generate the Voronoi diagram in linear or slightly super-linear time. (And this claim is supported by practical experiments.)

- However, the worst case still is $O(n^2)$, no matter how efficiently all nearest-neighbor queries are answered!

Geometric Hashing for Nearest-Neighbor Searching

- The bounding box of the input points is partitioned into rectangular cells of uniform size by means of a regular grid.

- For every cell c, all points of $\{p_1, p_2, \ldots, p_{i-1}\}$ that lie in c are stored with c. (Alternatively, only one point is stored per cell.)

- To find the point p_j nearest to point p_i:
  - ⋆ Determine the cell c that in which p_i lies.
  - ⋆ By searching in c (and possibly in its neighboring cells, if c is empty), we find a first candidate for the nearest neighbor.
  - ⋆ Let δ be the distance from p_i to this point.
  - ⋆ We continue searching in c and in those cells around c which are intersected by a clearance circle with radius δ centered at p_i.
  - ⋆ Whenever a point that is closer is found, we update δ appropriately.
  - ⋆ The search stops once no unsearched cell exists that is intersected by the clearance circle.

Geometric Hashing for Nearest-Neighbor Searching (cont'd)

- If all points are stored per cell then the true nearest neighbor is found. If only one point is stored per cell then this approach yields a (hopefully) good initial candidate for the nearest neighbor.

- What is a suitable resolution of the grid? There is no universally valid answer. In any case, the grid should not use more than $O(n)$ memory!

- Personal experience:
  - ⋆ Grids of the form $(w \cdot \sqrt{n}) \times (h \cdot \sqrt{n})$ seem to work nicely, with $w \cdot h = c$ for some constant c.
  - ⋆ The parameters w, h are chosen to adapt the resolution of the grid to the aspect ratio of the bounding box of the points.
  - ⋆ By experiment: $1 \leq c \leq 2$.

- This basic scheme can be tuned considerably, e.g., by switching to 2D-trees if a small sample of the points indicates that the points are distributed highly non-uniformly.

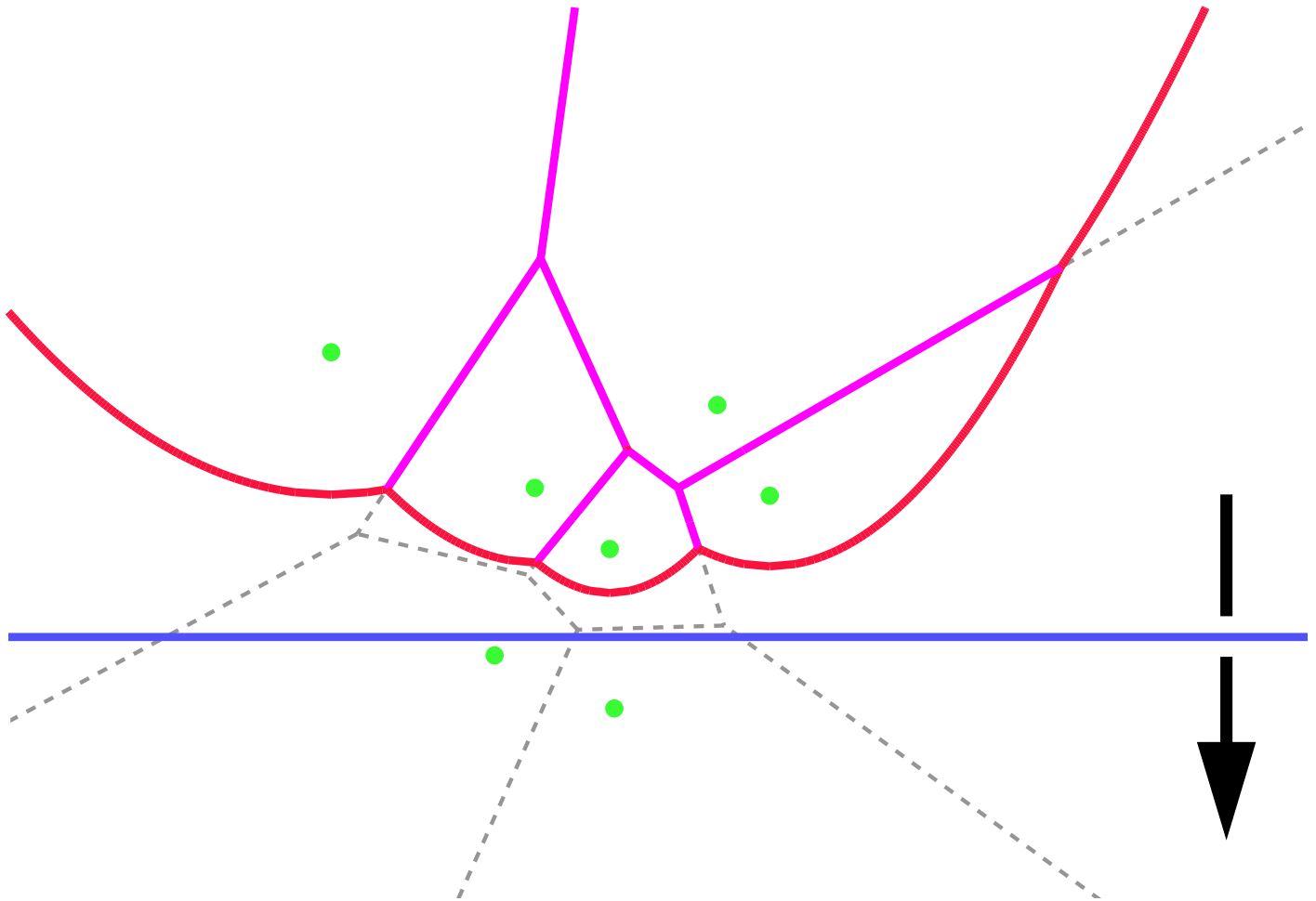
- Note: grid-based nearest-neighbor searching will work best for points that are distributed uniformly, and will fail miserably if all points end up in one cell!

Sweep-Line Algorithm

- Can a sweep-line algorithm be applied to compute the Voronoi diagram?
- Principal problem: when the sweep line reaches an extreme (e.g., top-most) vertex of $\mathcal{VP}(p_i)$, it has not yet moved over p_i.
- Thus, the information on the point sites is missing when a Voronoi polygon is first encountered and Voronoi nodes are to be computed.
- This problem is independent of the sweep direction chosen. Thus, w.l.o.g., we move the sweep line ℓ from top to bottom.
- Remarkable idea (by S. Fortune): rather than keeping the actual intersection of the Voronoi diagram with ℓ, we maintain information on that part of the Voronoi diagram of the points above ℓ that is not affected by points below ℓ.
- That part of the Voronoi diagram under construction lies above a *beach line* consisting of parabolic arcs: each parabolic arc is defined by ℓ and a point above ℓ.

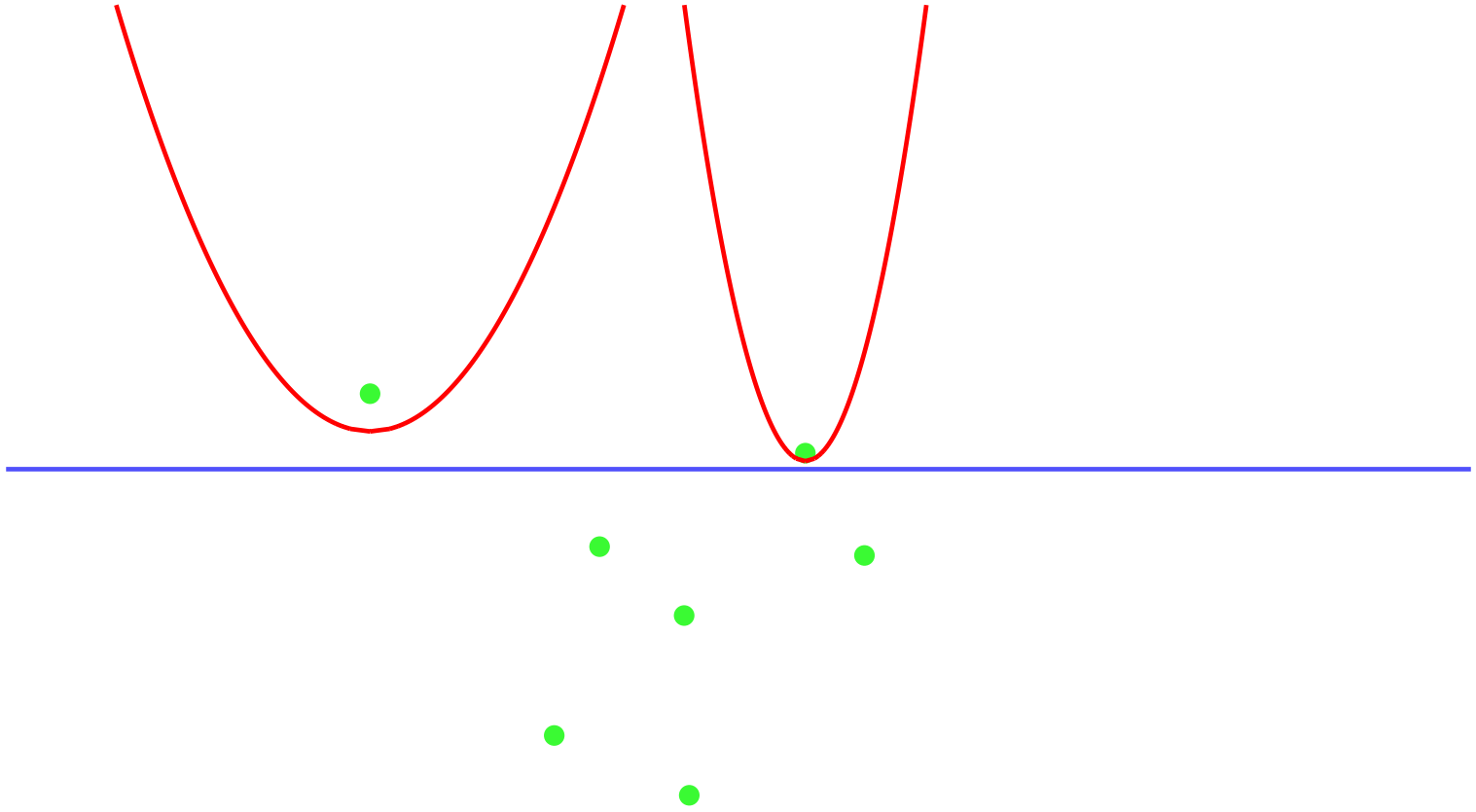
Sweep-Line Algorithm: Beach Line

- The part of the Voronoi diagram that will not change any more as the sweep line continues to move downwards lies above the beach line formed by the lower envelope of parabolic arcs.



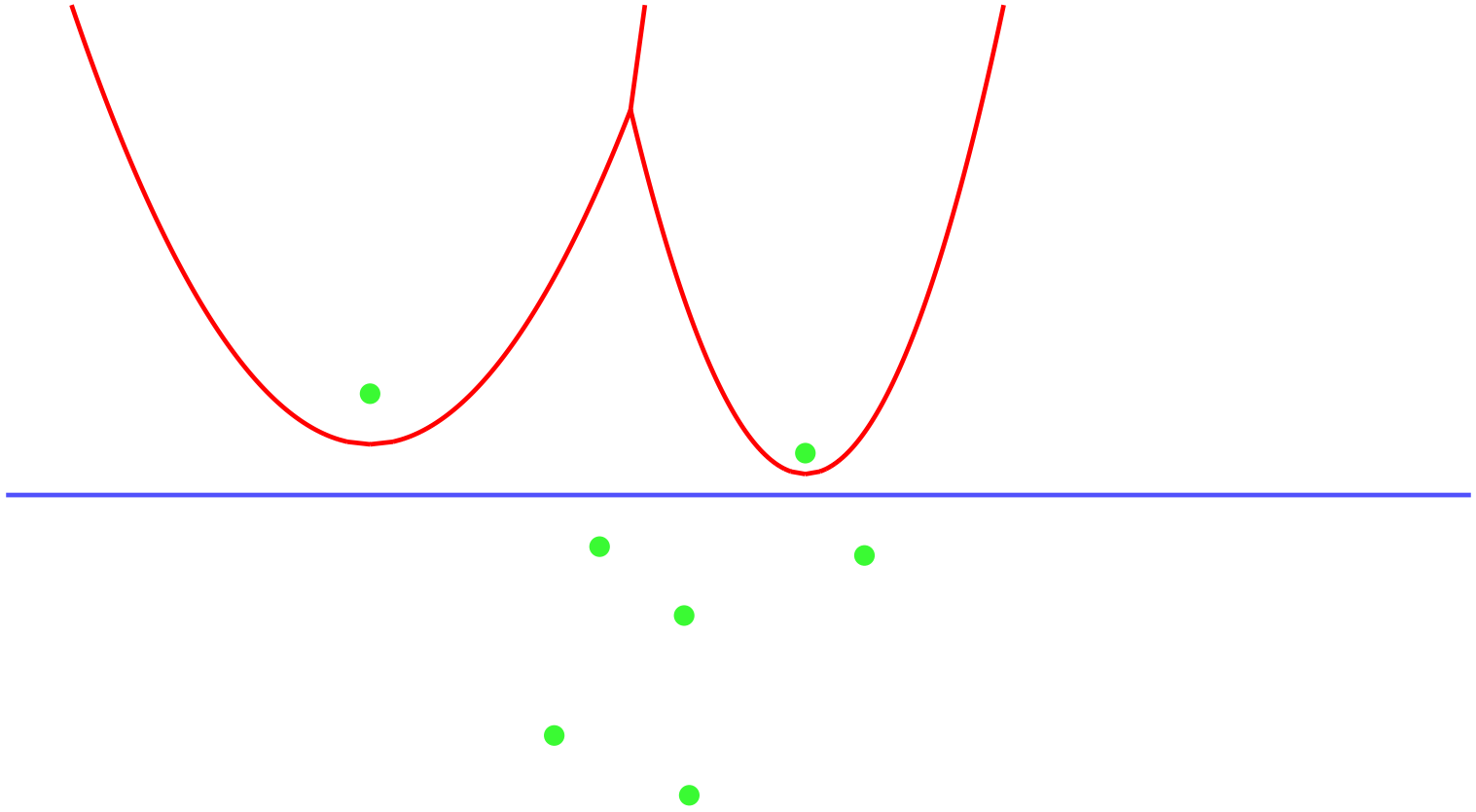
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



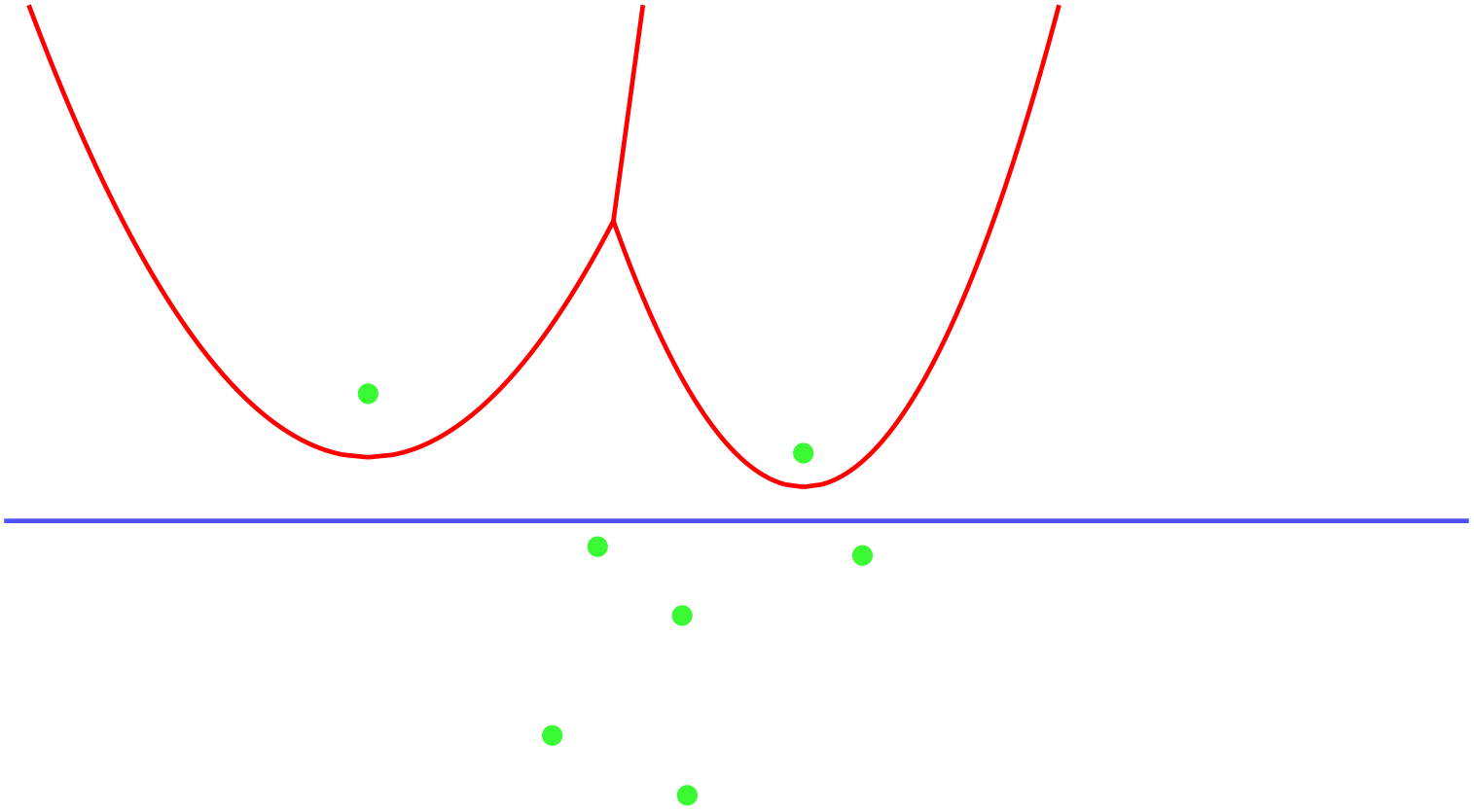
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



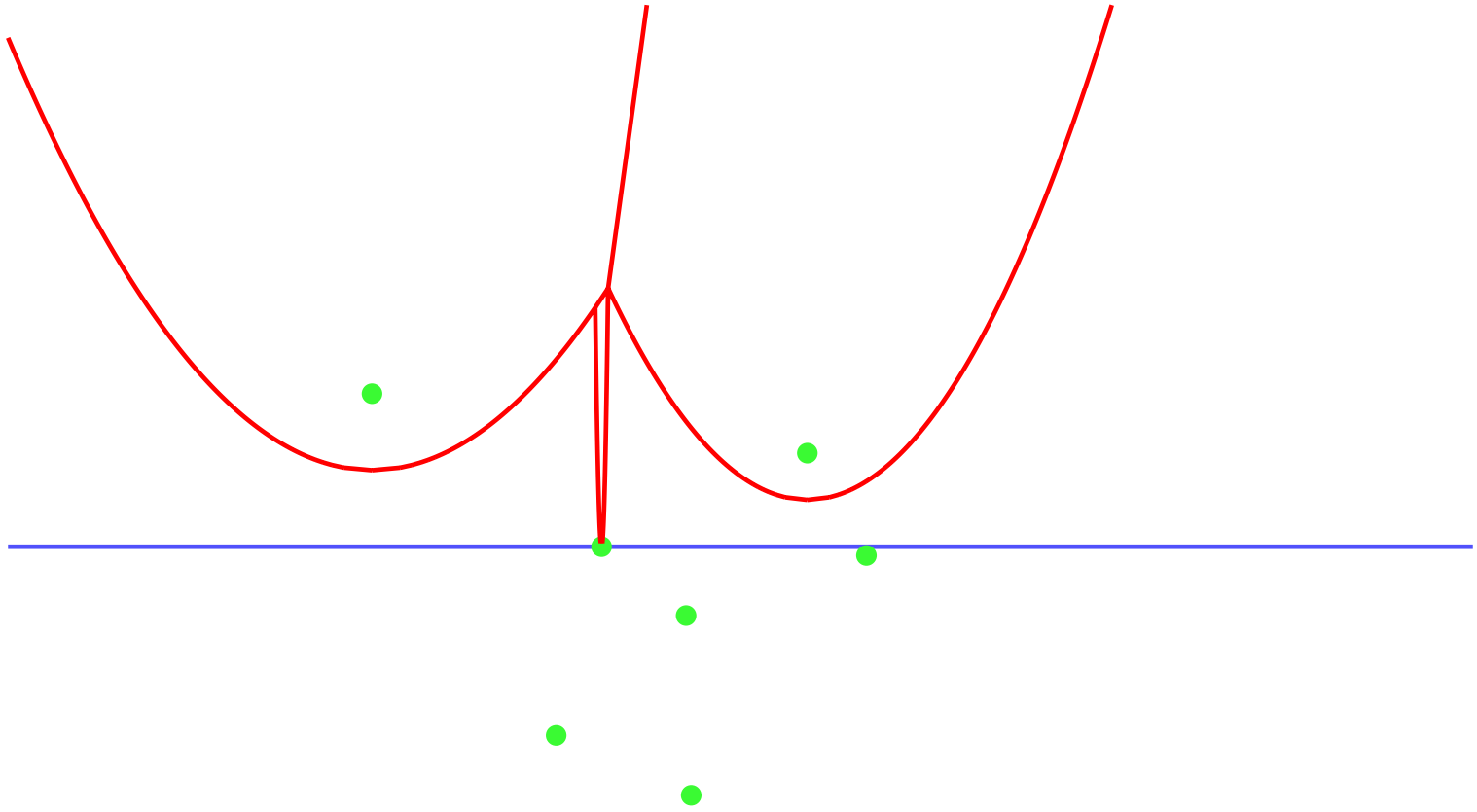
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



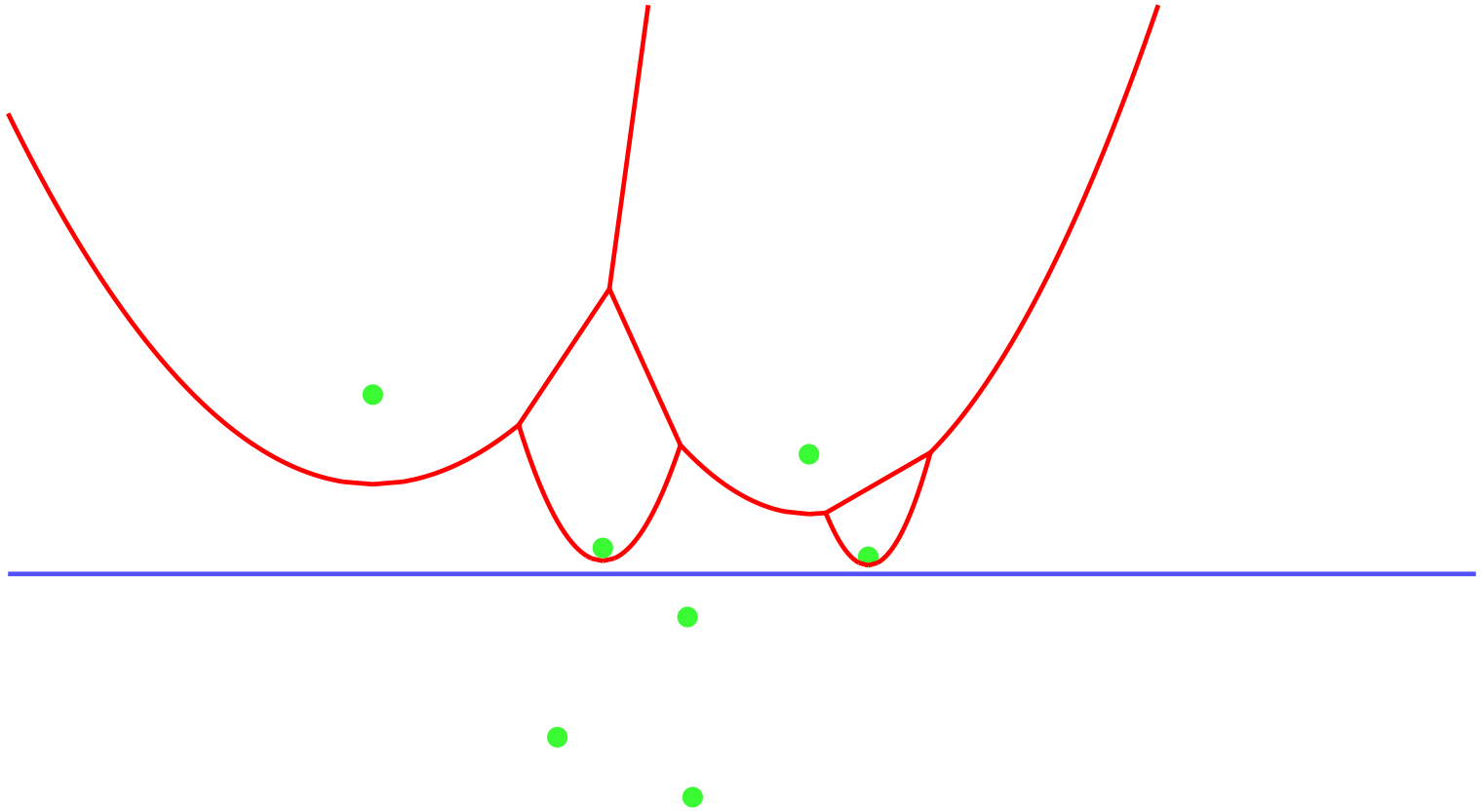
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



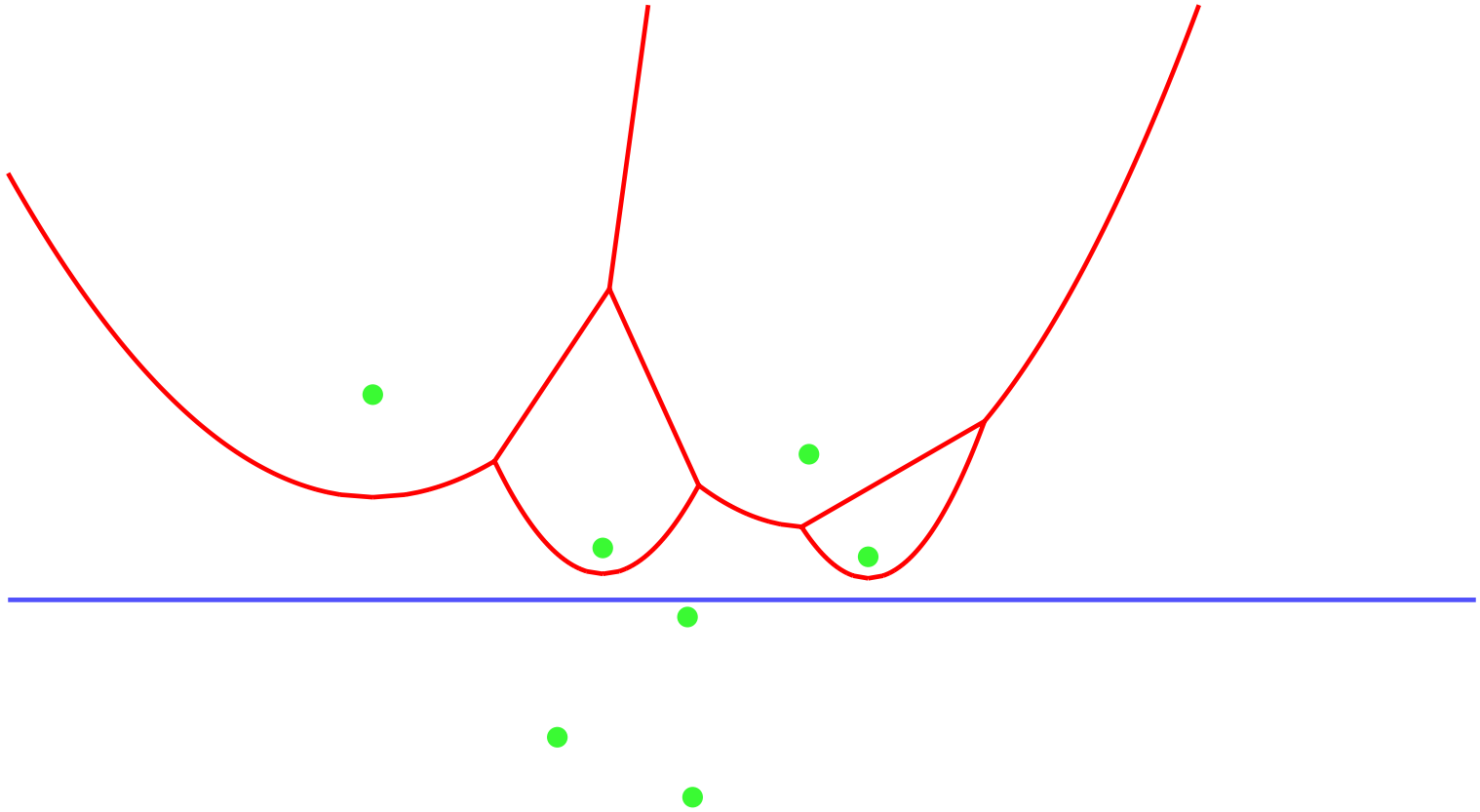
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



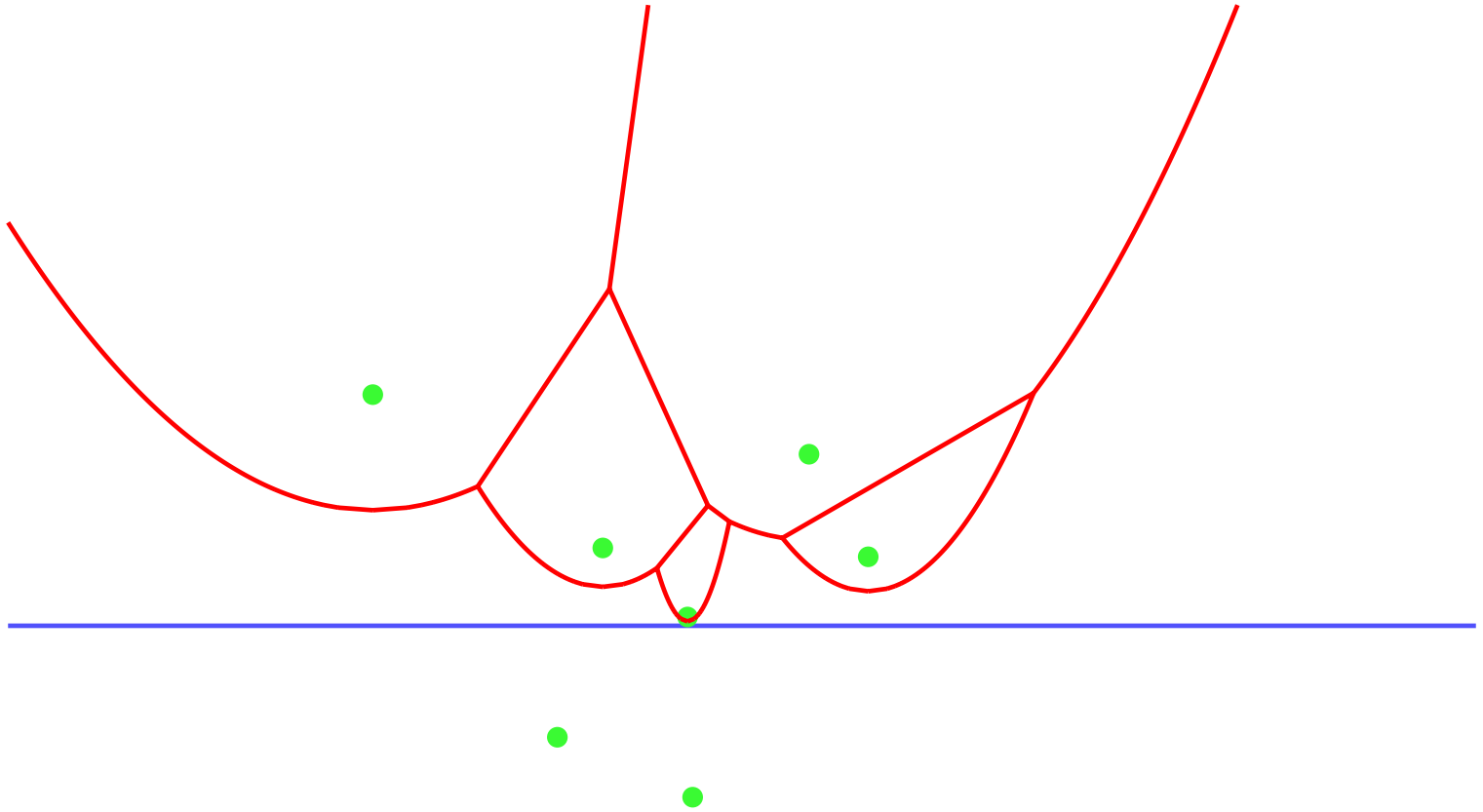
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



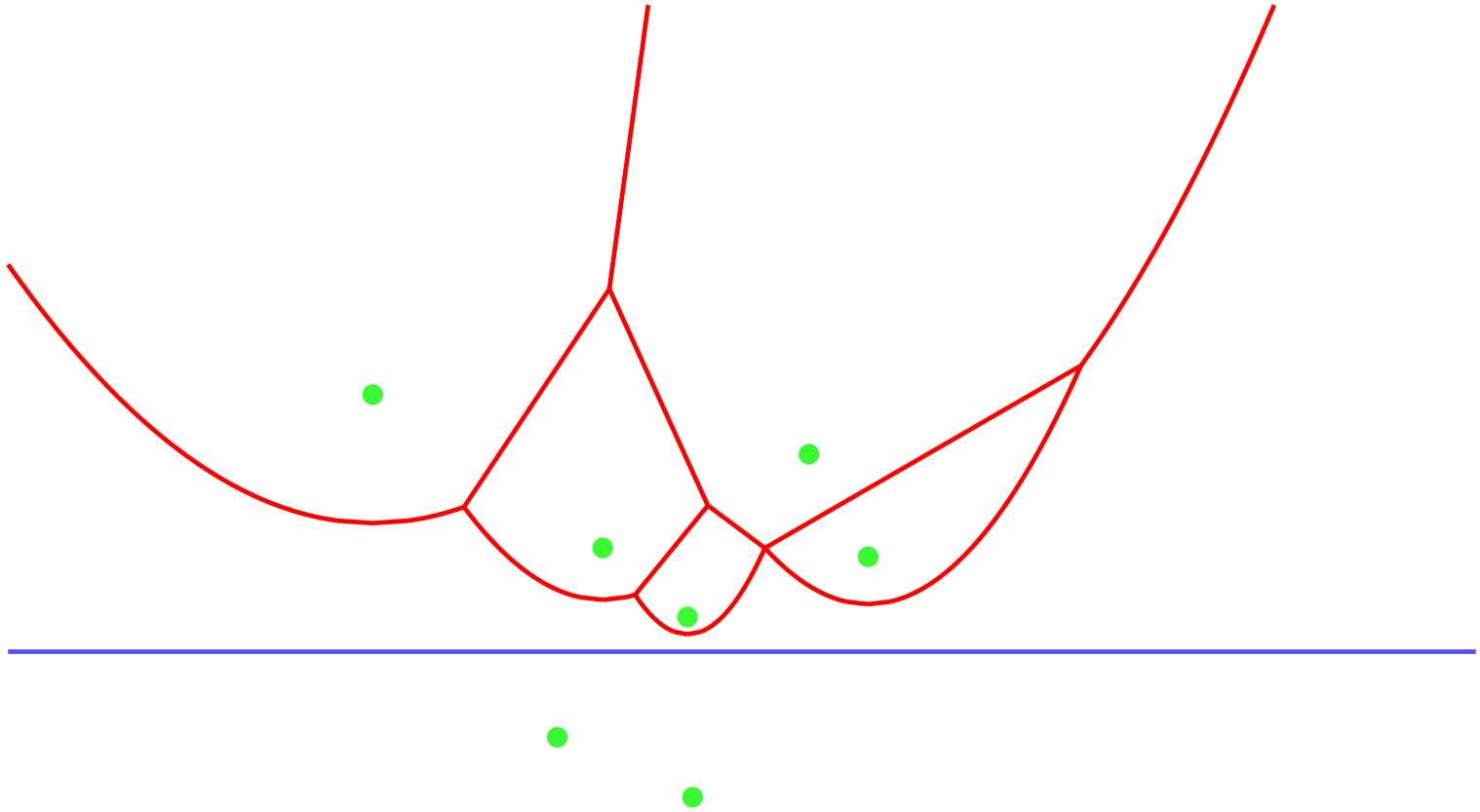
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



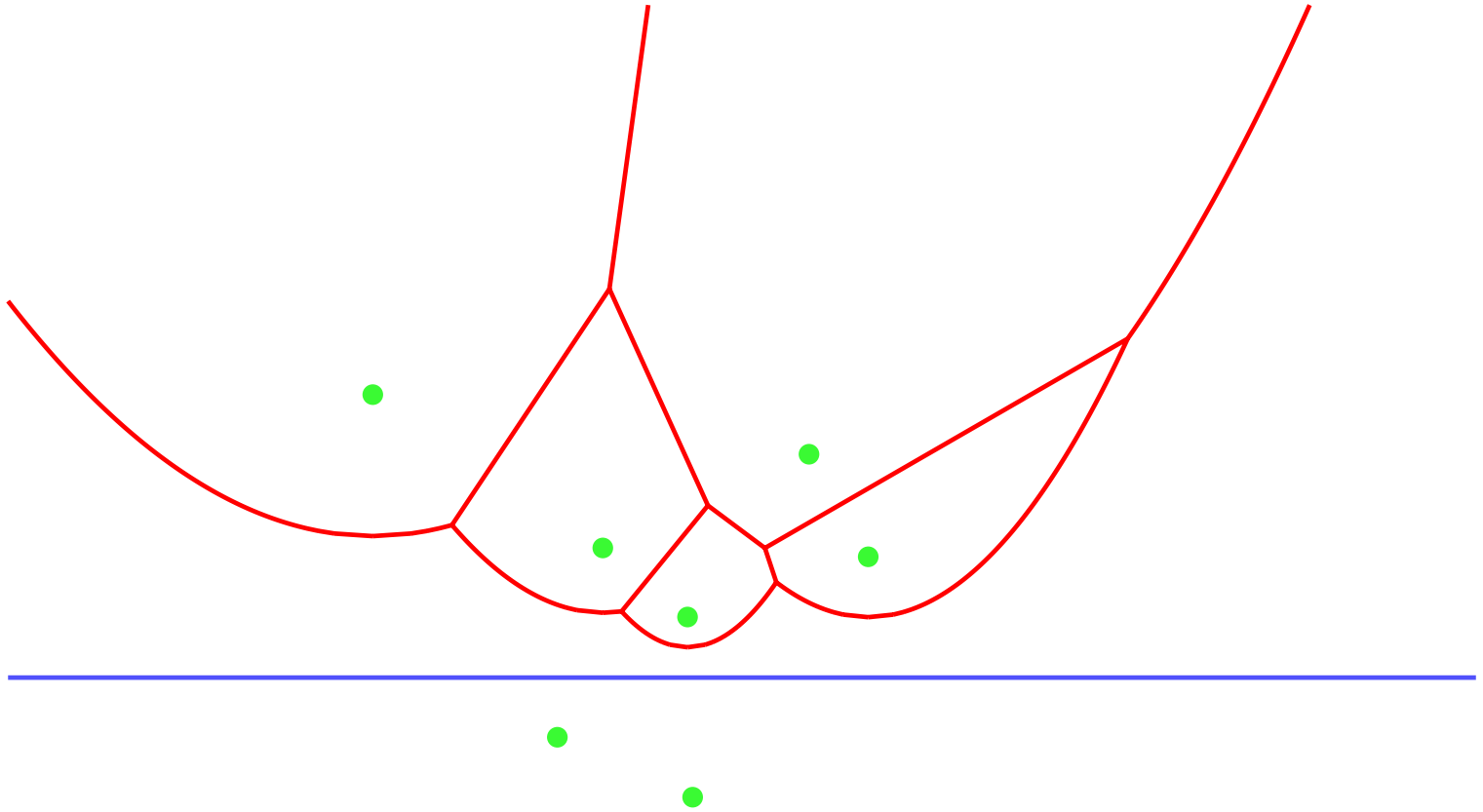
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



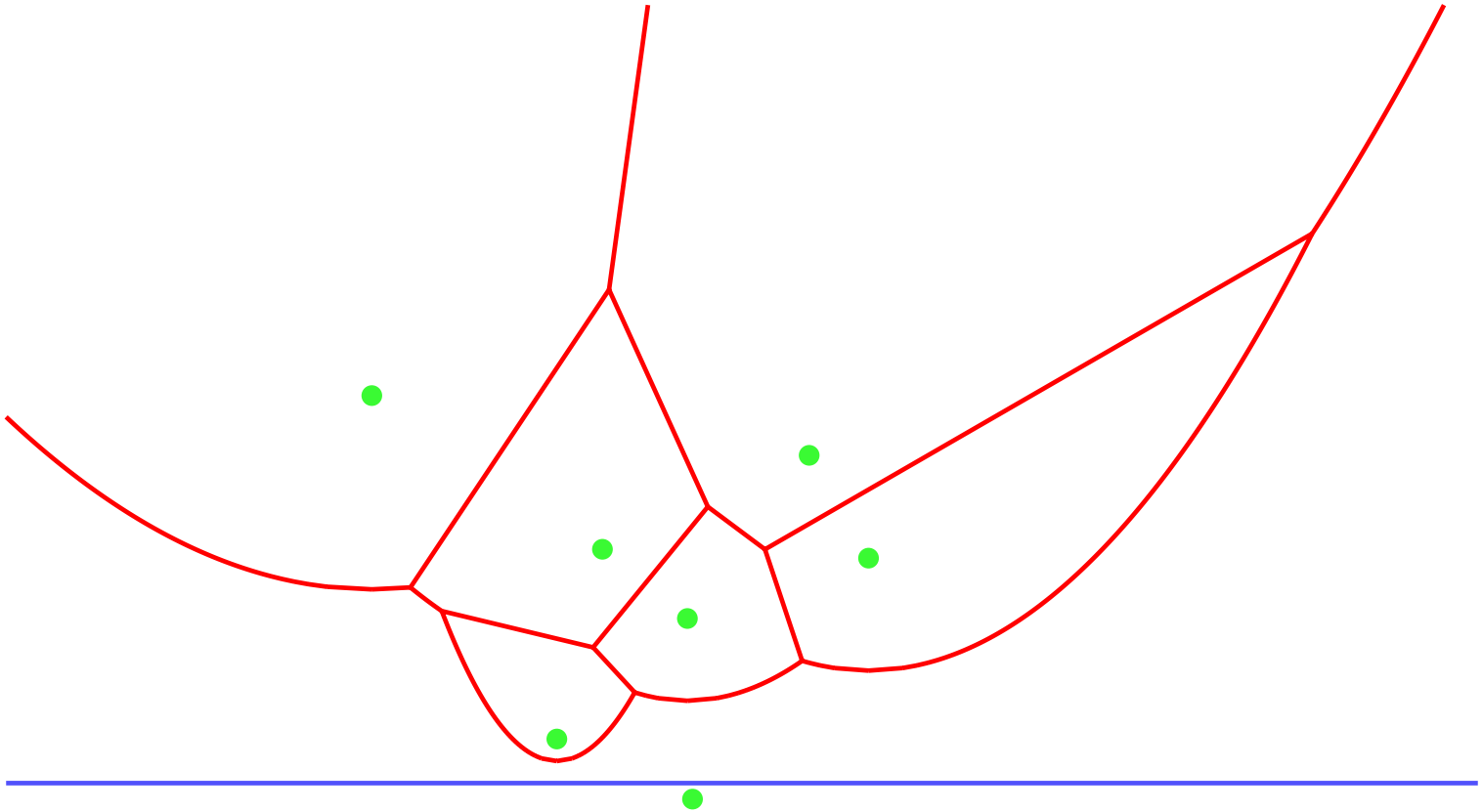
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



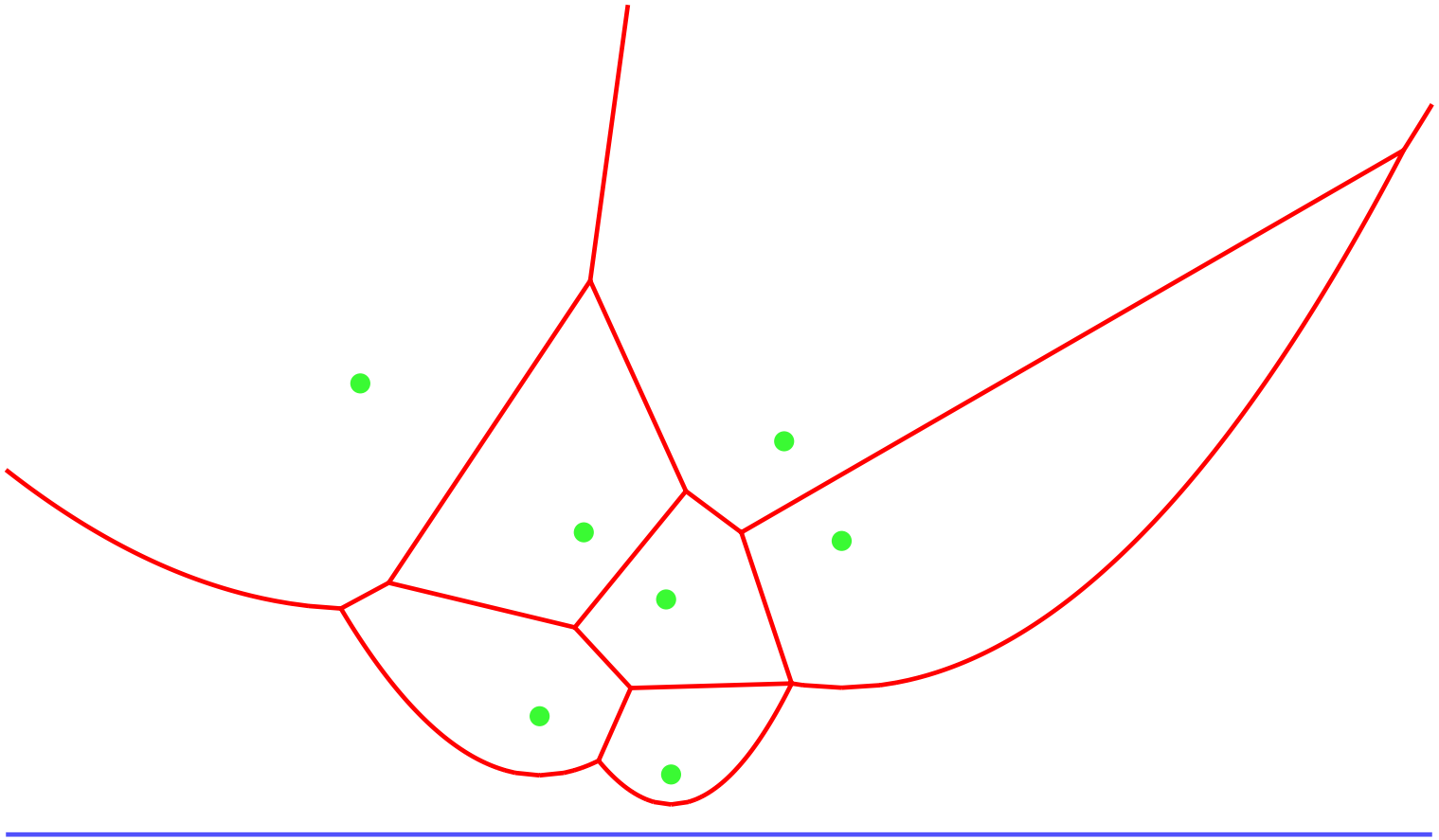
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



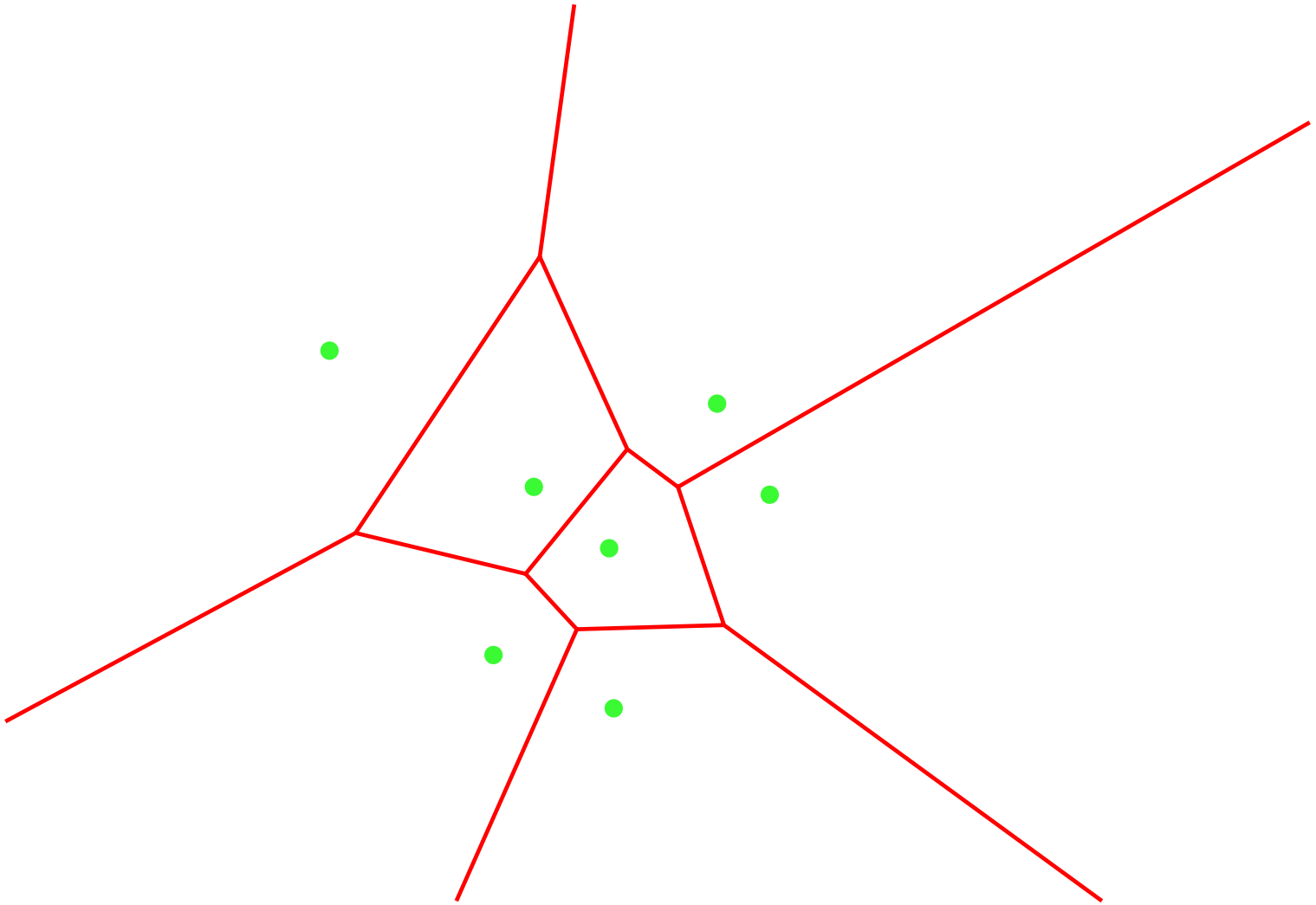
Animation of Sweep-Line Algorithm

- The beach line moves downwards as the sweep-line is moved from top to bottom.



Animation of Sweep-Line Algorithm

- A full sweep reveals the complete Voronoi diagram.



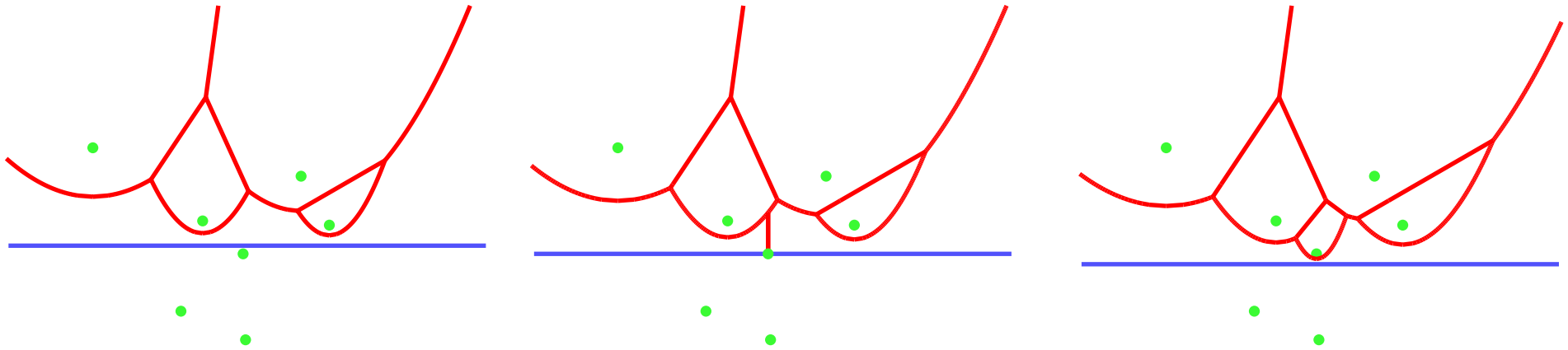
Sweep-Line Algorithm: Events

- The following two events need to be considered for the event-point schedule:

  1. Site event:
     - ⋆ The sweep line ℓ passes through an input point, and a new parabolic arc needs to be inserted into the beach line.
  2. Circle event:
     - ⋆ A parabolic arc of the beach line vanishes, i.e., degenerates to a point v, and a new Voronoi node has to be inserted at v.
     - ⋆ What does this mean for the sweep line ℓ? What is the appropriate y-position of ℓ to catch this event?

Sweep-Line Algorithm: Site Event

- If the sweep line ℓ passes through an input point then a new parabolic arc needs to be inserted into the beach line. Initially, this arc is degenerate.

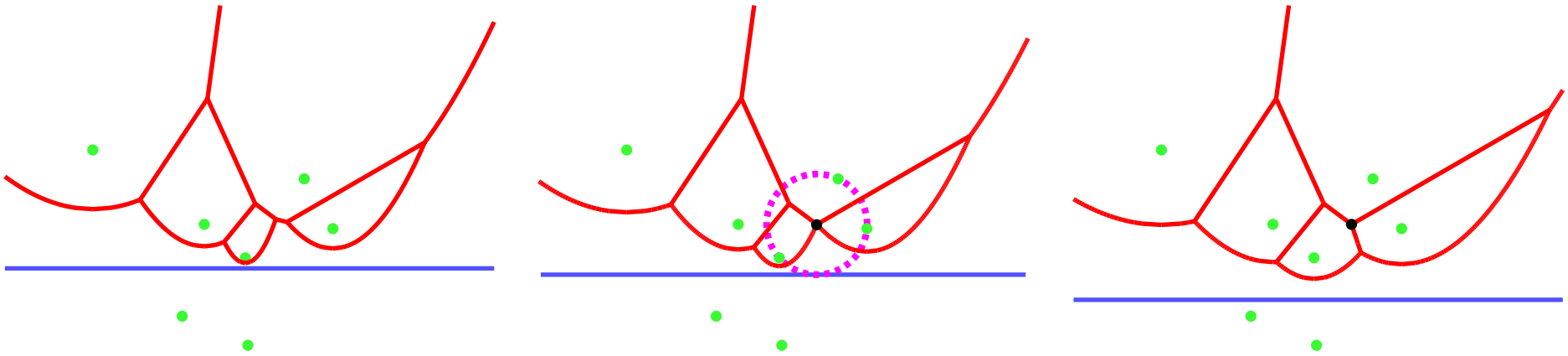


- This event occurs whenever the sweep line ℓ passes through an input point p_i.

- It is responsible for the initialization of a new Voronoi region that will become $\mathcal{VR}(p_i)$.

Sweep-Line Algorithm: Circle Event

- If a parabolic arc of the beach line degenerates to a point v then a new Voronoi node needs to be inserted at v.

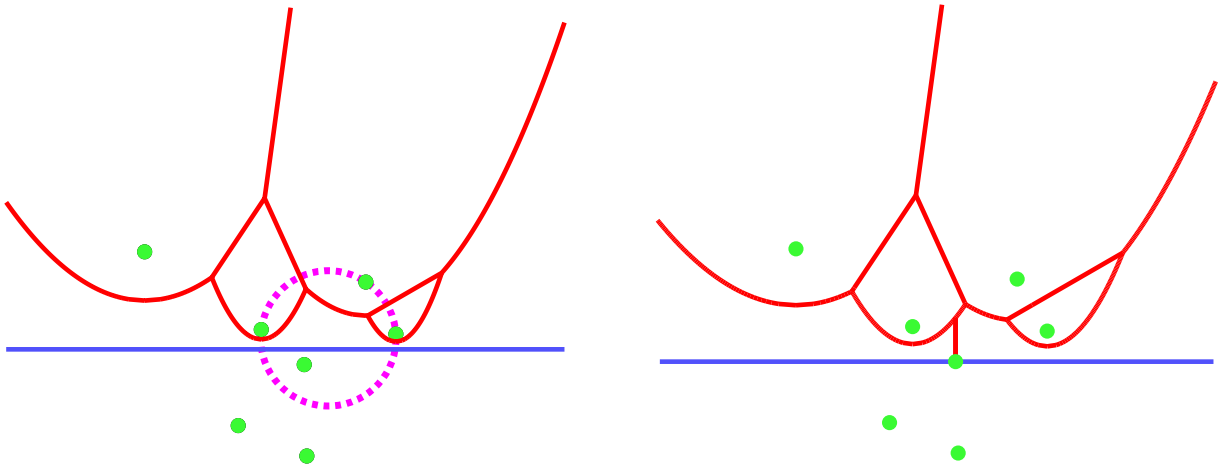


- A circle event occurs when the sweep line ℓ passes over the south pole of a circle through the three defining input points p_i, p_j, p_k of three consecutive parabolic arcs of the beach line.

- The center v of such a circle is equidistant to p_i, p_j, p_k and also to ℓ; it becomes a new node of the Voronoi diagram.

Sweep-Line Algorithm: False Alarms

- Not all scheduled circle events correspond to valid new Voronoi nodes: a circle event has to be processed only if its defining three parabolic arcs still are consecutive members of the beach line at the time when the sweep line ℓ passes over the south pole of the circle.



Sweep-Line Algorithm: Event-Point Schedule and Sweep-Line Status

- All input points are stored in sorted order (according to y-coordinates) in the event-point schedule.
- Whenever three parabolic arcs become consecutive for the first time – when a site event occurs – the y-coordinate of the corresponding circle event is inserted into the event-point schedule at the appropriate place.
- Lemma: The beach line is monotone with respect to the x-axis.
- Parabolic arcs have to be inserted into the beach line when processing site events, and have to be deleted when processing circle events.
- Both structures are best represented as balanced binary search trees, since this allows logarithmic insertion/deletion.

Sweep-Line Algorithm: Analysis

- Lemma: An arc can appear on the beach line only through a site event.
- Corollary: The beach line is a sequence of at most $2n - 1$ parabolic arcs.
- Lemma: An arc can disappear from the beach line only through a circle event.
- Theorem: The sweep-line algorithm computes the Voronoi diagram of n points in $O(n \log n)$ time, using $O(n)$ storage.

Construction via Lifting to 3D

- Consider the transformation that maps a point $p = (p_x, p_y)$ to the non-vertical plane $h(p) \equiv z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$ in $\mathbb{R}^3$.

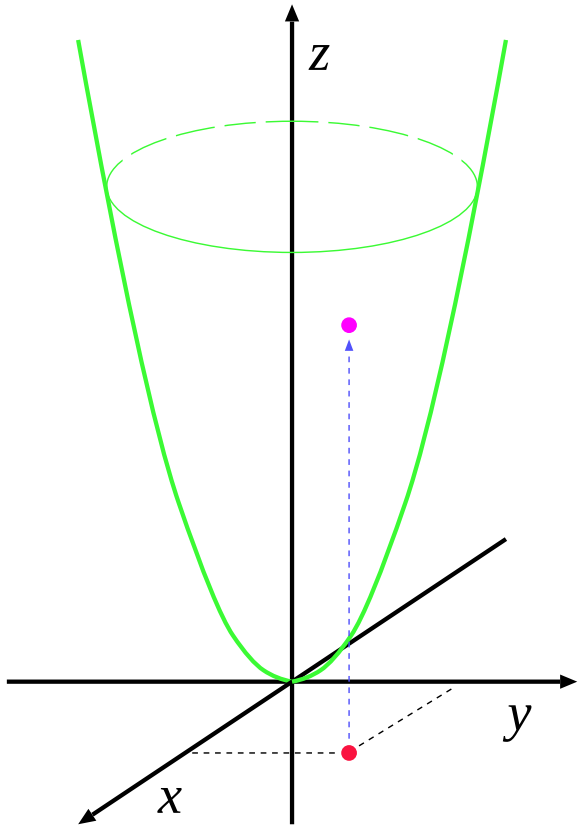
- This plane is tangent to the unit paraboloid $z = x^2 + y^2$ at the point $(p_x, p_y, p_x^2 + p_y^2)$.

- Let $h^+(p)$ be the half-space defined by $h(p)$ which contains the unit paraboloid.

- For $S := \{p_1, p_2, \ldots, p_n\}$, consider the convex polyhedron $\mathcal{P} := \cap_{1 \le i \le n} h^+(p_i)$.

- Lemma: The normal projection of the vertices and edges of $\mathcal{P}$ onto the xy-plane yields $\mathcal{VD}(S)$.

- Since $\mathcal{P}$ can be obtained in $O(n \log n)$ time, we have yet another $O(n \log n)$ algorithm for computing Voronoi diagrams.

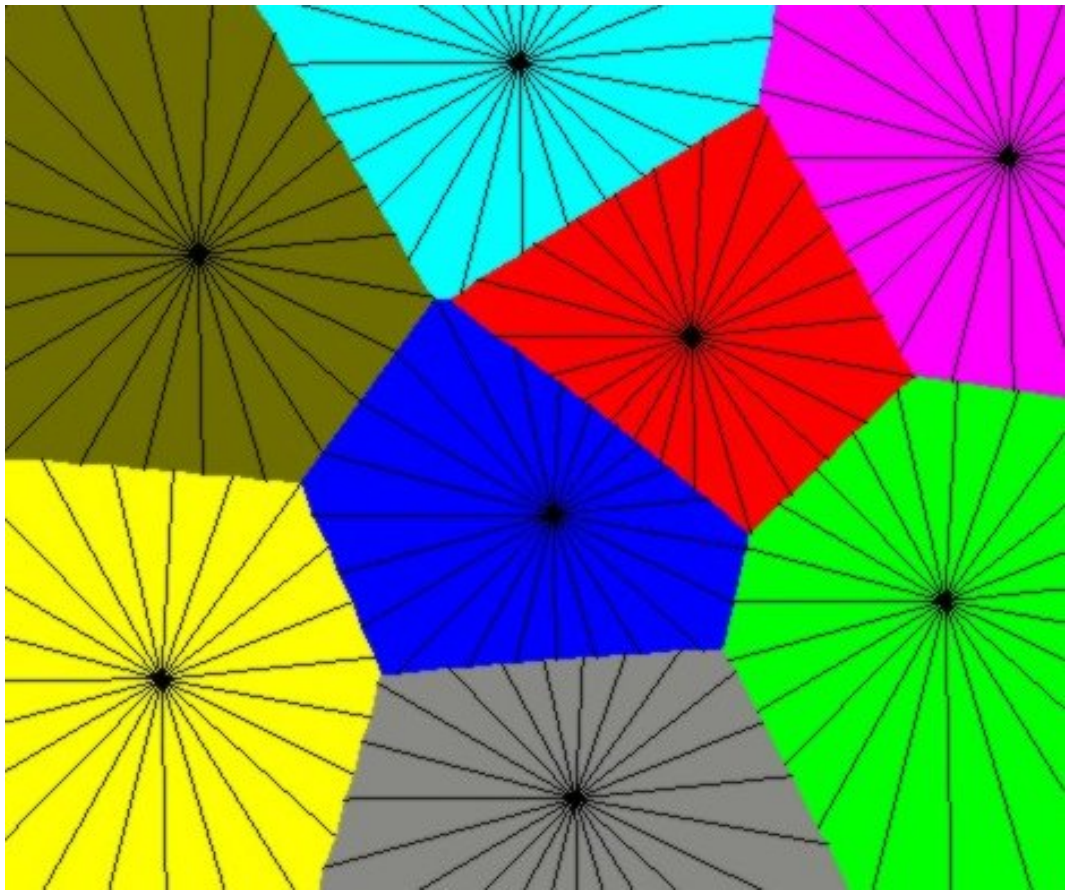


Approximate Voronoi Diagram by Means of Graphics Hardware

- Regard $\mathbb{R}^2$ as the xy-plane of $\mathbb{R}^3$, and construct upright circular unit cones at every point of S. (All cones point upwards, are of the same size and form the same angle with the xy-plane!) Assign a unique color to every cone.

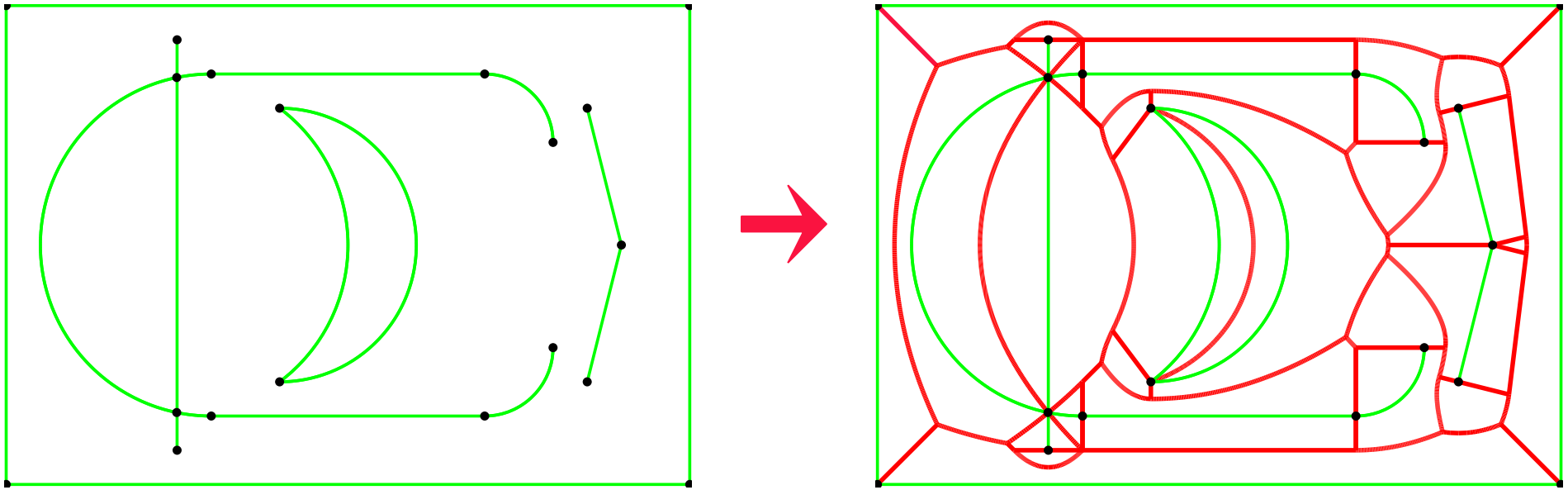
Approximate Voronoi Diagram by Means of Graphics Hardware

- Look at the cones from below the xy-plane, and use normal projection to project them on the xy-plane. This yields a colored subdivision of the xy-plane, i.e., of $\mathbb{R}^2$, where each cell corresponds to a Voronoi region of a point of S.



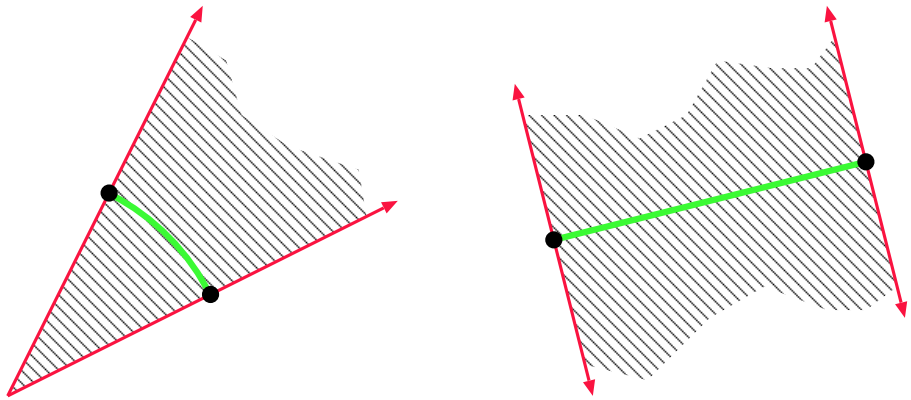
Generalized Voronoi Diagram

- We consider a set S of n "sites" (points, straight-line segments, and circular arcs).
- For technical reasons we assume that all end-points of all segments and arcs are members of S. Furthermore, the segments and arcs are allowed to intersect only at common end-points. Such a set of sites is called "*admissible*".
- Intuitively, the Voronoi diagram of S partitions the Euclidean plane into regions that are closer to one site than to any other.



Generalized Voronoi Diagram: Cone of Influence

- Technical problem: we need to avoid “two-dimensional” bisectors.
- Def.: The *cone of influence*, $CI(s)$, of
  - ⋆ a circular arc s is the closure of the cone bounded by the pair of rays originating in the arc’s center and extending through its endpoints;
  - ⋆ a straight-line segment s is the closure of the strip bounded by the normals through its endpoints;
  - ⋆ a point s is the entire plane.



Generalized Voronoi Diagram: Definitions

- Consider an admissible set S of n sites, and two sites $s_1, s_2 \in S$.

- Definitions:

  ⋆ The *bisector* $b(s_1, s_2)$ gives the loci of points that are equidistant to s_1 and s_2 and that belong to $CI(s_1) \cap CI(s_2)$.

  ⋆ The *Voronoi region* of s_i is defined as

$$\mathcal{VR}(s_i) := \{q \in CI(s_i) : d(s_i, q) \leq d(S \setminus \{s_i\}, q)\}.$$

  ⋆ The *(generalized) Voronoi polygon* of s_i is defined as

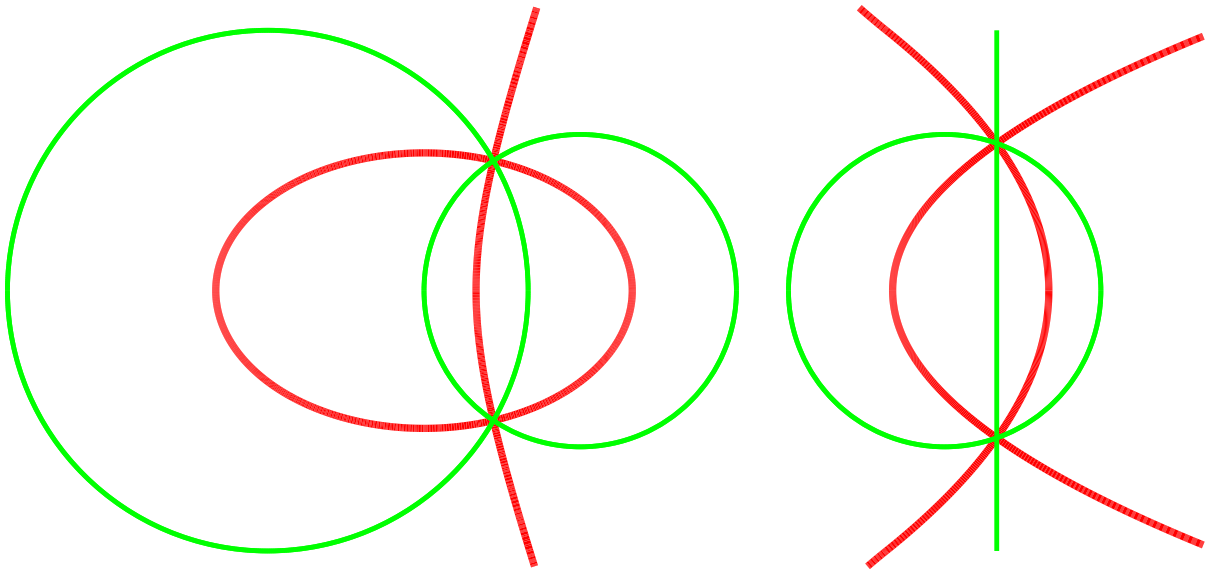
$$\mathcal{VP}(s_i) := \{q \in CI(s_i) : d(s_i, q) = d(S \setminus \{s_i\}, q)\}.$$

  ⋆ The *(generalized) Voronoi diagram* of S is defined as

$$\mathcal{VD}(S) := \bigcup_{1 \leq i \leq n} \mathcal{VP}(s_i).$$

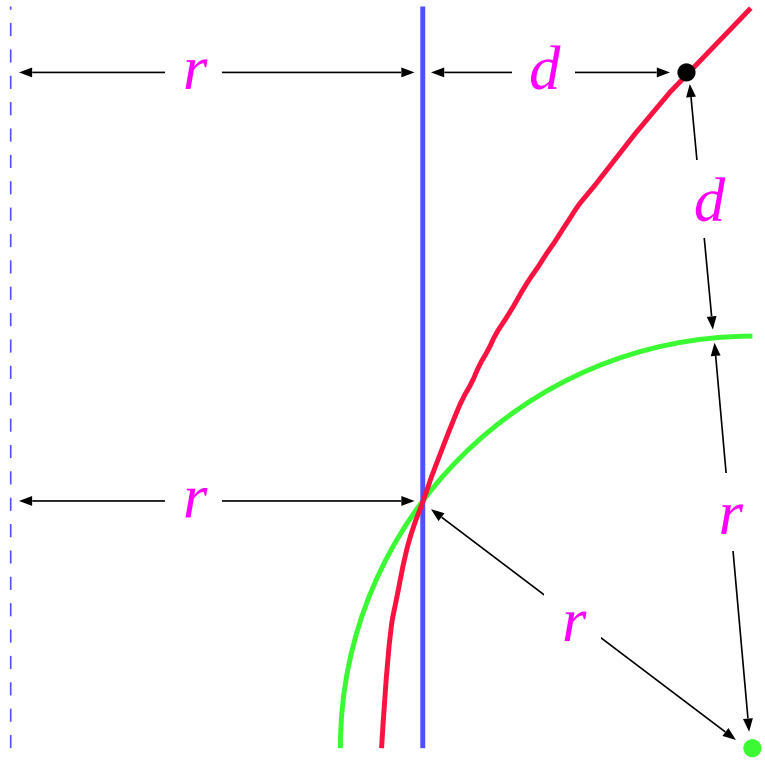
Generalized Voronoi Diagram: Bisectors

- Lemma: $\mathcal{VD}$(S) is a planar graph and consists of $O(n)$ parabolic, hyperbolic, elliptic and straight-line bisectors.



Generalized Voronoi Diagram: Bisectors (cont'd)

- Lemma: $\mathcal{VD}$(S) is a planar graph and consists of $O(n)$ parabolic, hyperbolic, elliptic and straight-line bisectors.

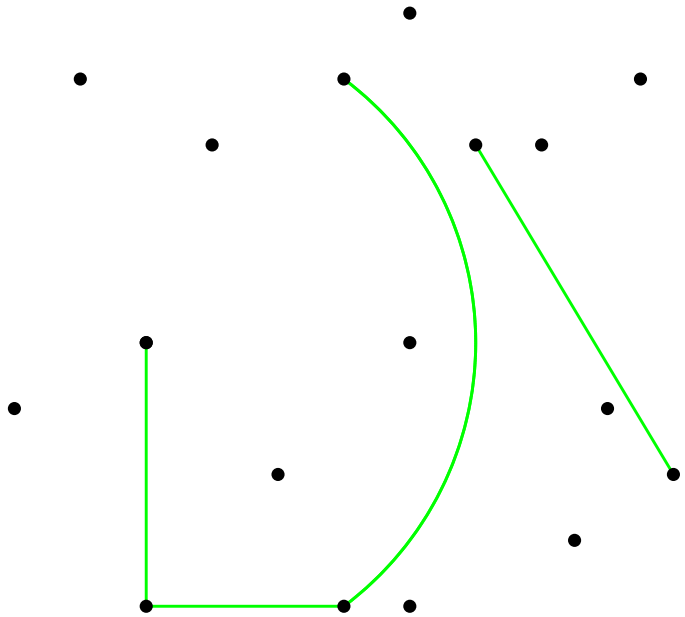


Generalized Voronoi Diagram: Algorithms

- Several $O(n \log n)$ expected-time algorithms for polygons and/or line segments.
- Sweep-line algorithm (for points and line segments): $O(n \log n)$ worst-case time [Fortune 1987].
- Divide&conquer algorithm: $O(n \log n)$ worst-case time [Yap 1987].
- Randomized incremental construction: $O(n \log n)$ expected time [Held&Huber 2008].
- Note: the Voronoi diagram of a (convex) polygon can be constructed in linear time [Aggarwal et al. 1989, Chin et al. 1999]!

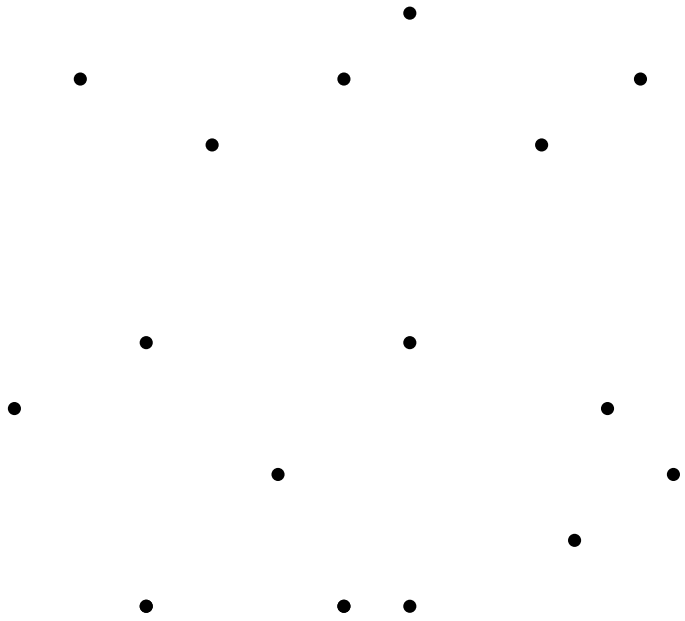
Generalized Voronoi Diagram: Randomized Incremental Construction

- How can we construct the (generalized) Voronoi diagram of the green sites?



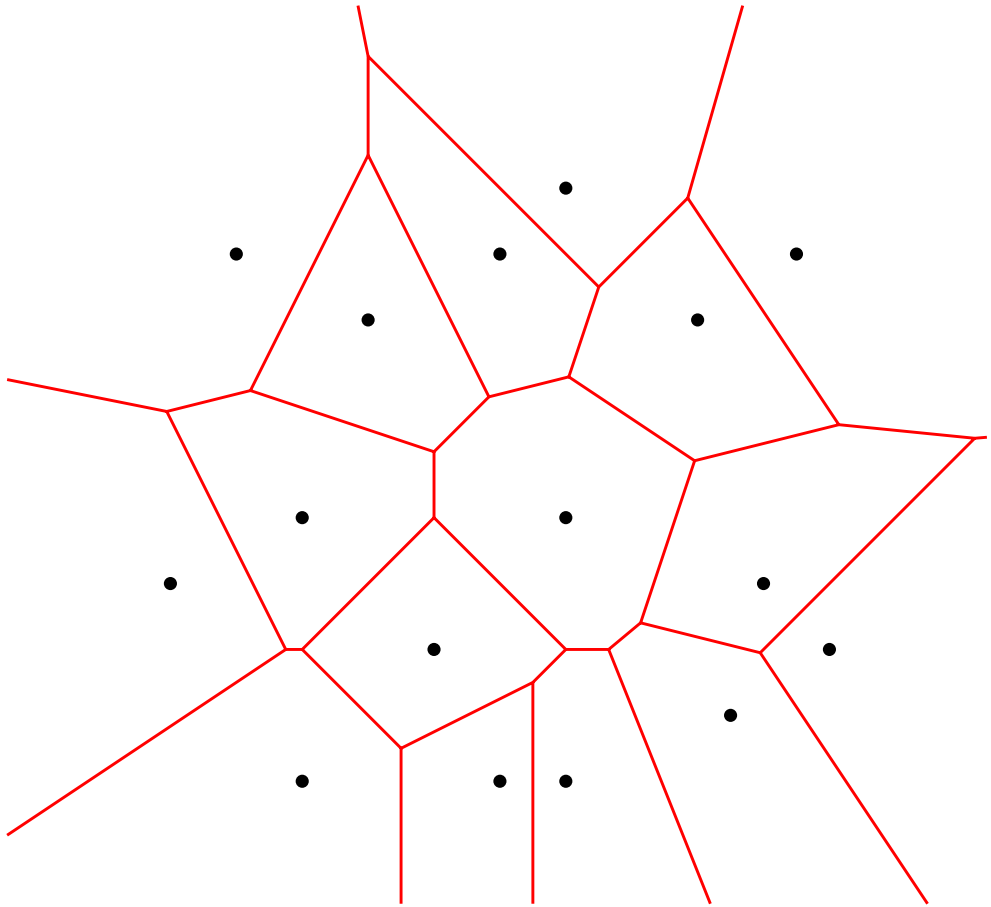
Generalized Voronoi Diagram: Randomized Incremental Construction

- Start with the vertices of S.



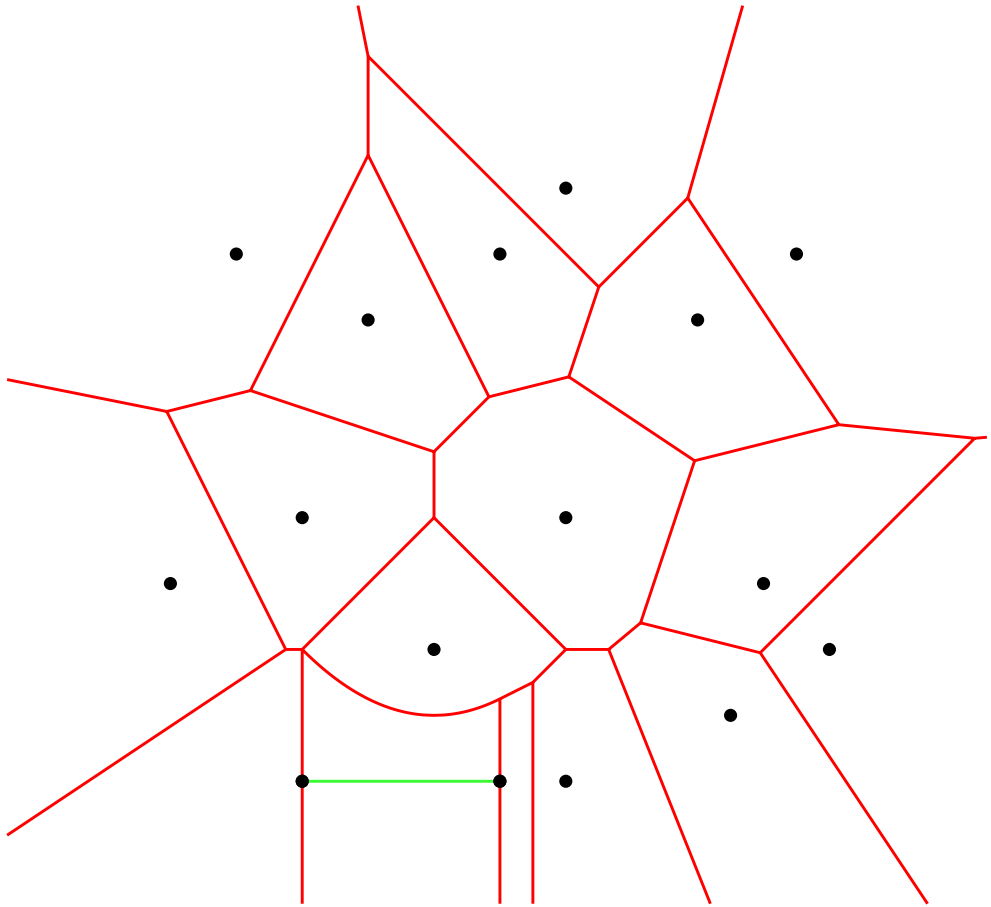
Generalized Voronoi Diagram: Randomized Incremental Construction

- Start with the vertices of S, and compute their Voronoi diagram.



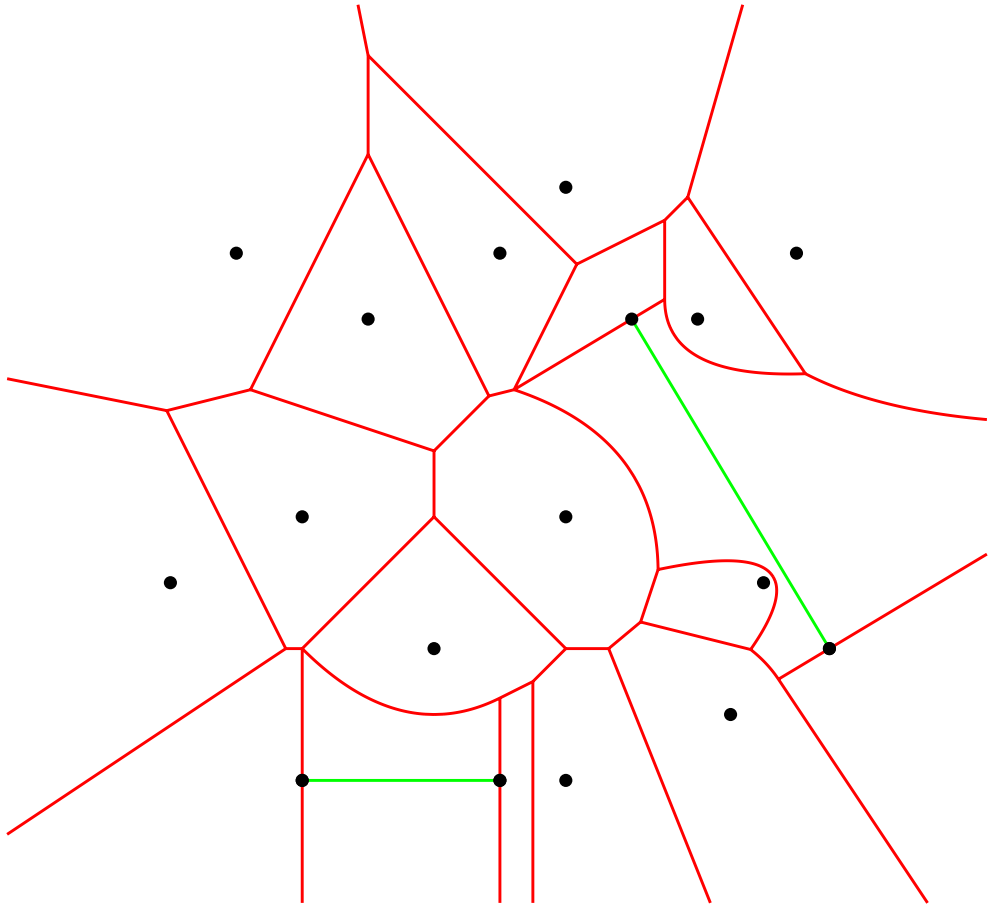
Generalized Voronoi Diagram: Randomized Incremental Construction

- Start with the vertices of S, and compute their Voronoi diagram. Insert segments.



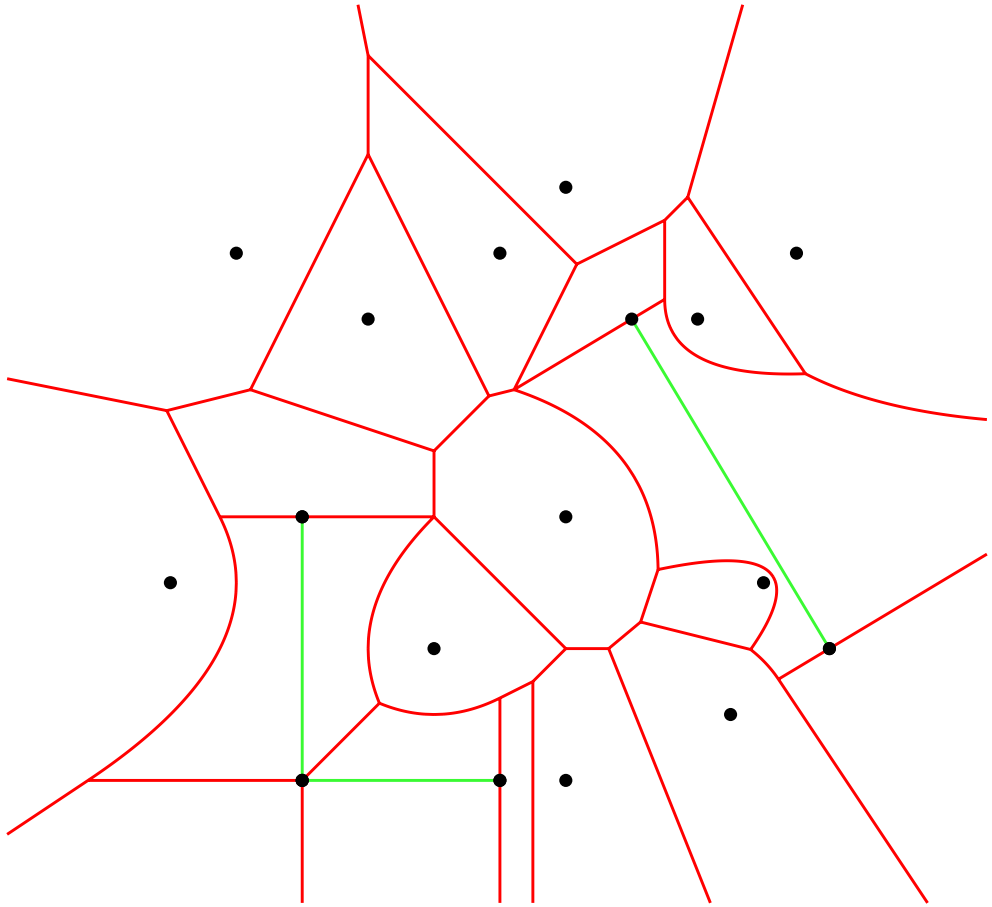
Generalized Voronoi Diagram: Randomized Incremental Construction

- Start with the vertices of S, and compute their Voronoi diagram. Insert segments, randomly.



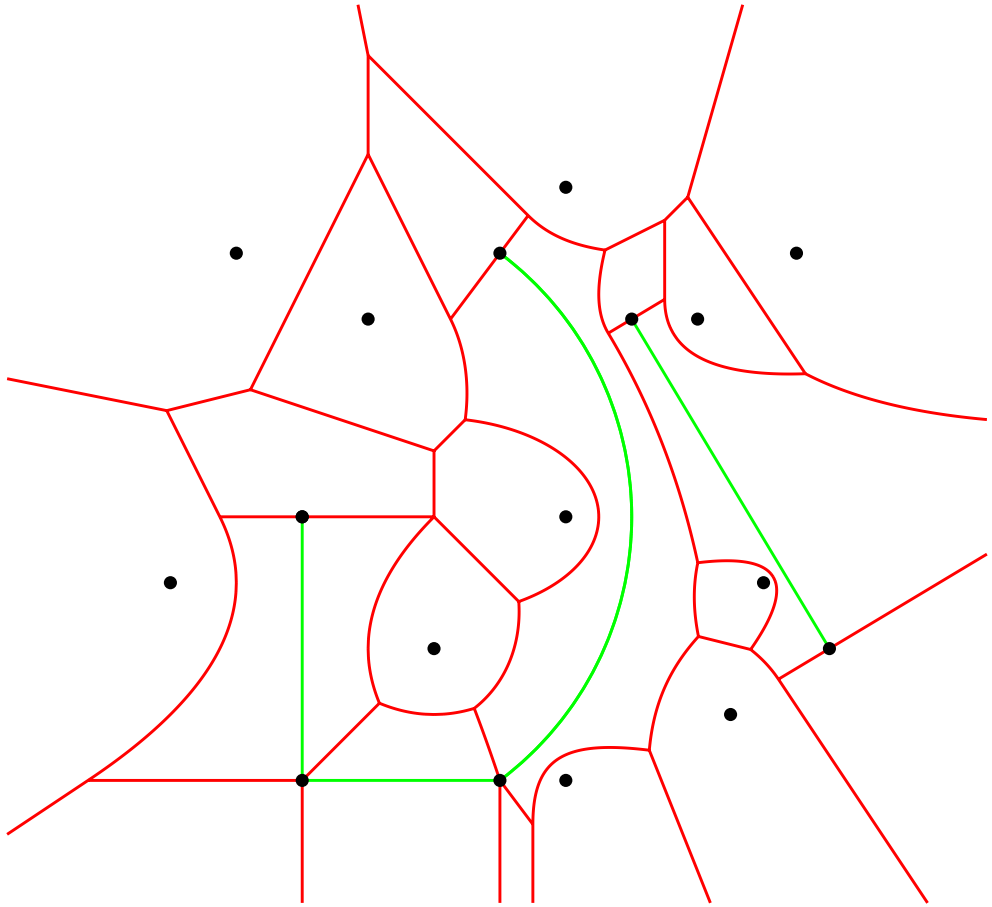
Generalized Voronoi Diagram: Randomized Incremental Construction

- Start with the vertices of S, and compute their Voronoi diagram. Insert segments, randomly, one after the other.



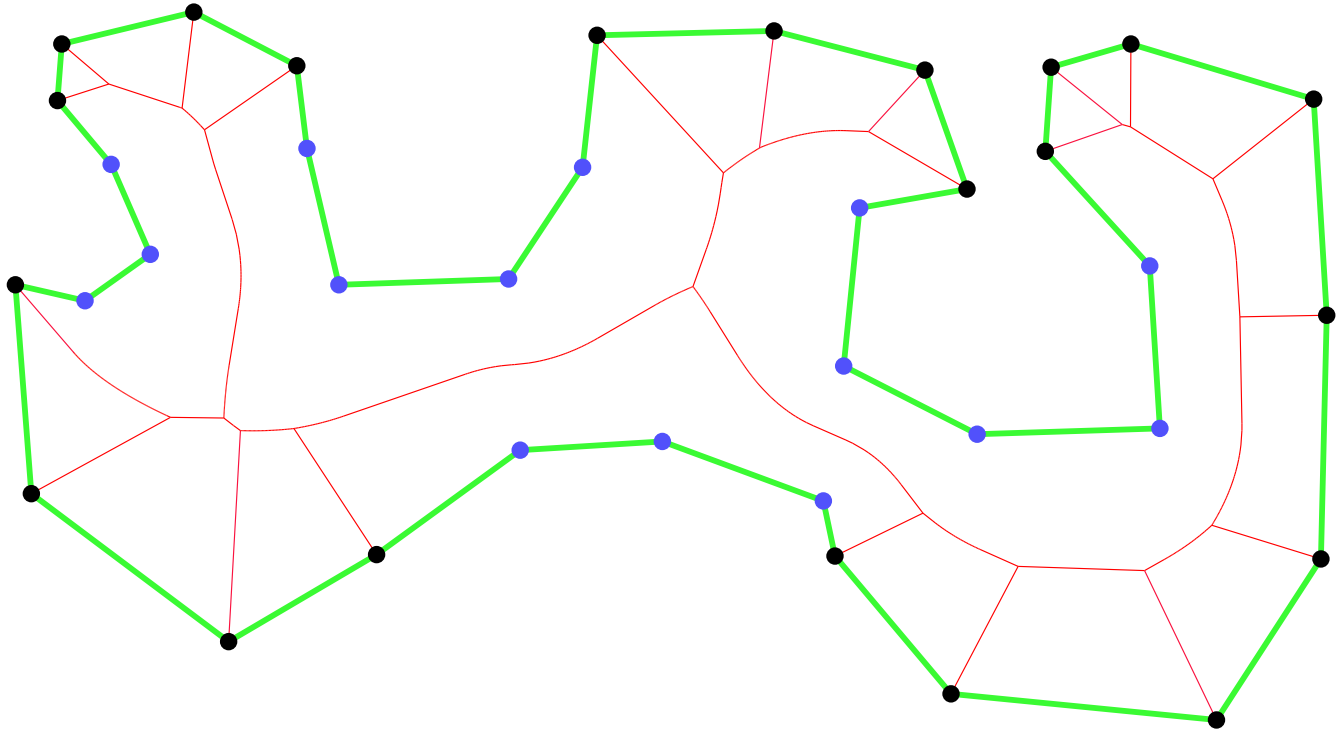
Generalized Voronoi Diagram: Randomized Incremental Construction

- Start with the vertices of S, and compute their Voronoi diagram. Insert segments, randomly, one after the other. Same for the arcs.



Generalized Voronoi Diagram: Medial Axis

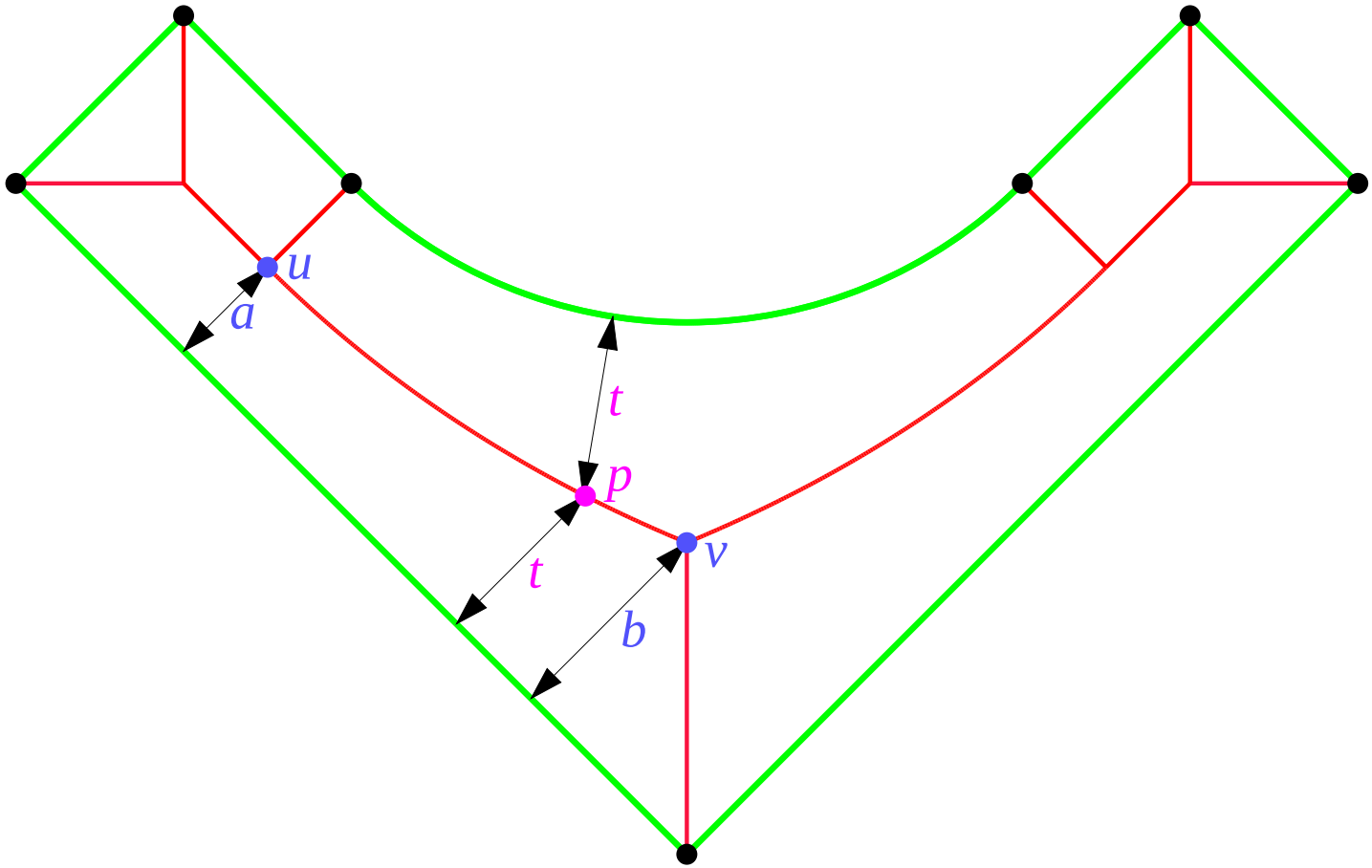
- The *medial axis* is a subset of the Voronoi diagram; it contains only those points on the Voronoi diagram which have at least two disjoint footprints on the boundary.



Generalized Voronoi Diagram: Parameterization of Voronoi Edges

- We assign a clearance-based parameterization $f : [a,b] \rightarrow \mathbb{R}^2$ to every edge e, where a is the minimum and b is the maximum clearance of points of e.

- The coordinates of a point p of e with clearance t are obtained by evaluating f: we have $p = f(t)$.



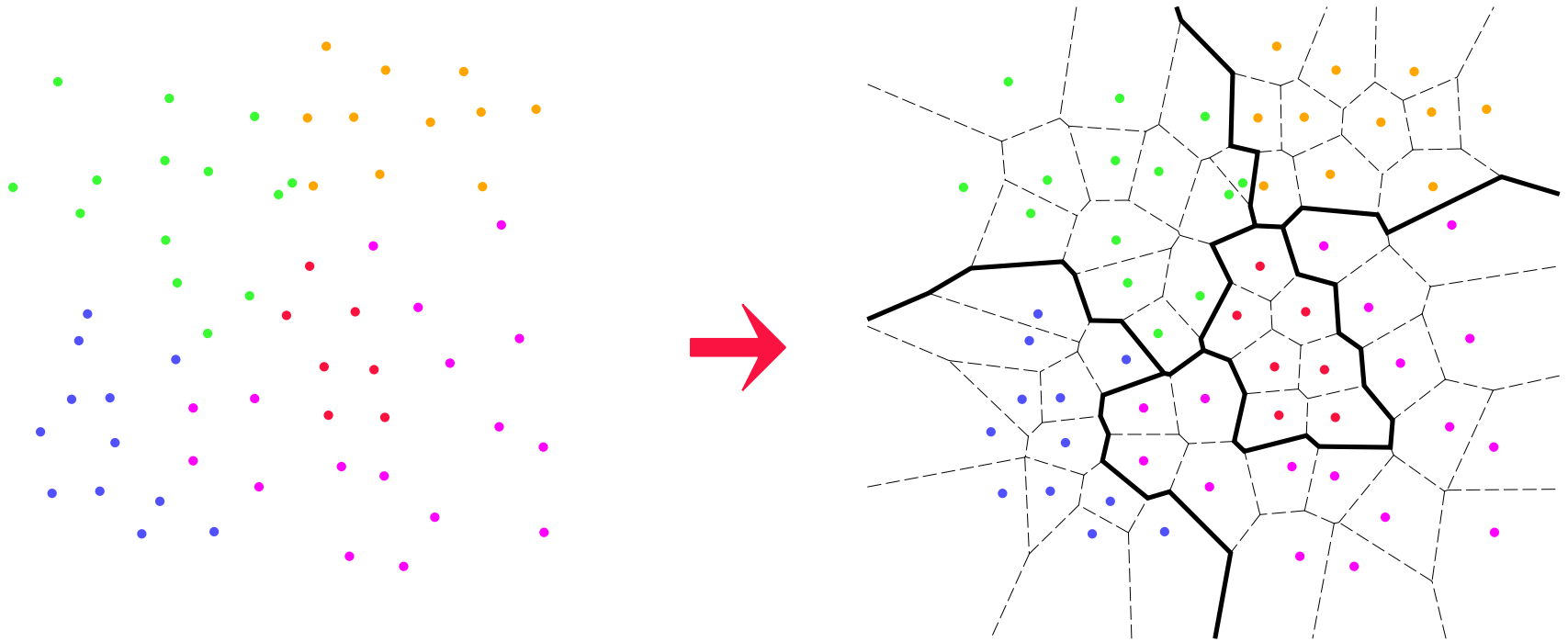
Applications of Voronoi Diagrams

- We do already know that the availability of the Voronoi diagram allows us to solve the following problems in $O(n)$ time:
  - ⋆ CLOSESTPAIR,
  - ⋆ ALLNEARESTNEIGHBORS,
  - ⋆ TRIANGULATION.
- In the sequel, we will study
  - ⋆ Statistical classification and shape estimation,
  - ⋆ EUCLIDEANMINIMUMSPANNINGTREE (EMST),
  - ⋆ approximate EUCLIDEANTRAVELINGSALESMANTOUR (ETST),
  - ⋆ MAXIMUMEMPTYCIRCLE,
  - ⋆ OFFSETTING,
  - ⋆ FINDING A GOUGE-FREE PATH.

Statistical Classification and Shape Estimation

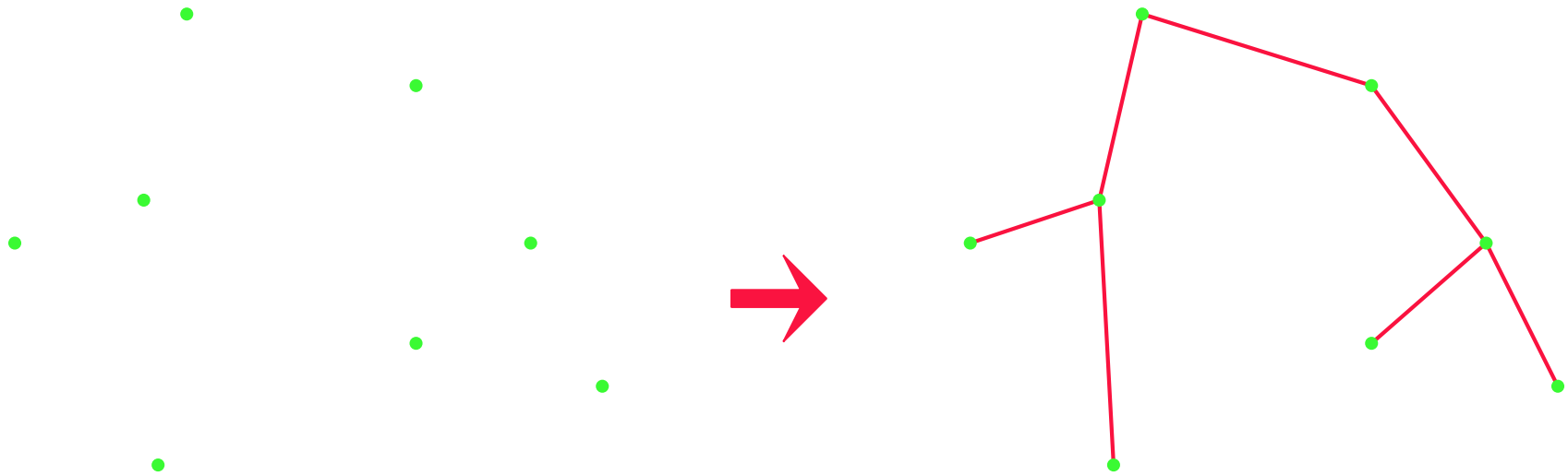
- Given are sets of differently colored points in the plane. What is a suitable partition of the plane according to the colors of the points?

- We can compute the Voronoi diagram and color every Voronoi cell with its point's color.



Euclidean Minimum Spanning Tree

- Consider a set $S := \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$, and assume that we want to compute a Euclidean minimum spanning tree of S.



Euclidean Minimum Spanning Tree (cont'd)

- Obviously, we could apply standard techniques of graph theory by computing an EMST on the graph $\mathcal{G} := (V, E)$, where $V := S$ and $E := S \times S$, and where the Euclidean length of an edge is taken as its weight.

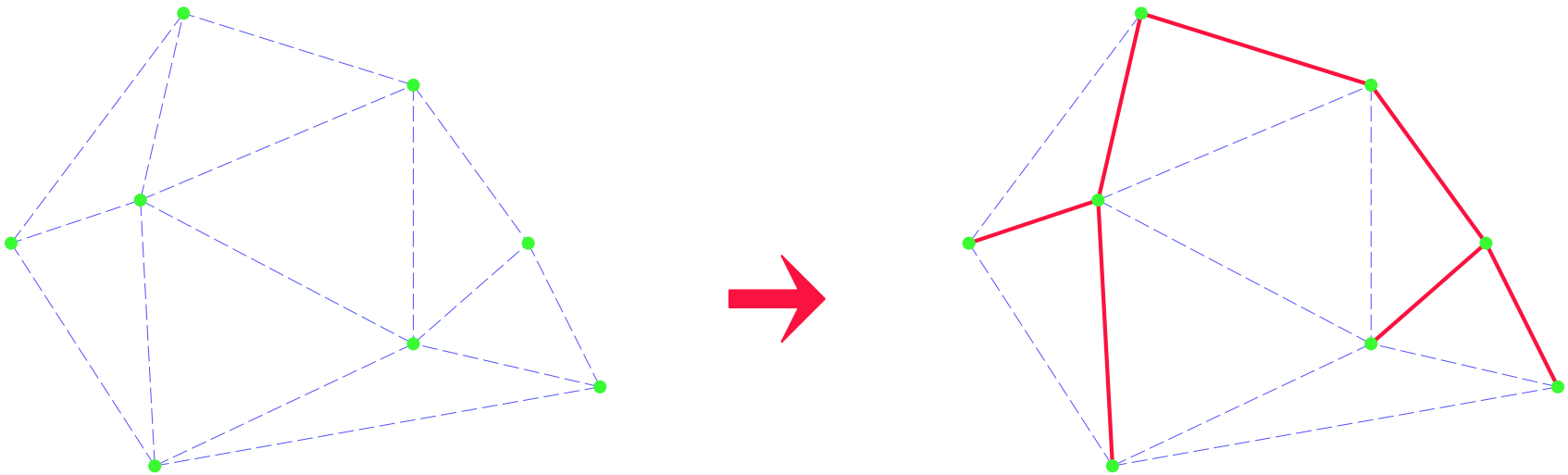
- Lemma [Prim]: Assume that $\mathcal{G}$ is connected, and let V_1, V_2 be a partition of V. There is a minimum spanning tree of $\mathcal{G}$ which contains the shortest of the edges with one terminal in V_1 and the other in V_2.

- *Prim's algorithm* starts with a small tree $\mathcal{T}$ and grows it until it contains all nodes of $\mathcal{G}$. Initially, $\mathcal{T}$ contains just one arbitrary node of V. At each stage one node not yet in $\mathcal{T}$ but closest to (a node of) $\mathcal{T}$ is added to $\mathcal{T}$. Prim's algorithm can be implemented to run in $O(|V|^2)$ time.

- *Kruskal's algorithm* begins with a spanning forest, where each forest is initialized with one node of V. It repeatedly joins two trees together by picking the shortest edge between them until a spanning tree of the entire graph is obtained. Kruskal's algorithm can be implemented to run in $O(|E| \log |E|)$ time.

Euclidean Minimum Spanning Tree (cont'd)

- Can we do any better than $O(n^2)$ when computing EMSTs?
- Lemma: $\mathcal{DT}(S)$ contains an EMST of S as sub-graph.

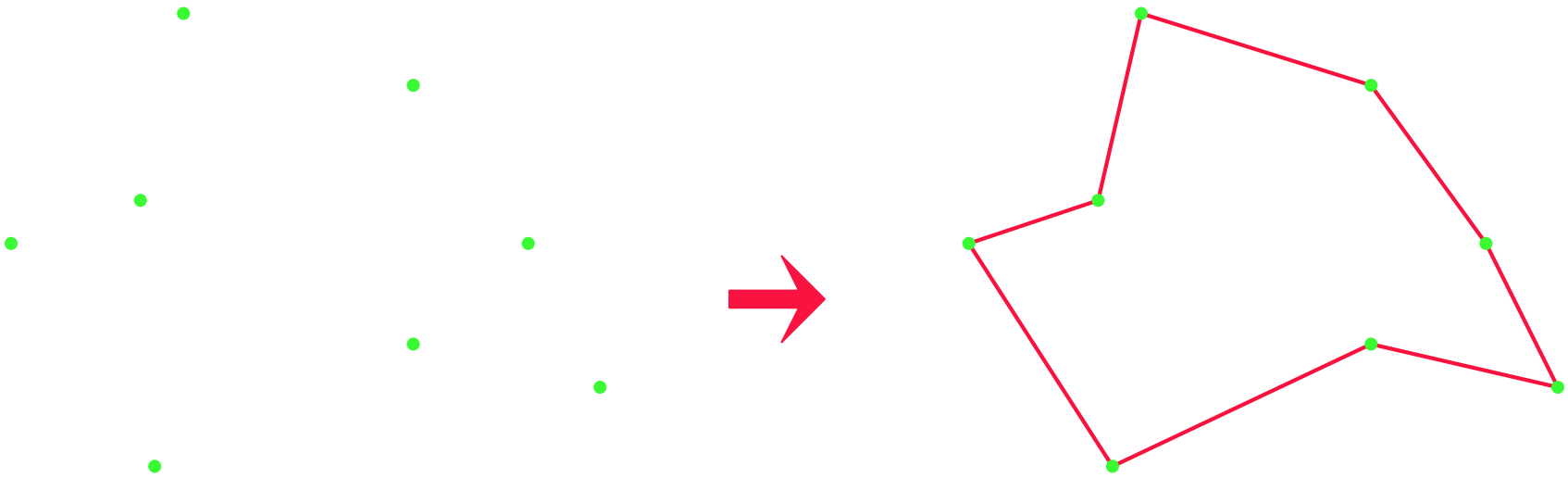


Euclidean Minimum Spanning Tree (cont'd)

- Thus, there is no need to consider the full graph on S.
- Rather, we can apply Kruskal's algorithm to $\mathcal{DT}(S)$, and obtain an $O(n \log n)$ algorithm for computing EMSTs.
- Lemma: An EMST of S can be computed from the Delaunay triangulation of S in time $O(n)$.
- Proof: Observe that $\mathcal{DT}(S)$ is a planar graph, and use Cheriton and Tarjan's "clean-up refinement" of Kruskal's algorithm.
- Note: an EMST is unique if all inter-point distances on S are distinct.

Approximate Traveling Salesman Tour

- The EUCLIDEANTRAVELINGSALESMANPROBLEM (ETSP) asks to compute a shortest closed path on $S \subset \mathbb{E}^2$ that visits all points of S.



Approximate Traveling Salesman Tour

- Theorem: TSP and ETSP are $\mathcal{NP}$-hard.
- Let *OPT* be the true length of a TSP tour, and let *APX* be the length of an approximate solution.
- Def.: An approximation algorithm provides a *constant-factor approximation* if a constant $c \in \mathbb{R}^+$ exists such that $APX \leq c \cdot OPT$ holds for all inputs.
- Constant-factor approximations to ETSP:
  - ⋆ Doubling-the-EMST heuristic: $c = 2$; runs in $O(n \log n)$ time.
  - ⋆ Christofides' heuristic: $c = 3/2$; runs in $O(n^3)$ time.

Approximate Traveling Salesman Tour (cont'd)

- Note that the Euclidean metric obeys the triangle inequality.

- Recent *polynomial-time approximation schemes* (PTAS):

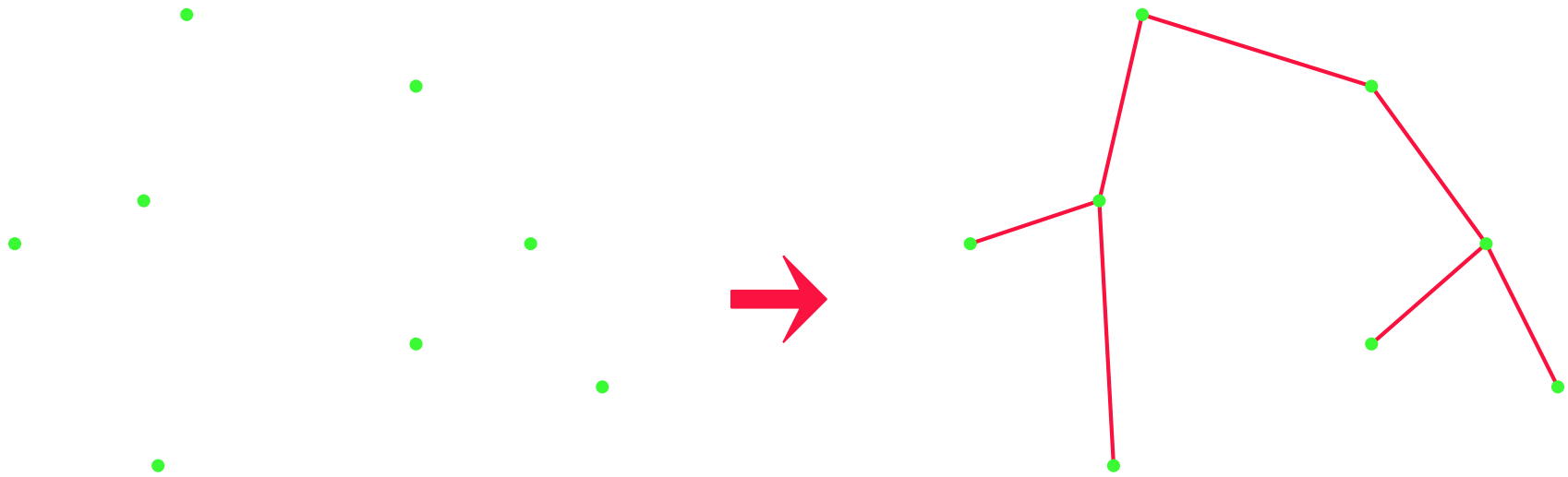
  ⋆ Arora (1996), Mitchell (1996), Rao and Smith (1998).
  ⋆ $c = 1 + \varepsilon$ (for $\varepsilon \in \mathbb{R}^+$).
  ⋆ Common to these algorithms is the fact that $O(1/\varepsilon)$ appears as exponent of n or $\log n$.

- Hardness of approximation for non-Euclidean TSPs with symmetric metric:

  ⋆ Papadimitriou and Vempala (2000): No polynomial-time constant-factor approximation algorithm which achieves $c \leq (1 + 1/219)$ exists, unless $\mathcal{P} = \mathcal{NP}$.

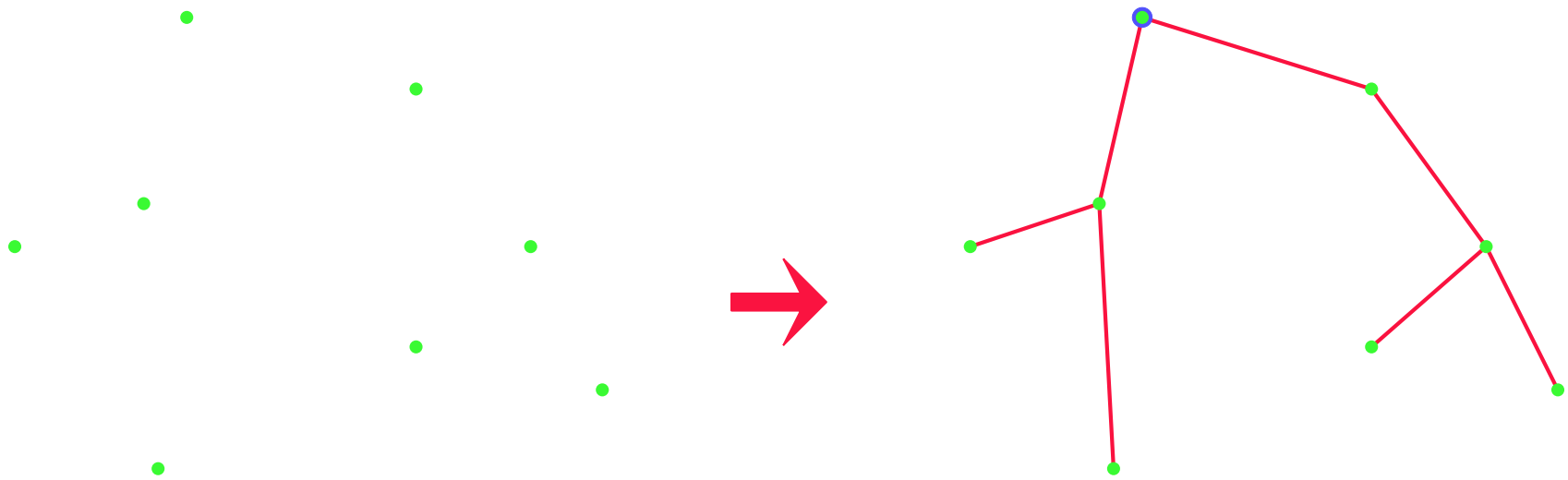
Approximate TST: Doubling-the-EMST Heuristic

- Compute the Euclidean minimum spanning tree $\mathcal{T}(S)$ of S.



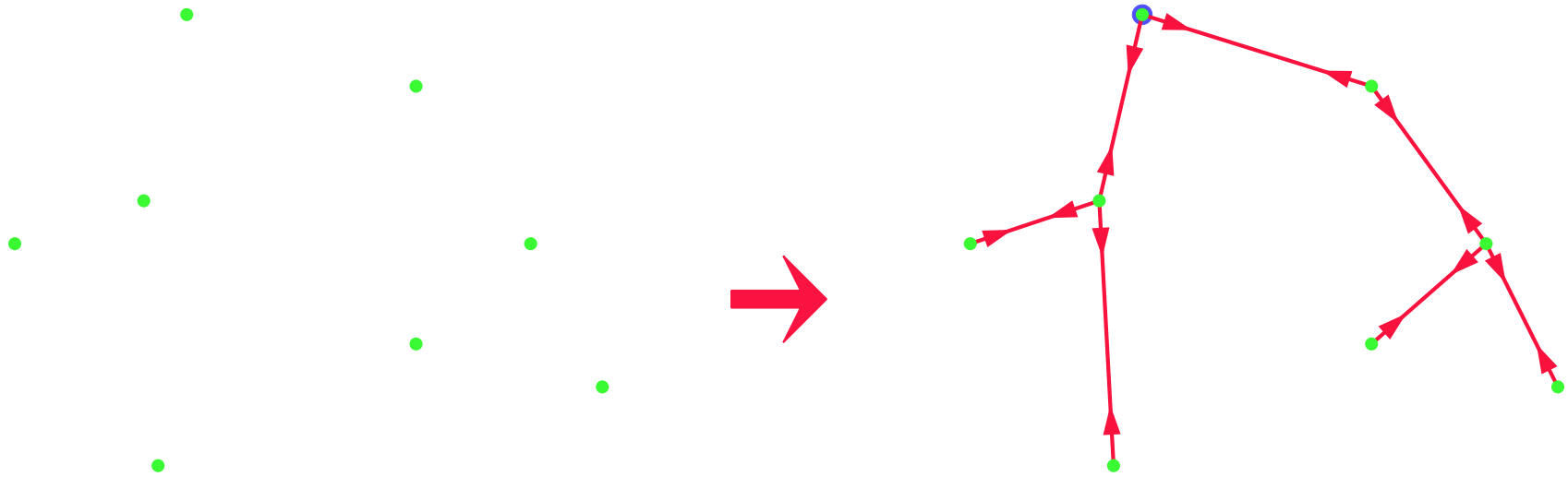
Approximate TST: Doubling-the-EMST Heuristic (cont'd)

- Select an arbitrary node v of $\mathcal{T}(S)$ as root.



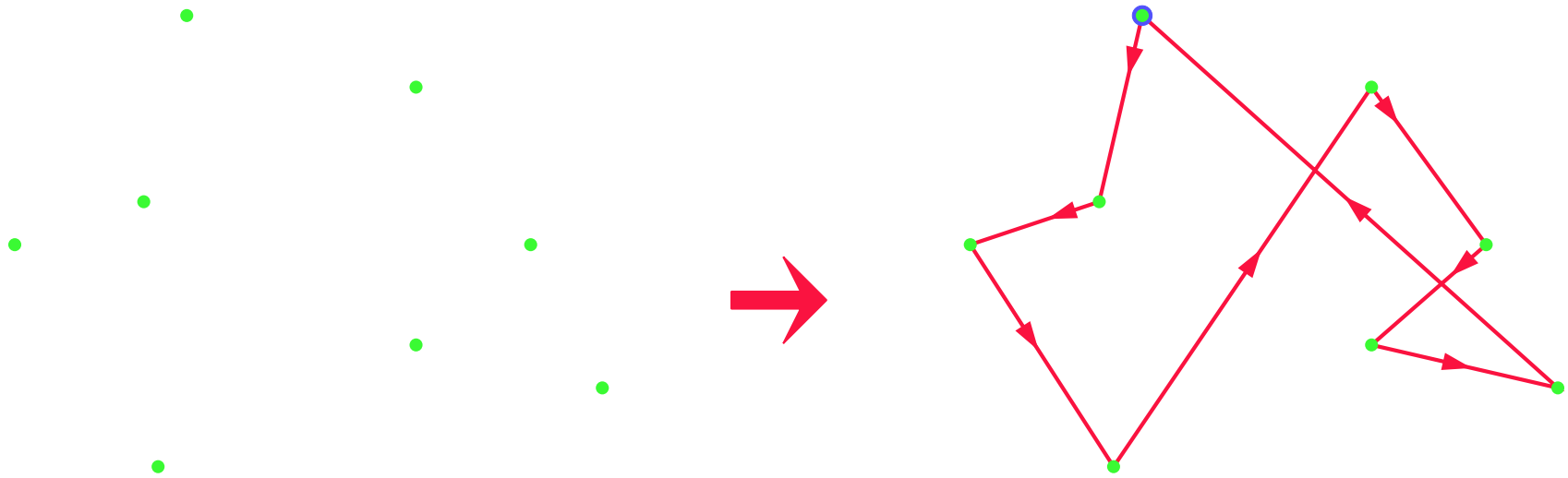
Approximate TST: Doubling-the-EMST Heuristic (cont'd)

- Compute an in-order traversal of $\mathcal{T}(S)$ rooted at v to obtain a tour $\mathcal{C}(S)$.



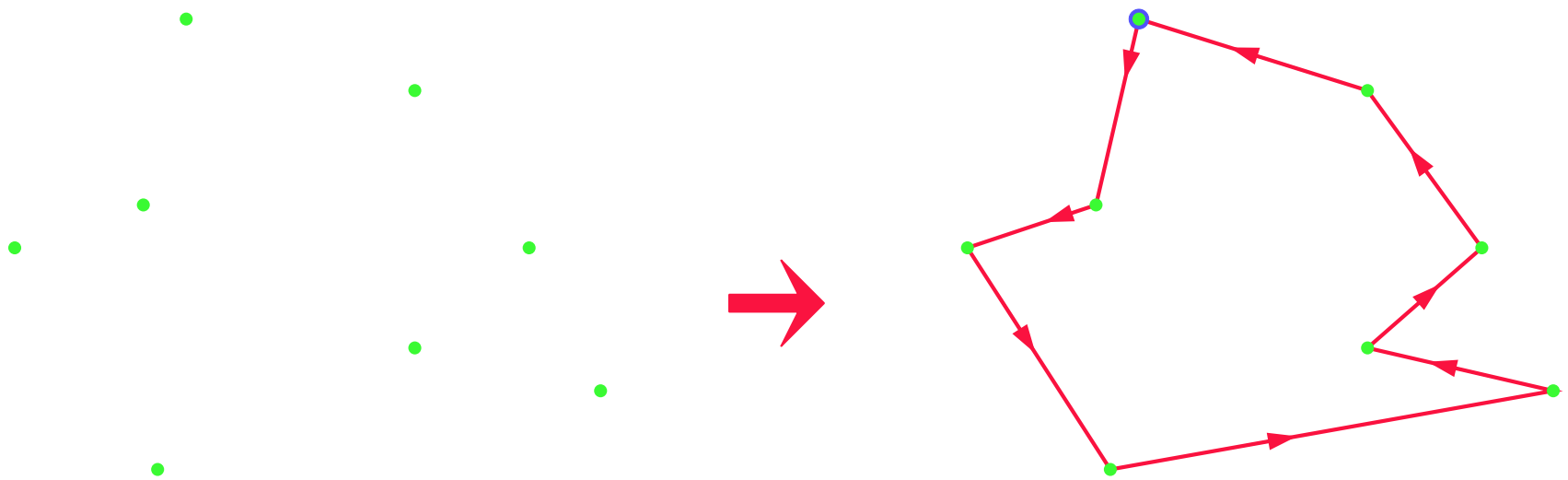
Approximate TST: Doubling-the-EMST Heuristic (cont'd)

- By-pass points already visited, thus shortening $\mathcal{C}(S)$.



Approximate TST: Doubling-the-EMST Heuristic (cont'd)

- Apply 2-opt moves (at additional computational cost).

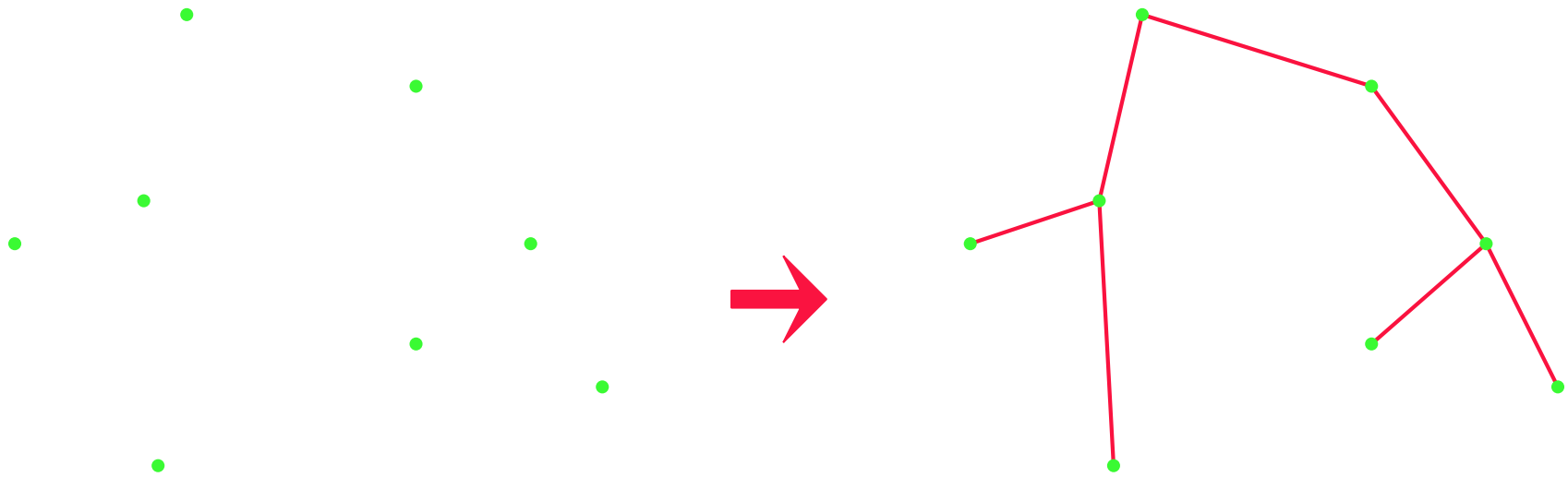


Approximate TST: Doubling-the-EMST Heuristic (cont'd)

- Time complexity: $O(n \log n)$ for computing the EMST $\mathcal{T}(S)$.
- Factor of approximation: $c = 2$.

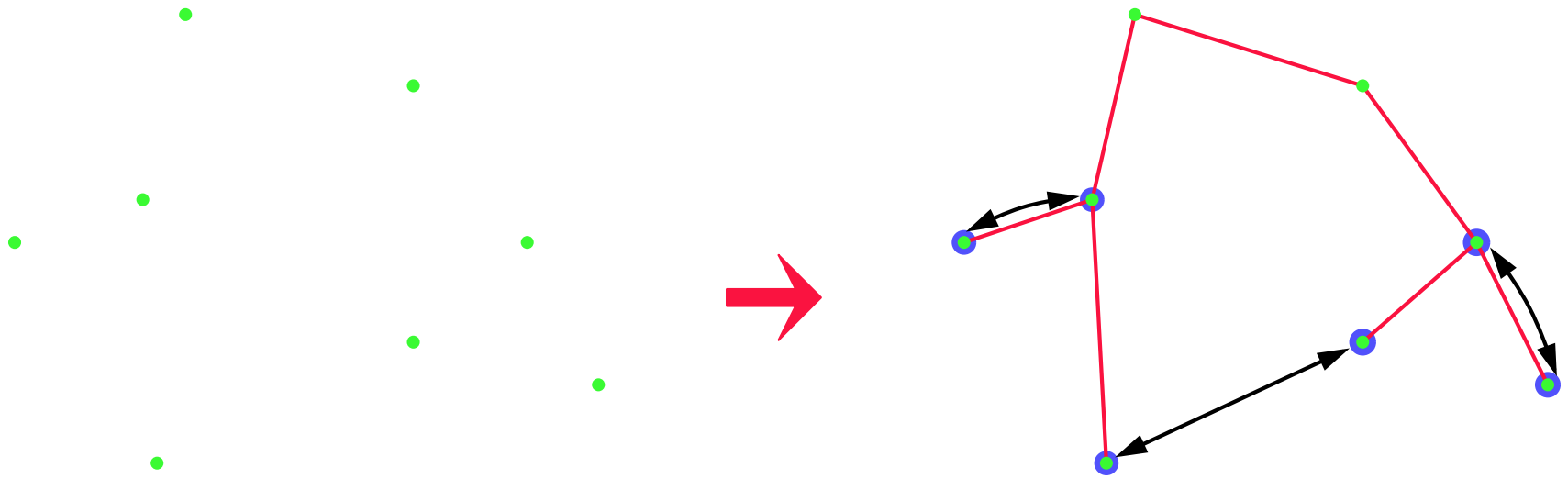
Approximate TST: Christofides' Heuristic

- Compute the Euclidean minimum spanning tree $\mathcal{T}(S)$ of S.



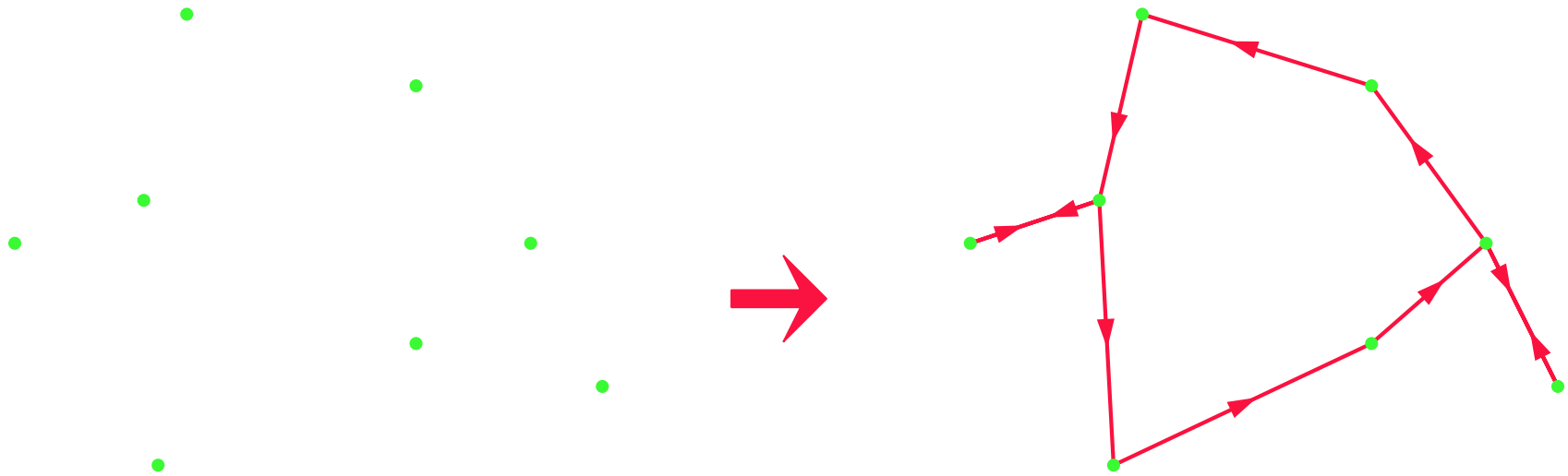
Approximate TST: Christofides' Heuristic

- Compute a minimum Euclidean matching $\mathcal{M}$ on the vertices of odd degree in $\mathcal{T}(S)$.



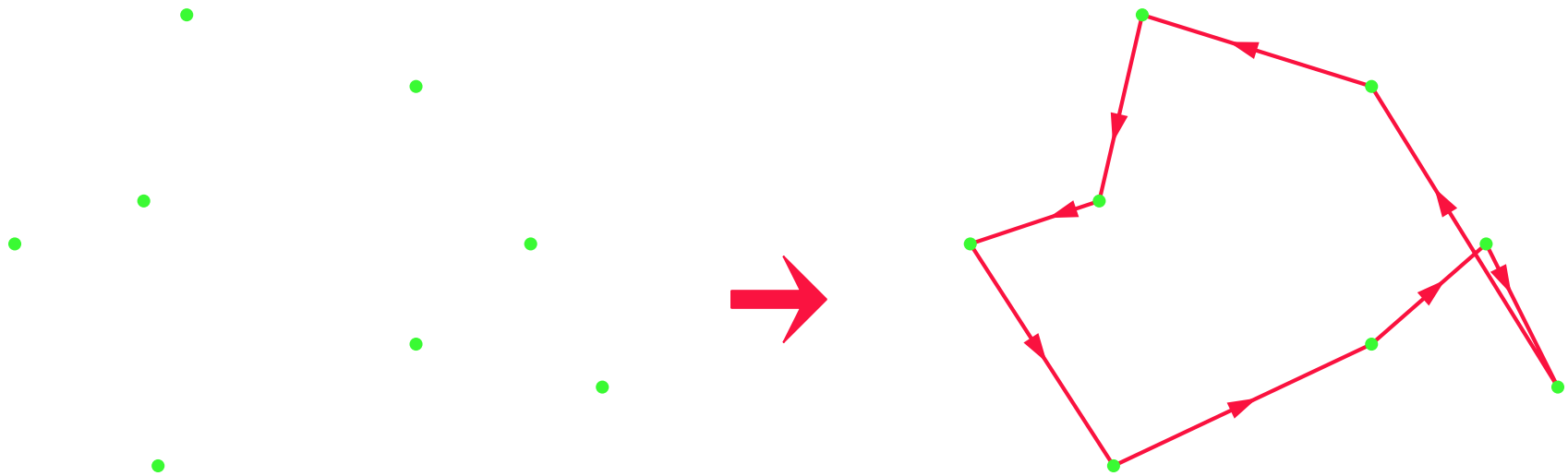
Approximate TST: Christofides' Heuristic (cont'd)

- Compute an Eulerian tour $\mathcal{C}$ on $\mathcal{T} \cup \mathcal{M}$.



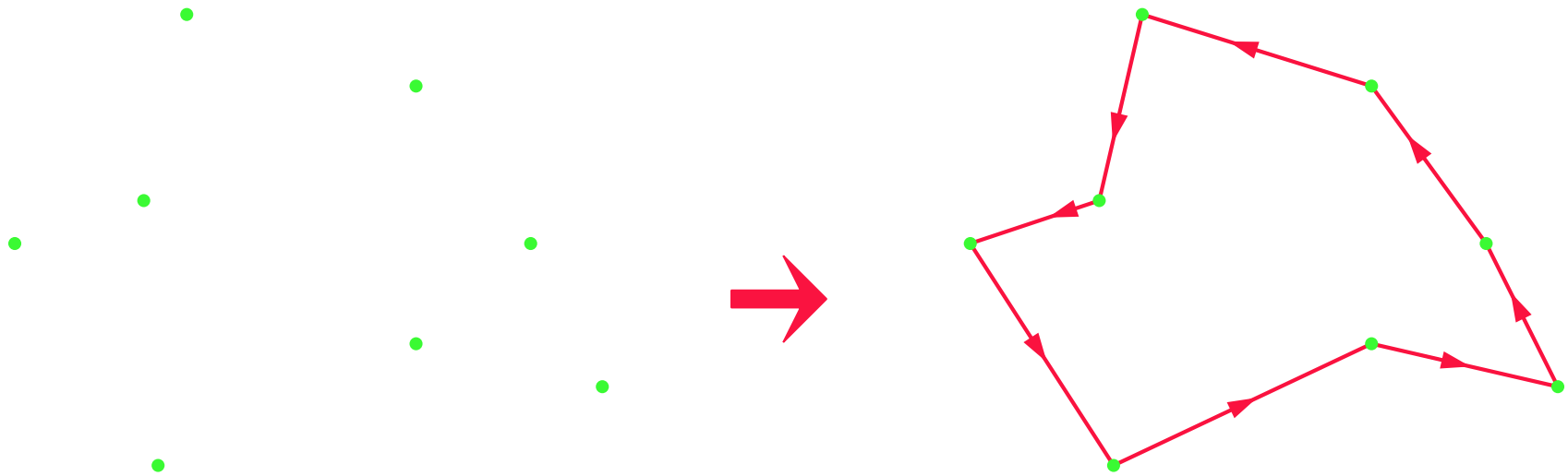
Approximate TST: Christofides' Heuristic (cont'd)

- By-pass points already visited, thus shortening $\mathcal{C}$.



Approximate TST: Christofides' Heuristic (cont'd)

- Apply 2-opt moves (at additional computational cost).

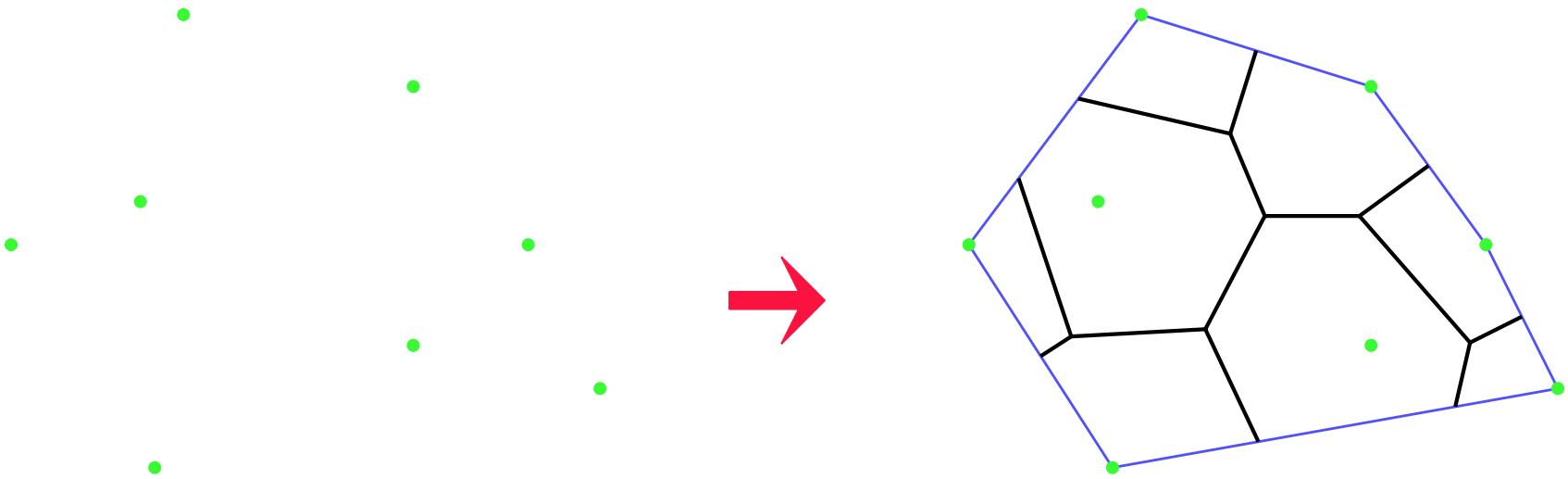


Approximate TST: Christofides' Heuristic (cont'd)

- Time complexity: $O(n^3)$ for computing the Euclidean matching.
- Factor of approximation: $c = \frac{3}{2}$.

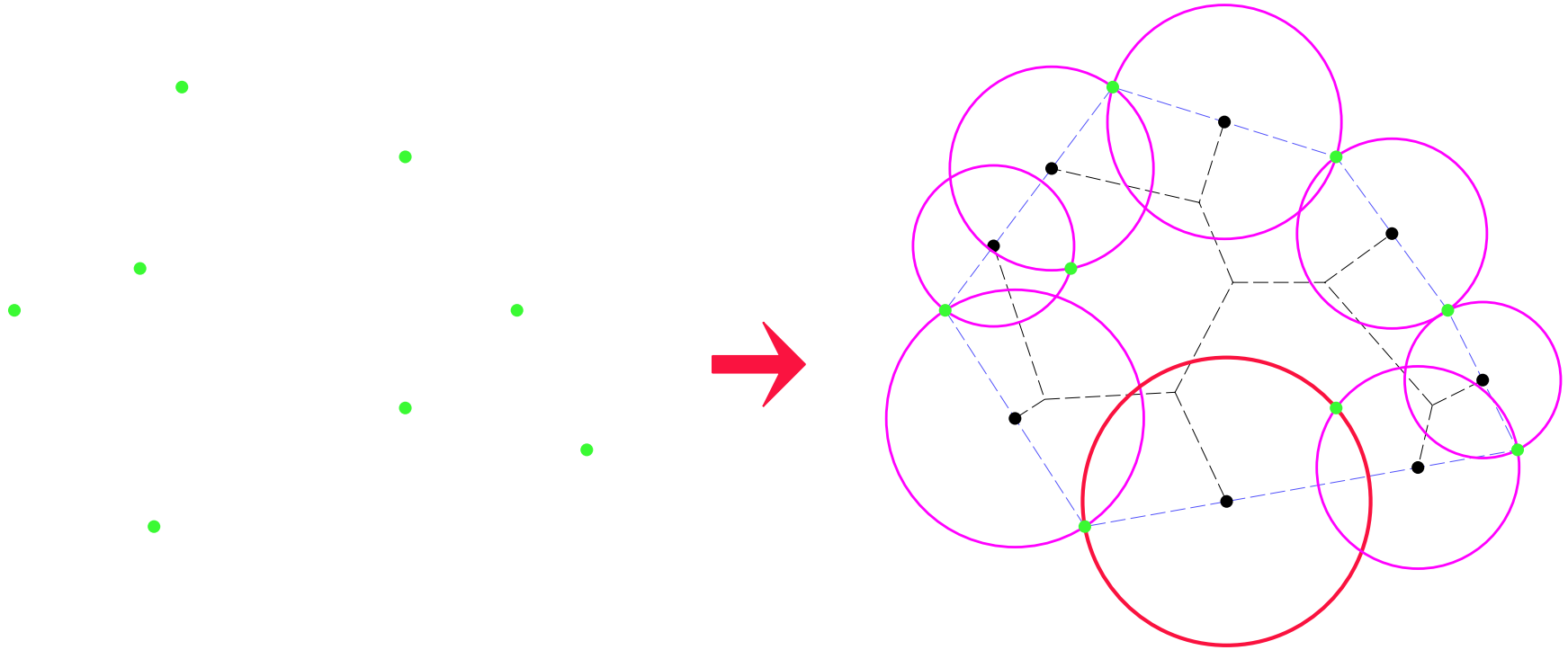
Maximum Empty Circle

- Restrict $\mathcal{VD}(S)$ to $CH(S)$.



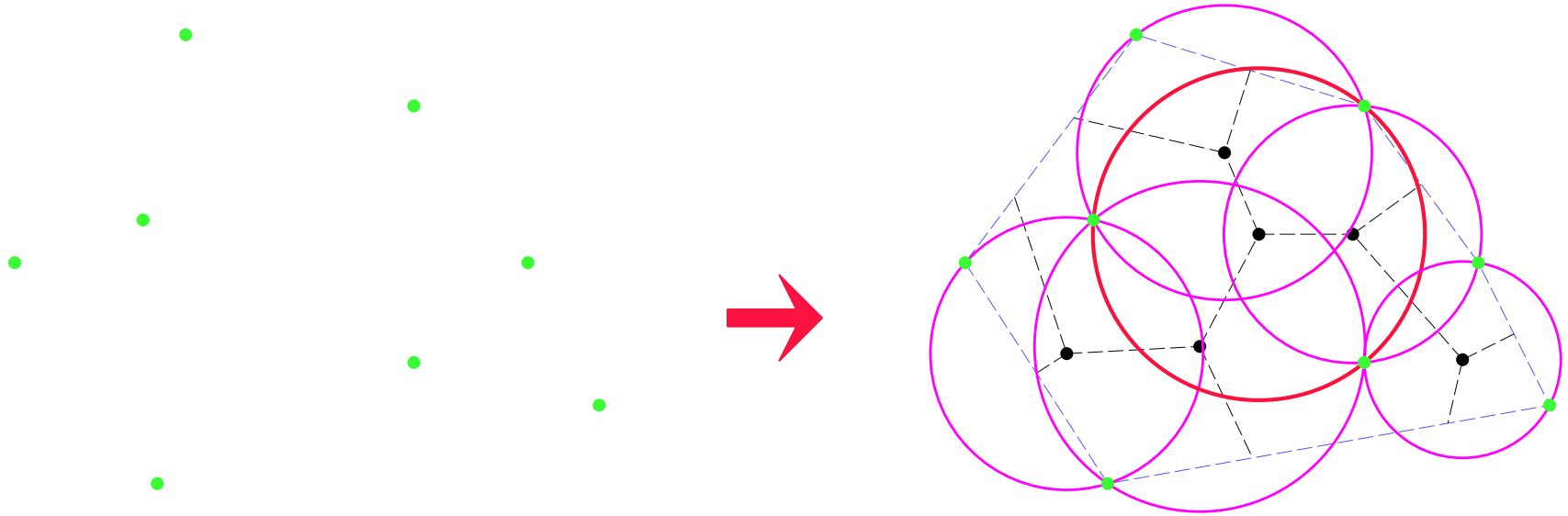
Maximum Empty Circle (cont'd)

- Determine the largest circle centered at an intersection of $\mathcal{VD}(S)$ and $CH(S)$.



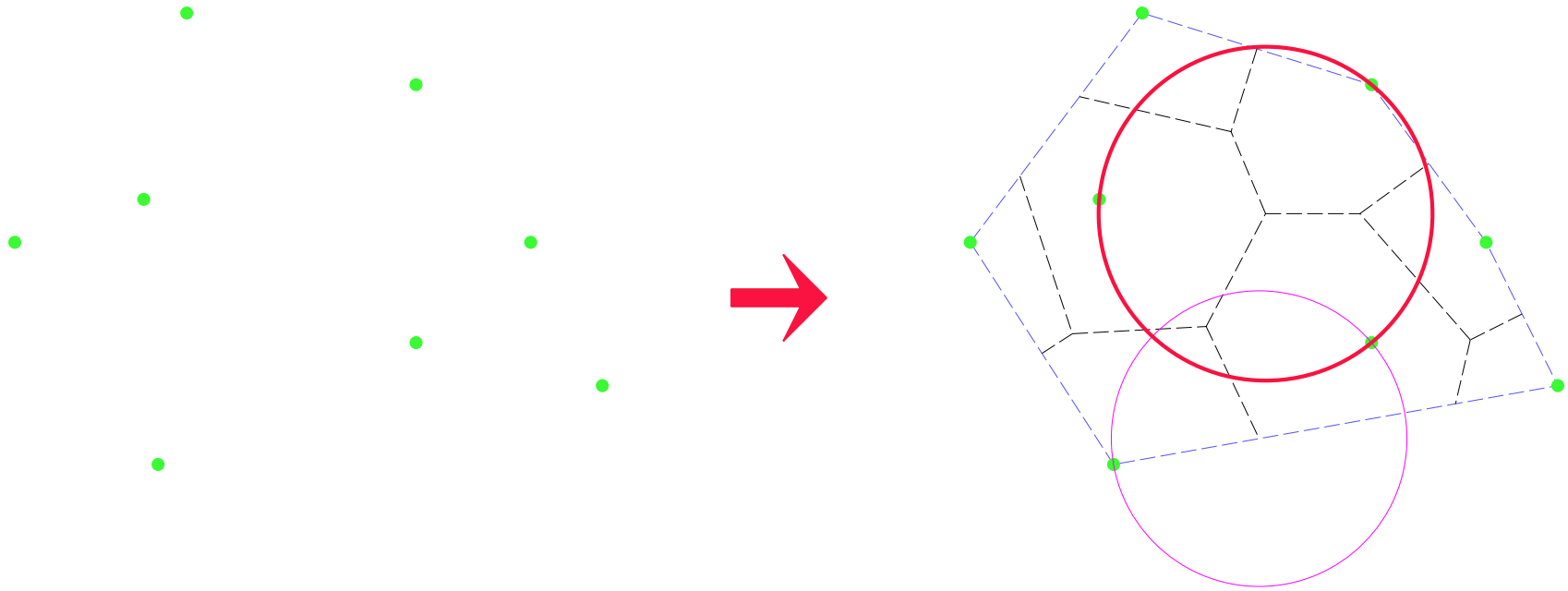
Maximum Empty Circle (cont'd)

- Determine the largest circle centered at an interior node of $\mathcal{VD}(S)$.



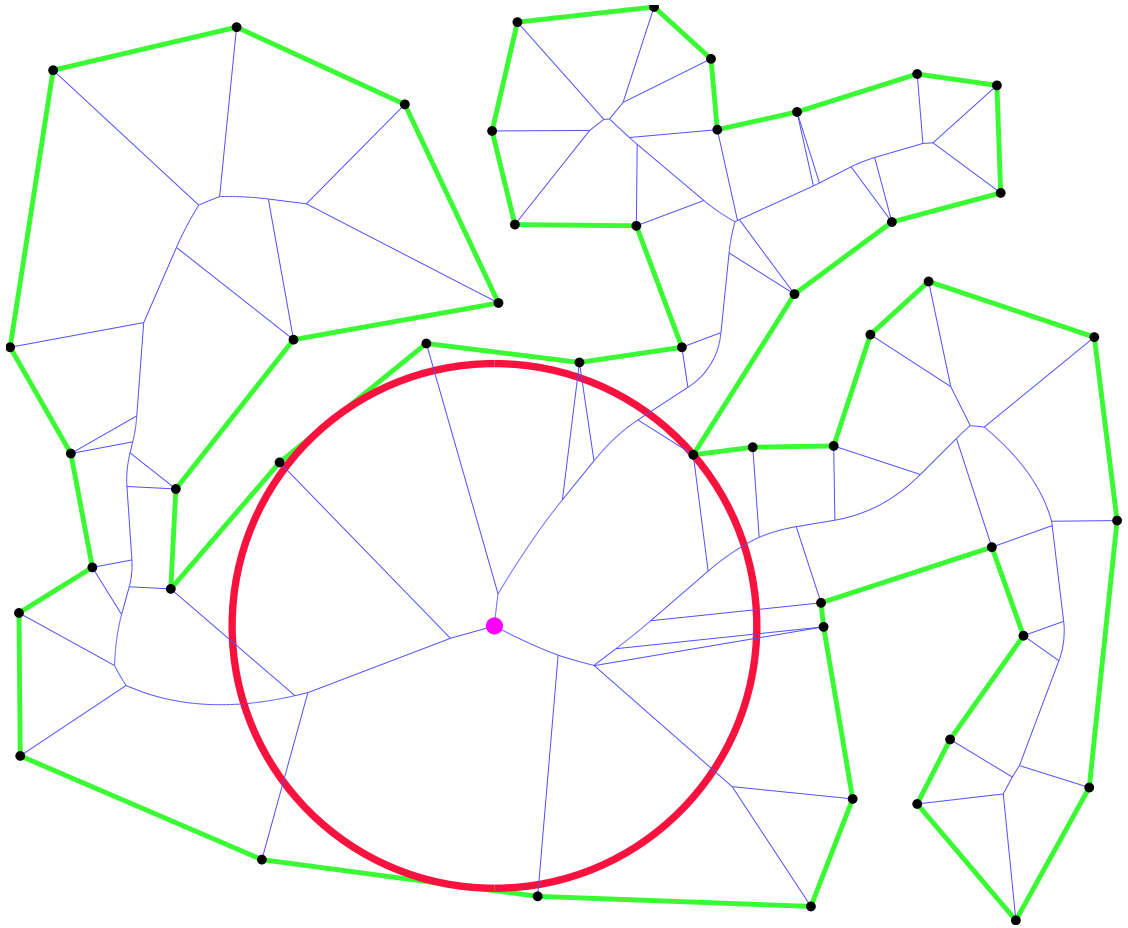
Maximum Empty Circle (cont'd)

- Pick the largest circle among those two categories of circles.



Maximum Inscribed Circle

- Similarly, scanning the Voronoi nodes interior to a polygon yields a maximum inscribed circle in $O(n)$ time.



Offsetting: Minkowski Sum and Difference

- Let A, B be sets, and a, b denote points of A respectively B.
- We define the translation of A by the vector b as

$$A_b := \{a + b : a \in A\}.$$

- The *Minkowski sum* of A and B is defined as

$$A \oplus B := \bigcup_{b \in B} A_b.$$

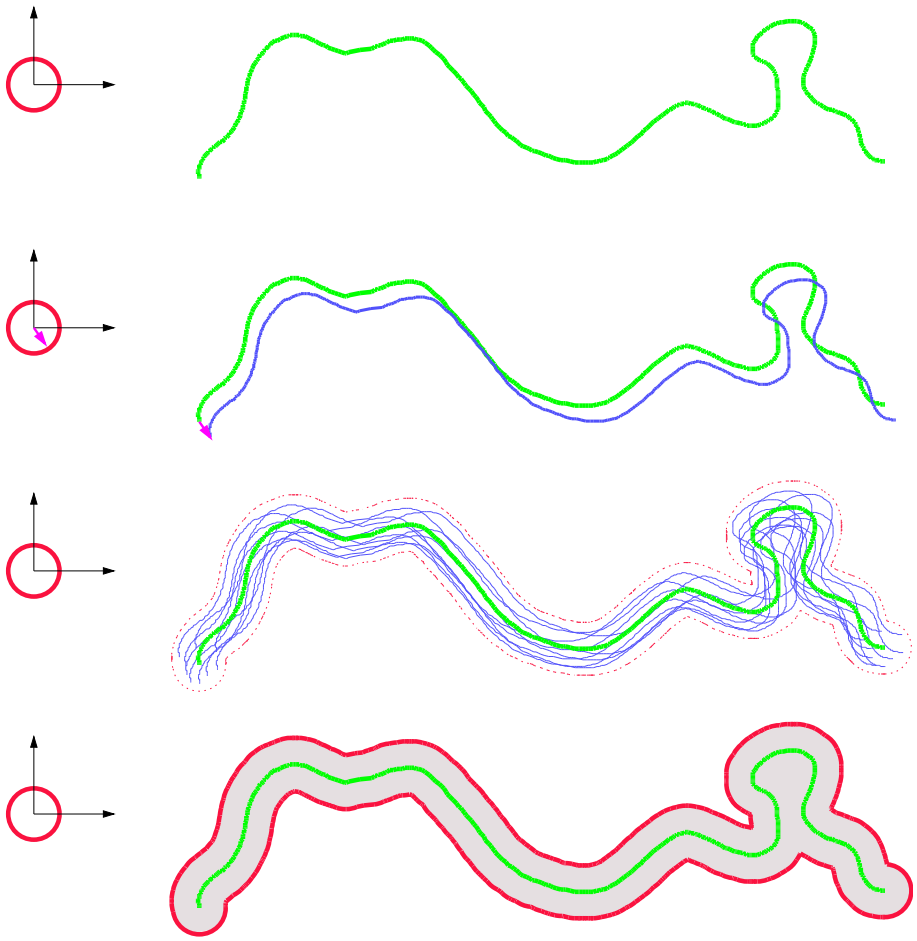
- The *Minkowski difference* of A and B is defined as

$$A \ominus B := \bigcap_{b \in B} A_{-b}.$$

- Note: In general, $(A \oplus B) \ominus B \neq A$.

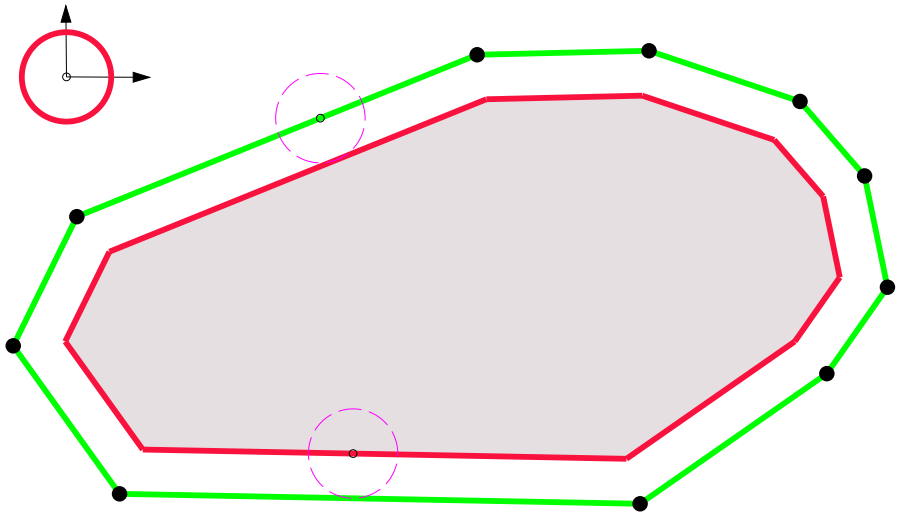
Offsetting: Sample Minkowski Sum

- Let A be a curve, and B be a circular disk centered at the origin. What is $A \oplus B$?



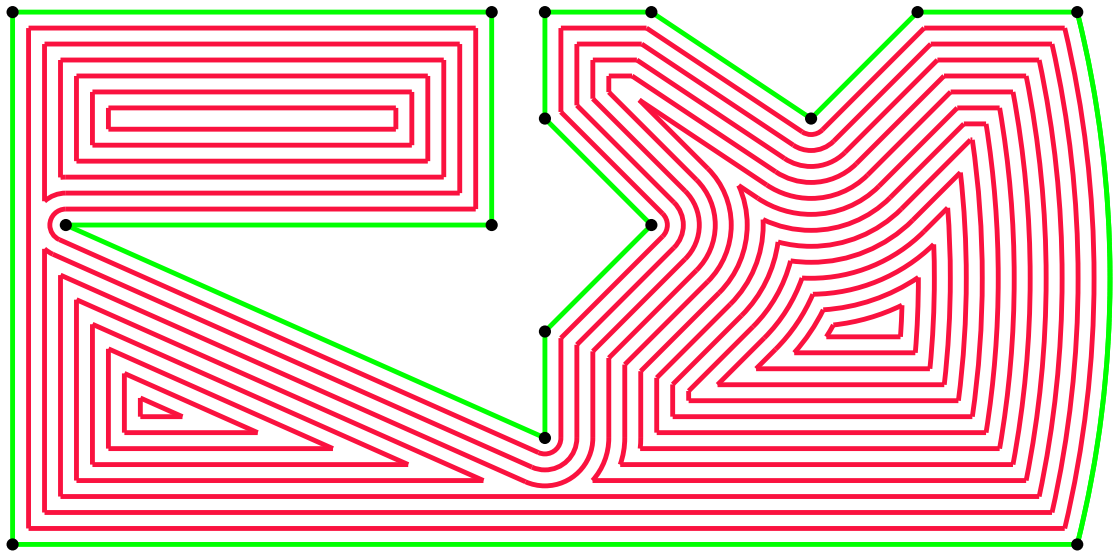
Offsetting: Sample Minkowski Difference

- Let A be a polygon, and B be a circular disk centered at the origin. What is $A \ominus B$?



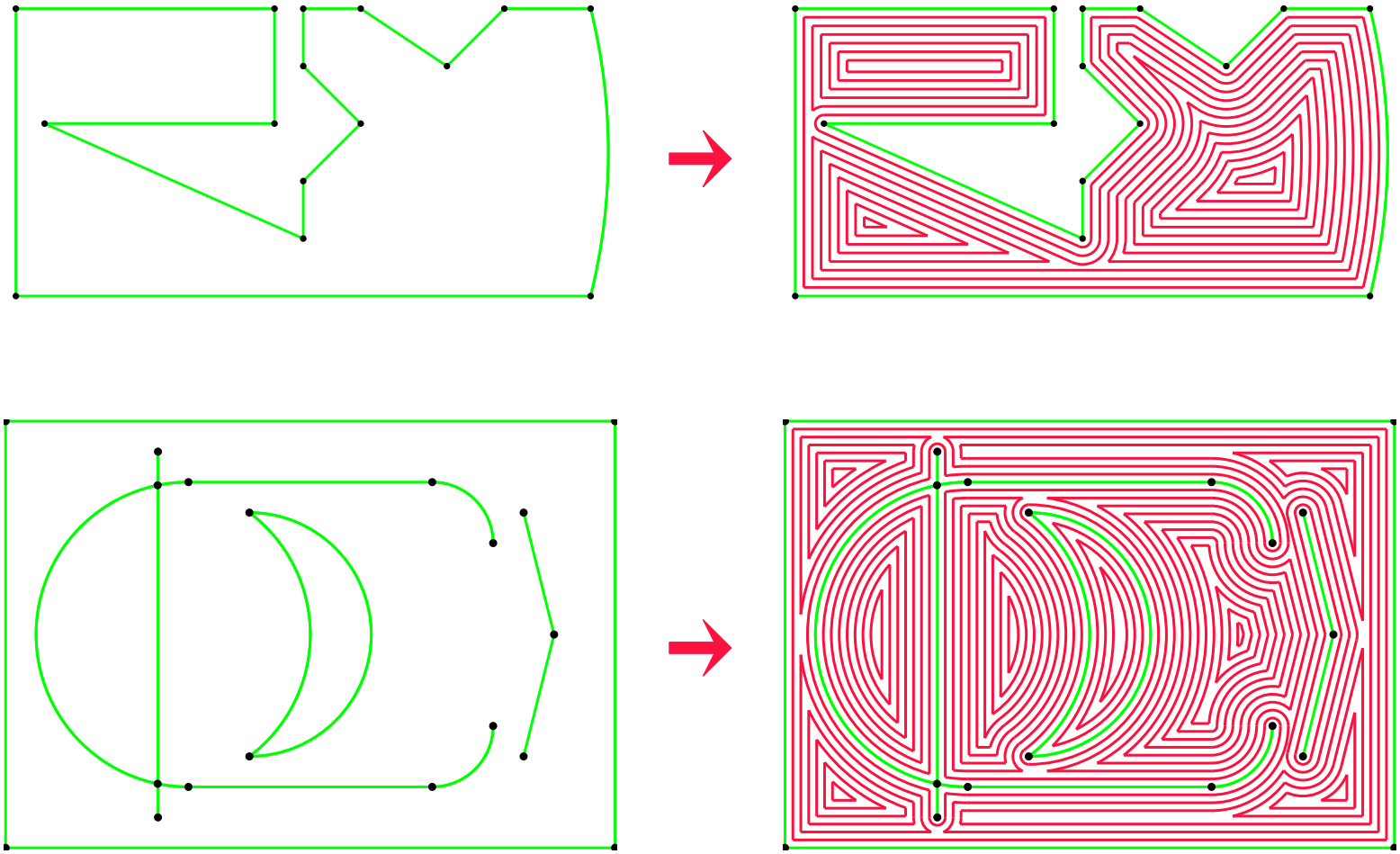
Offsetting: Topological Changes

- Minkowski sums and differences of an area A with a circular disk B centered at the origin are also called *offsets* (in CAD/CAM) and *buffers* (in GIS).
- Note: the boundary of an offset may contain circular arcs even if the input is purely polygonal.
- Note: offsetting may cause topological changes!



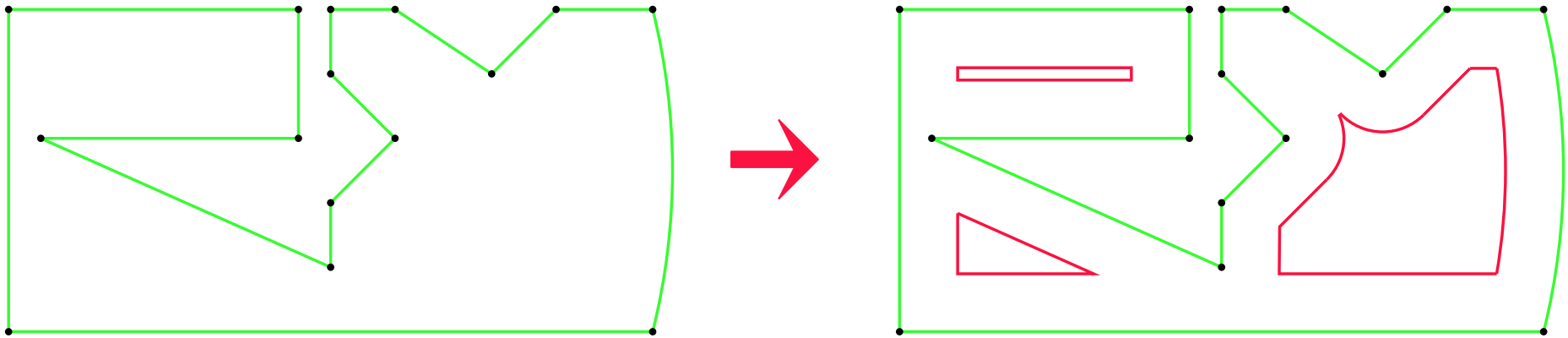
Computation of Offset Patterns

- How can we compute offset patterns reliably and efficiently?



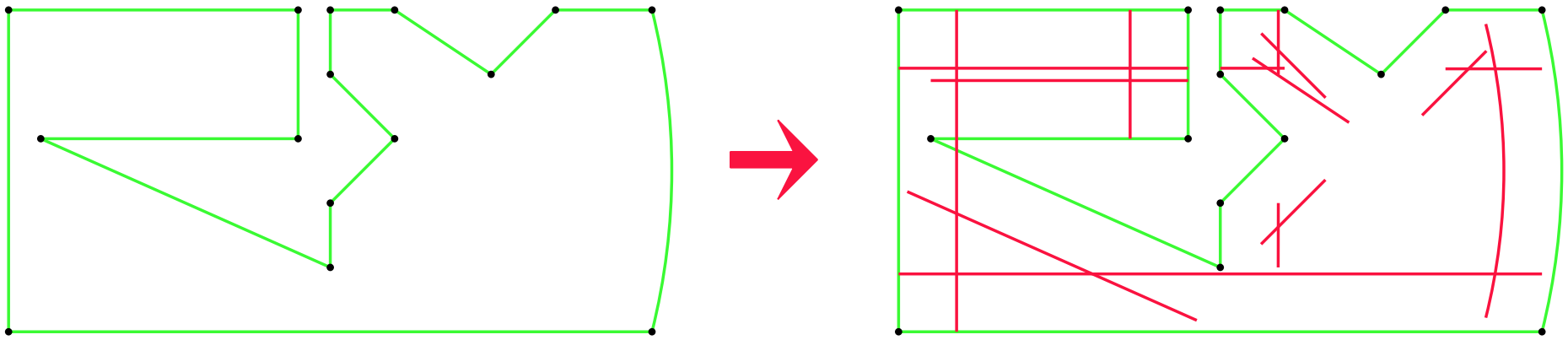
Conventional Offsetting

- How can we compute even just one individual offset?



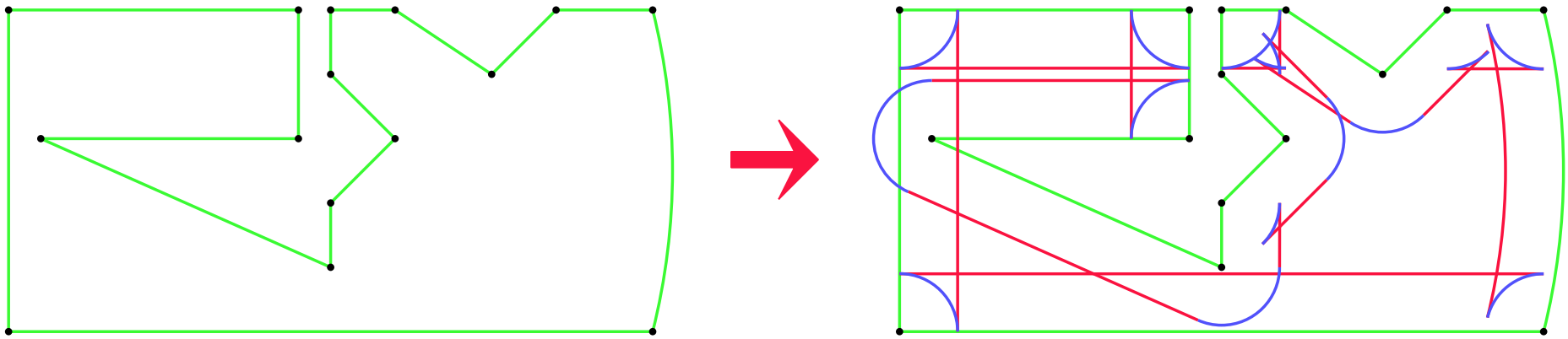
Conventional Offsetting (cont'd)

- First, we compute offset segments (resp. arcs) for every input segment (resp. arc).



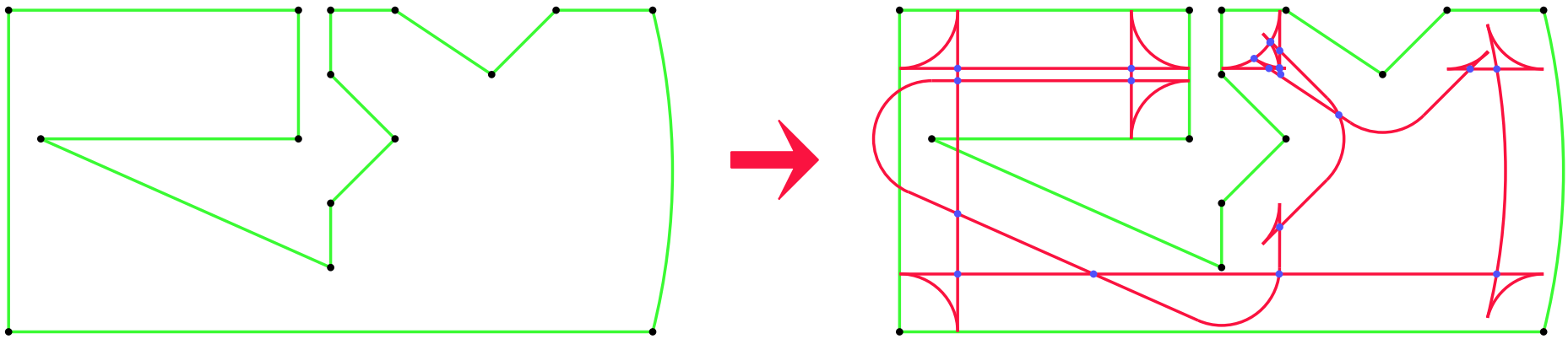
Conventional Offsetting (cont'd)

- In order to get one closed loop, we insert trimming arcs.



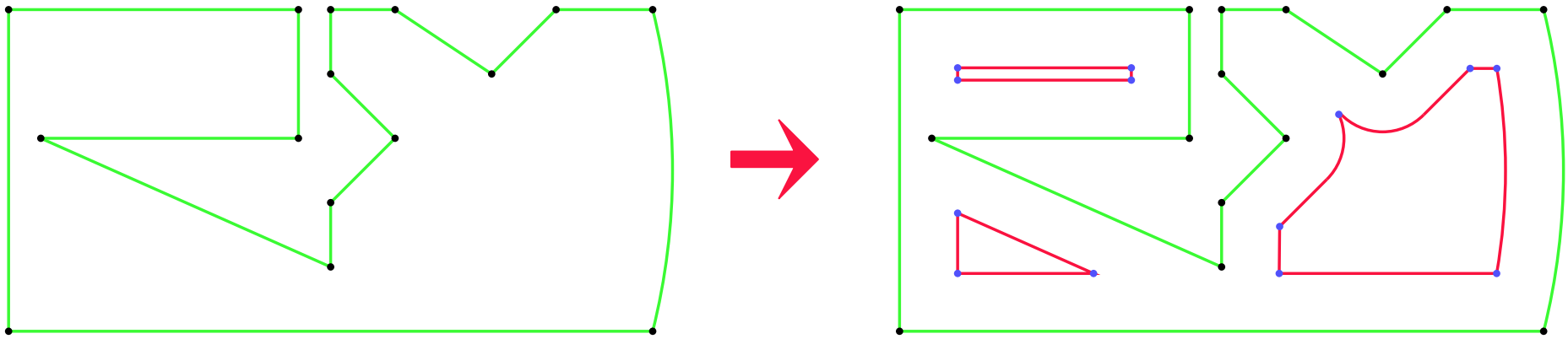
Conventional Offsetting (cont'd)

- Next, all self-intersections are determined.



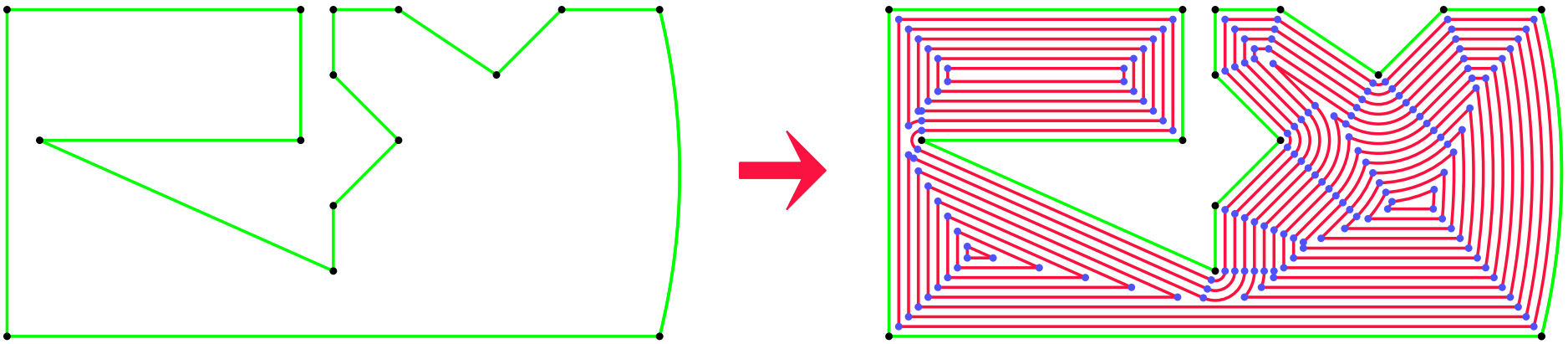
Conventional Offsetting (cont'd)

- Finally, all incorrect loops of the offset are removed.



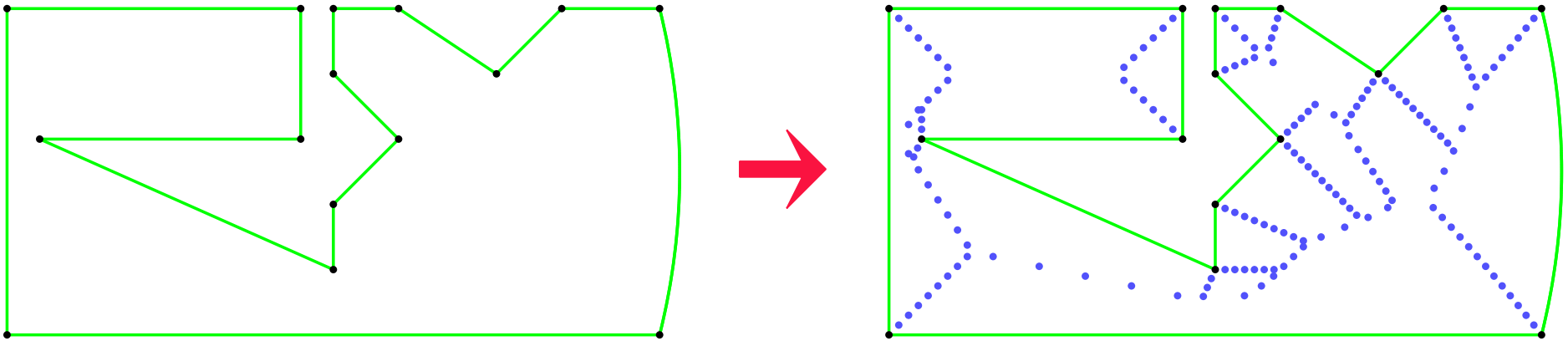
Voronoi-Based Offsetting

- We start with analyzing the positions of the end-points of the offset segments.



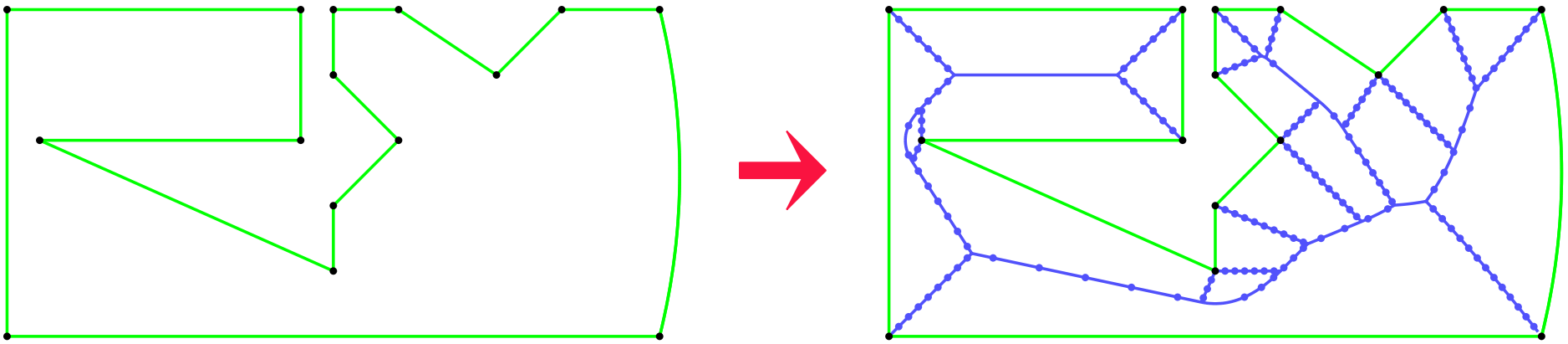
Voronoi-Based Offsetting (cont'd)

- This looks familiar!



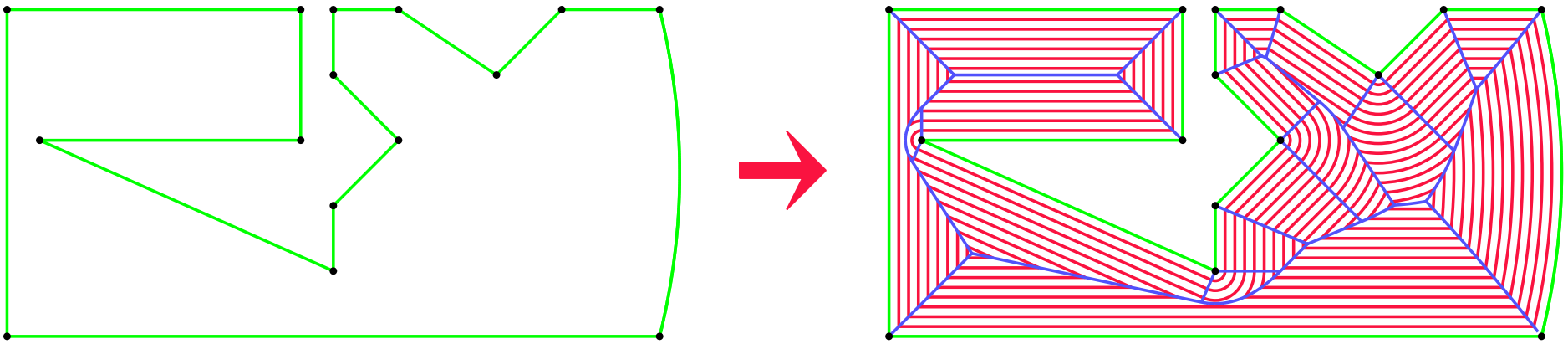
Voronoi-Based Offsetting (cont'd)

- Indeed, all end-points of offset segments lie on the Voronoi diagram!



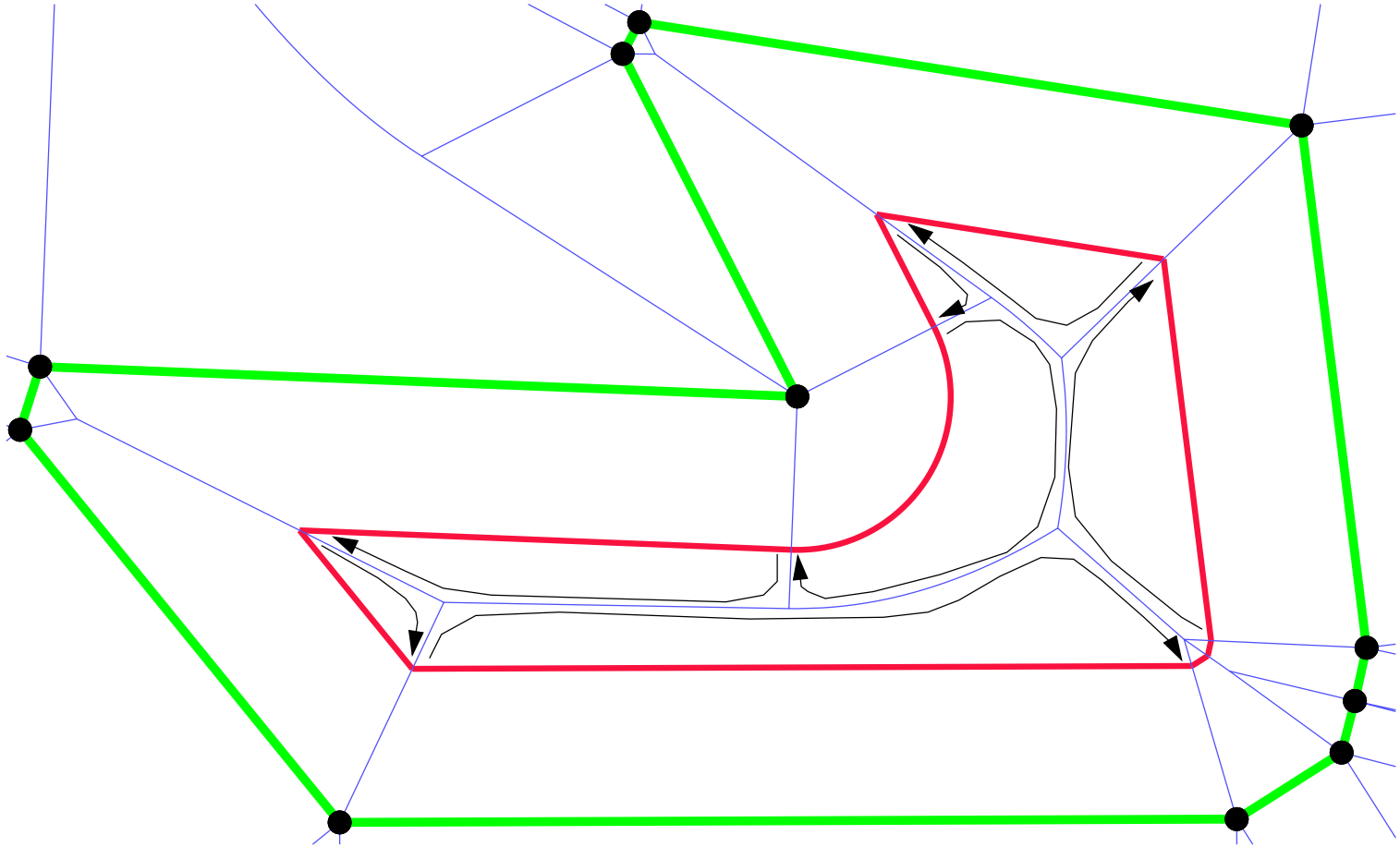
Voronoi-Based Offsetting (cont'd)

- Thus, a linear-time scan of the Voronoi diagram reveals the end-points of one offset.



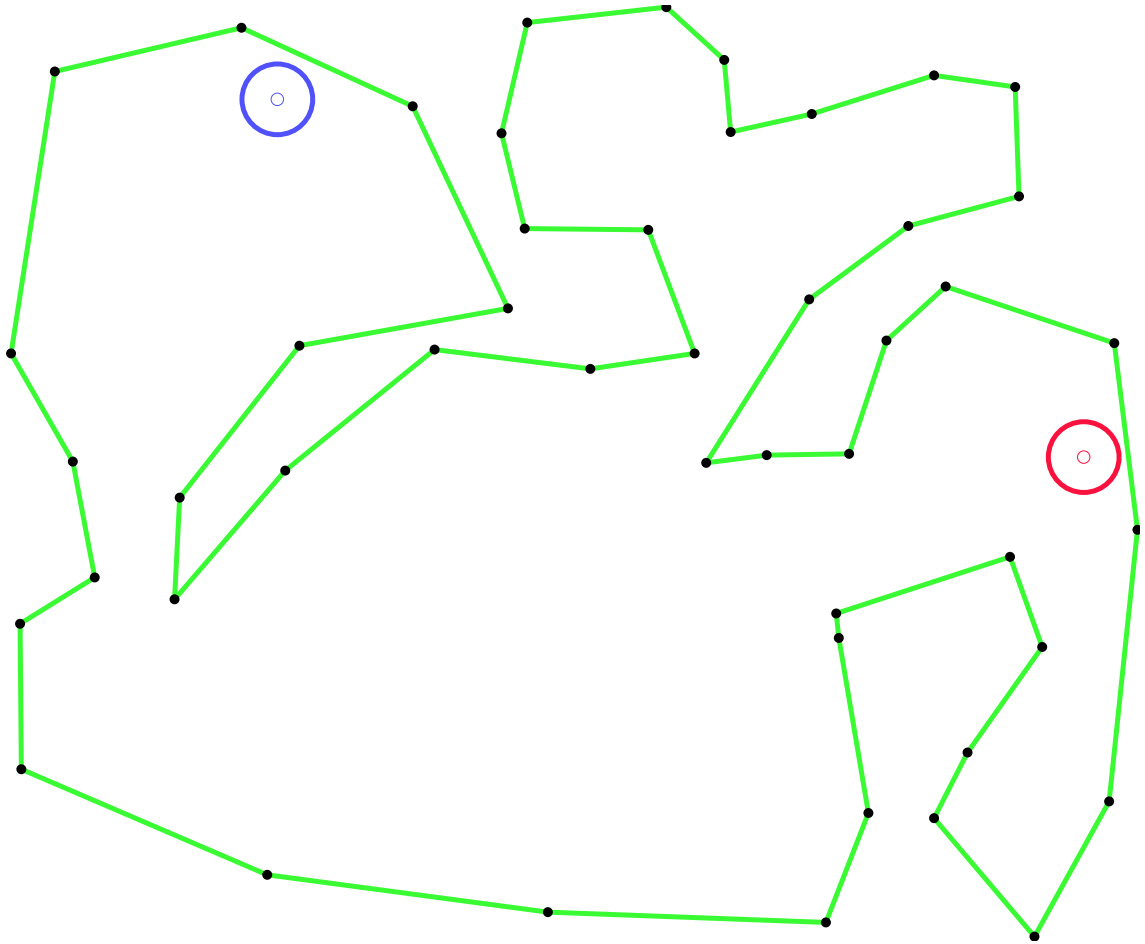
Voronoi-Based Offsetting (cont'd)

- Details of the scan to determine one offset.



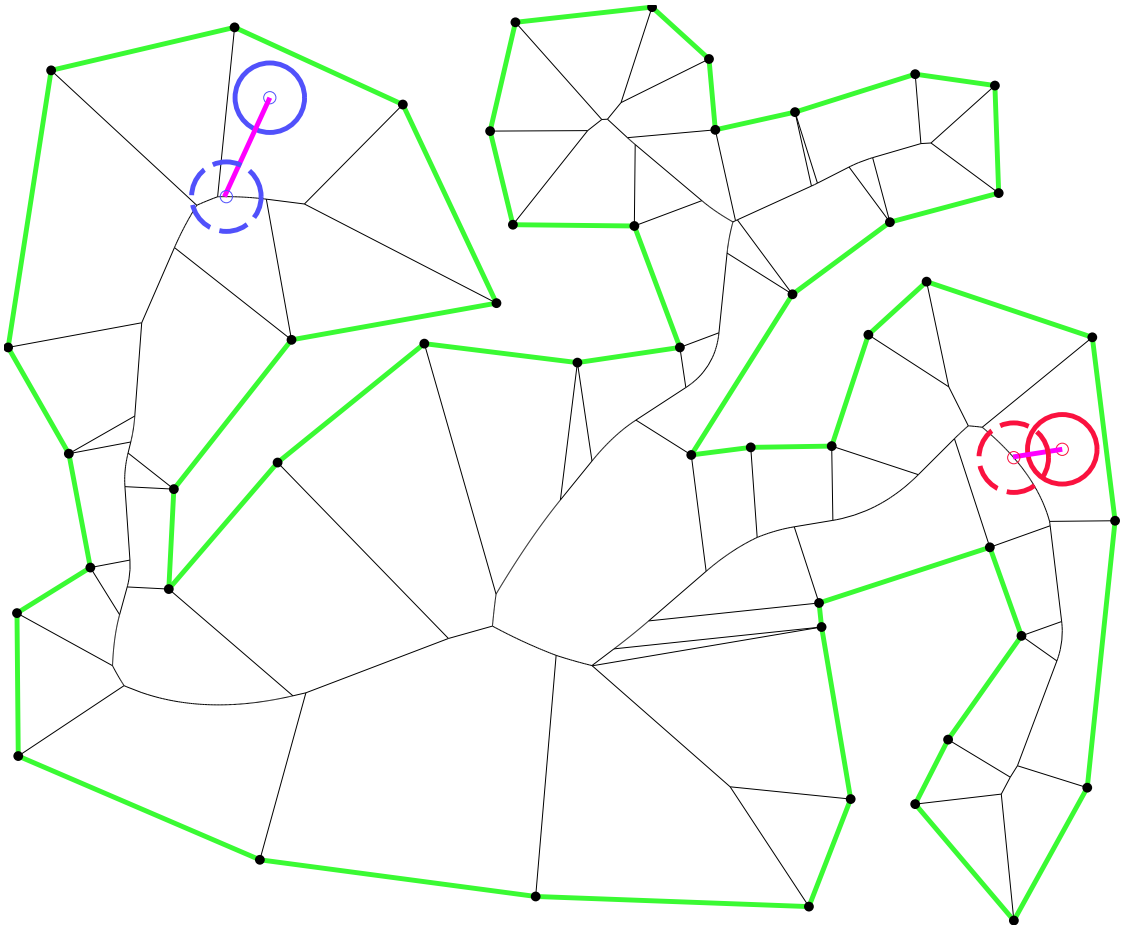
Finding a Gouge-Free Path

- Can we move the disk within the polygon from the blue to the red position?



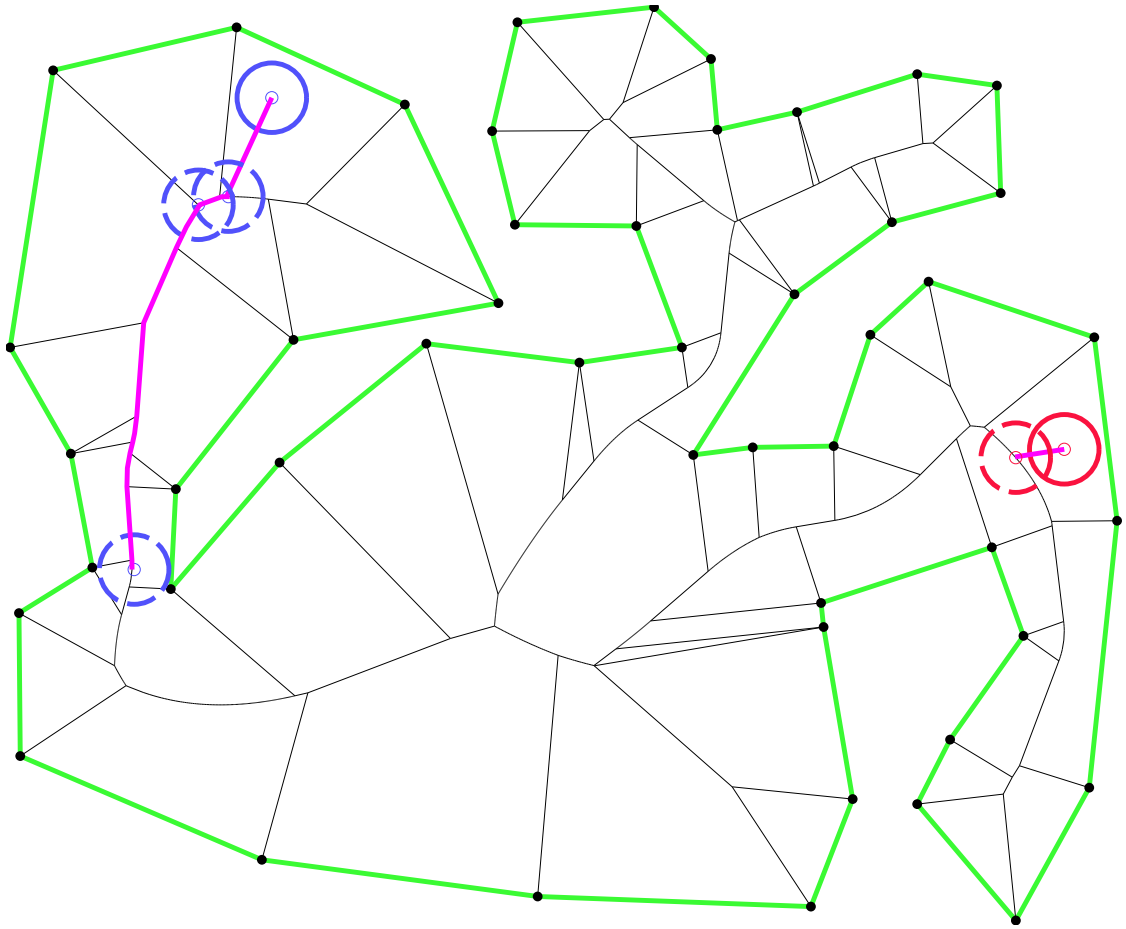
Finding a Gouge-Free Path (cont'd)

- Retraction method: project red and blue centers onto the Voronoi diagram.



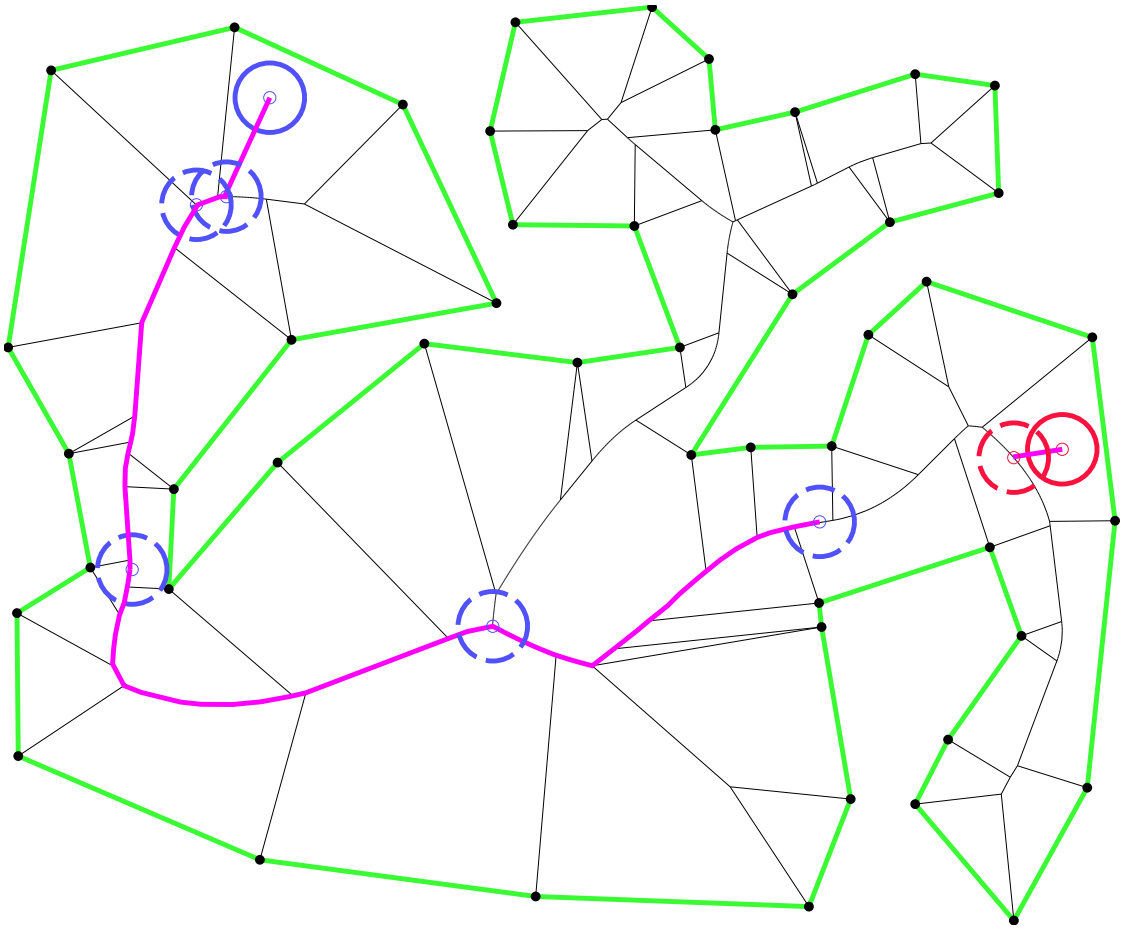
Finding a Gouge-Free Path (cont'd)

- Scan the Voronoi diagram to find a way from blue to red.



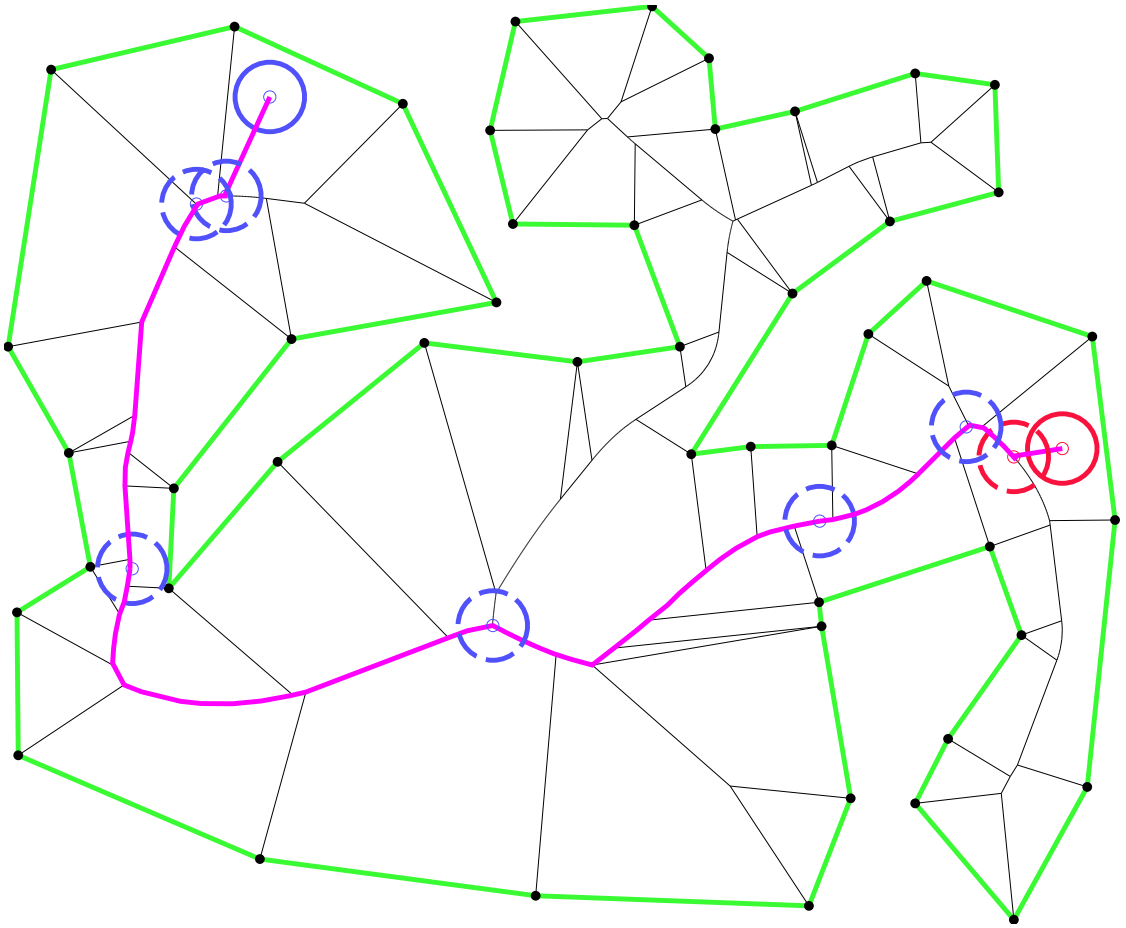
Finding a Gouge-Free Path (cont'd)

- Make sure to check the clearance while moving through a bottleneck.



Finding a Gouge-Free Path (cont'd)

- Indeed, this disk can be moved from blue to red!



Bottlenecks and Locally Inner-Most Points

- A linear-time scan of the VD reveals all bottlenecks and locally inner-most points.