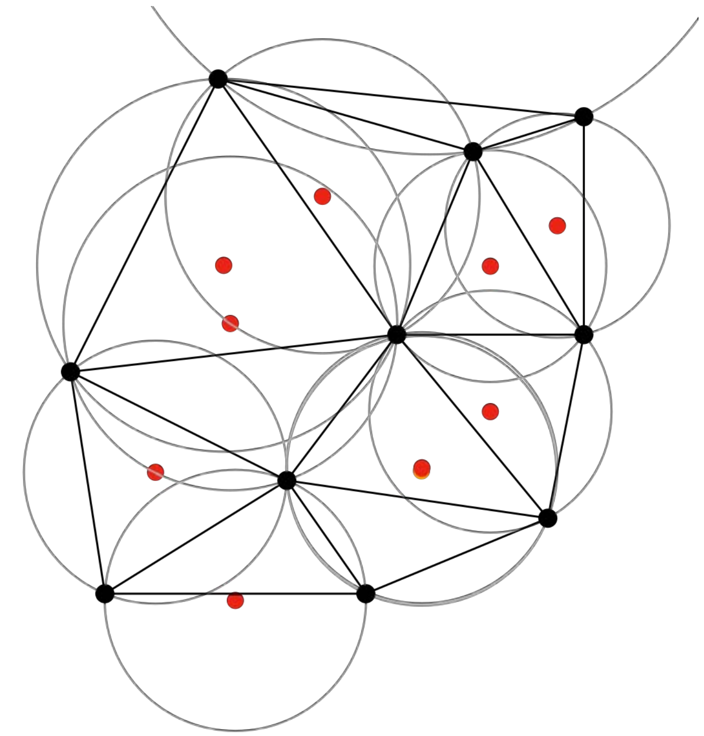
PROXIMITY STRUCTURES IN CGAL

The Computational Geometry Algorithms Library

Manuel Wettstein <manuelwe@inf.ethz.ch>

(Slides by Michael Hoffmann, based on work by Pierre Alliez, Andreas Fabri, Efi Fogel,
Lutz Kettner, Sylvain Pion, Monique Teillaud, Mariette Yvinec, and probably many others.)

GOALS



You can explain what is a ...

- triangulation
- Delaunay triangulation
- Voronoi diagram

and how these concepts are related.

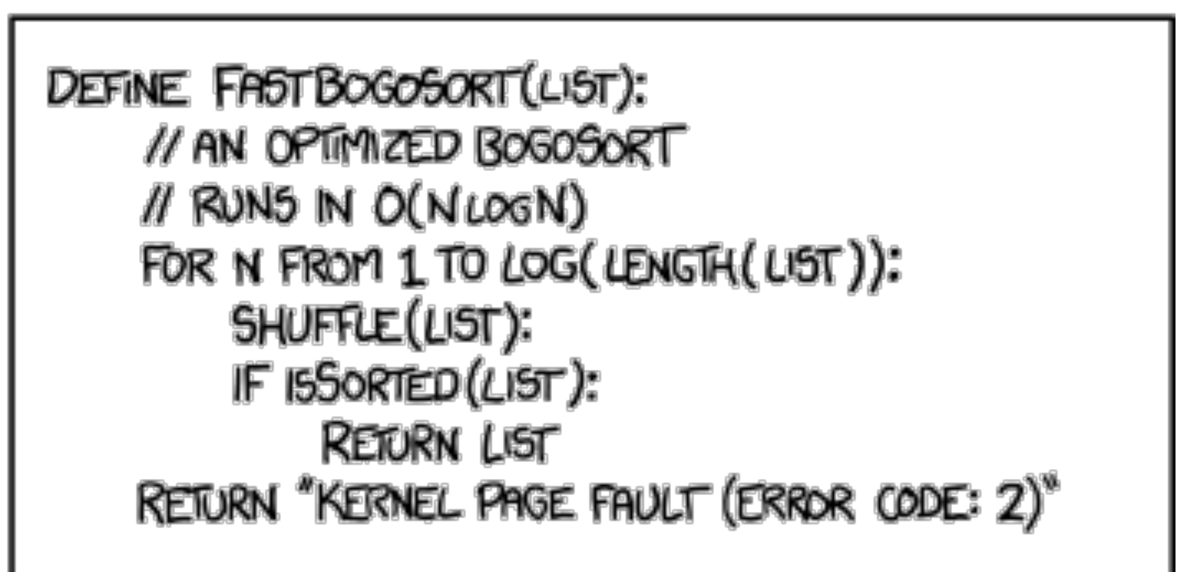
You know how various problems on distances can be solved using DTs/VDs.

You use Ts/DTs/VDs to model problems/ algorithms geometrically, where applicable.

MORE GOALS

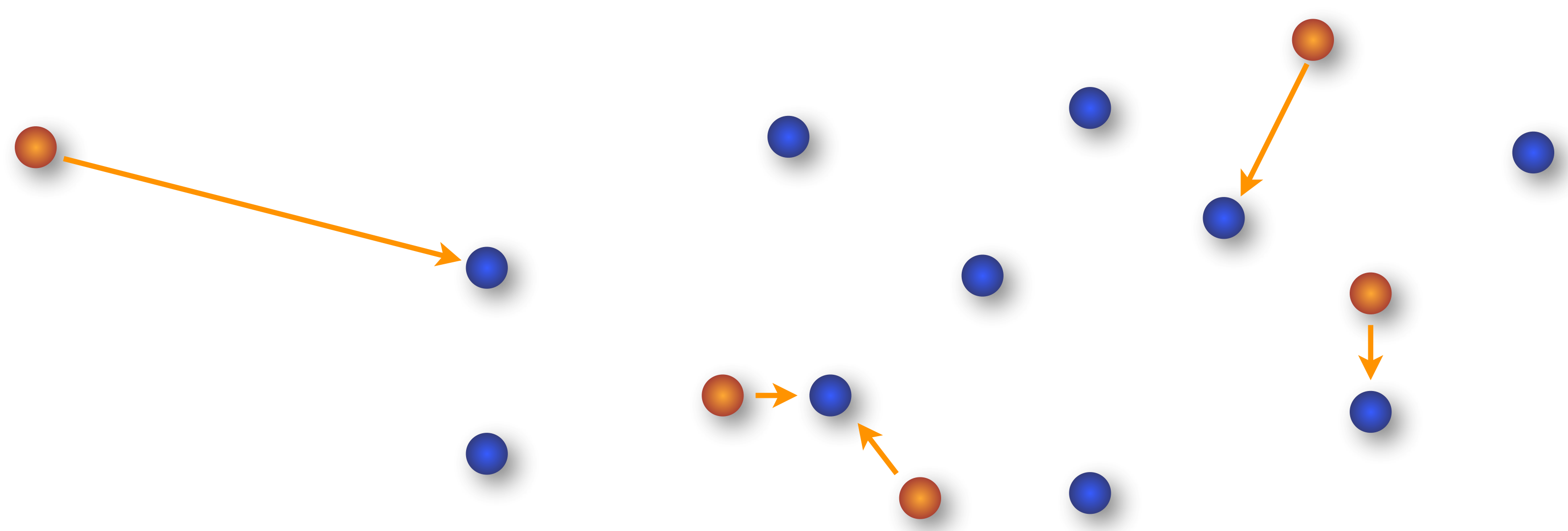
You can design and implement geometric algorithms using DTs and their relatives using CGAL.

You skillfully and creatively combine these geometric techniques with the combinatorial and graph algorithms you know.



http://xkcd.com/1185/

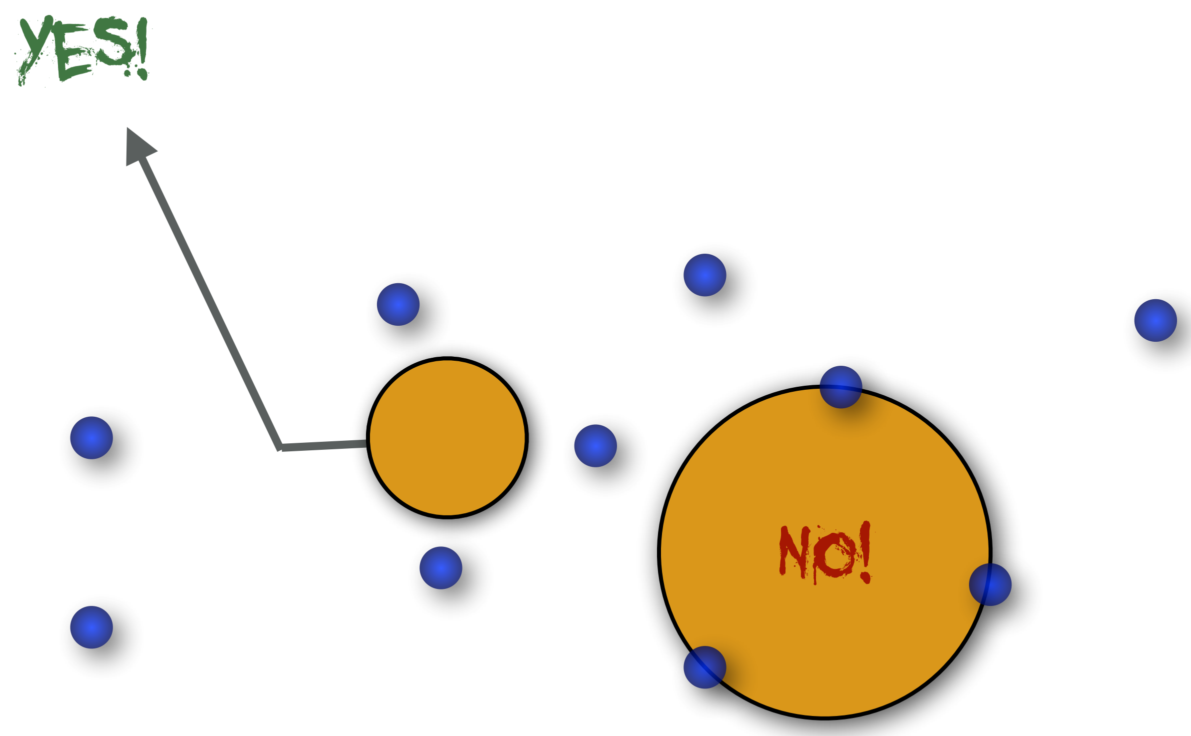
POST OFFICE PROBLEM



Process a set P of n points, s.t. for any given query point q (not necessarily from P) the closest point from P can be found quickly.

→ Don't want to spend time O(n) for every q.

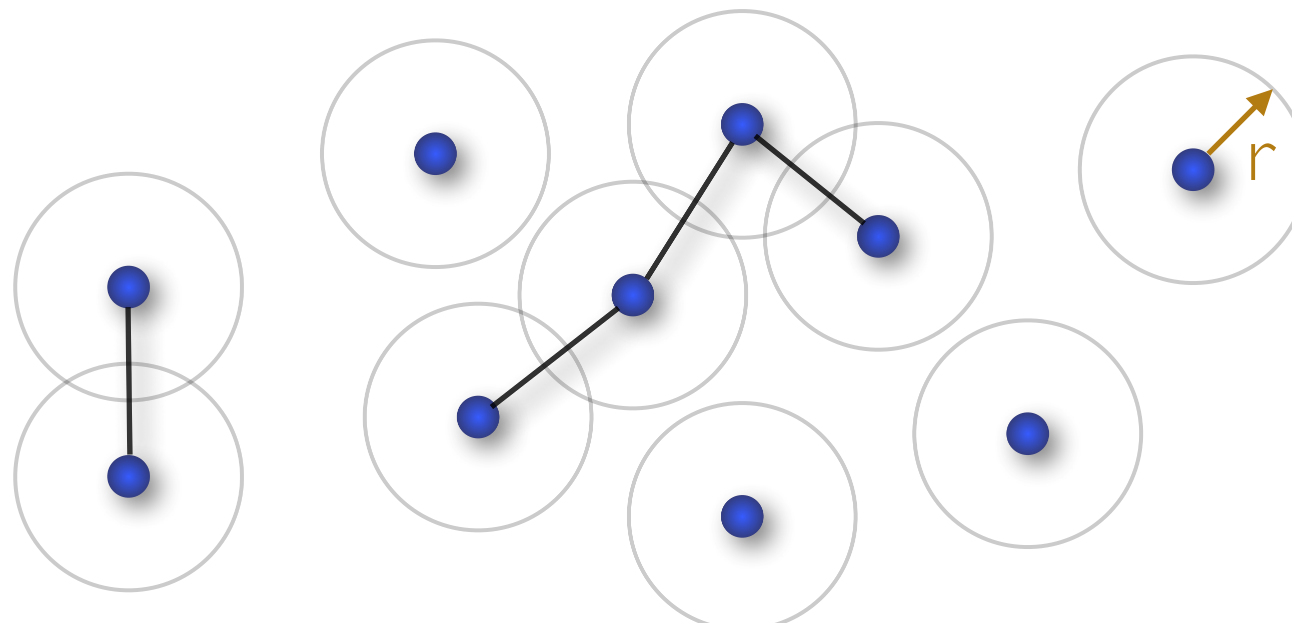
MOTION PLANNING



Decide whether a given disk D can escape from a point set P without ever touching any of the points.

→ Want to handle many different disks D efficiently.

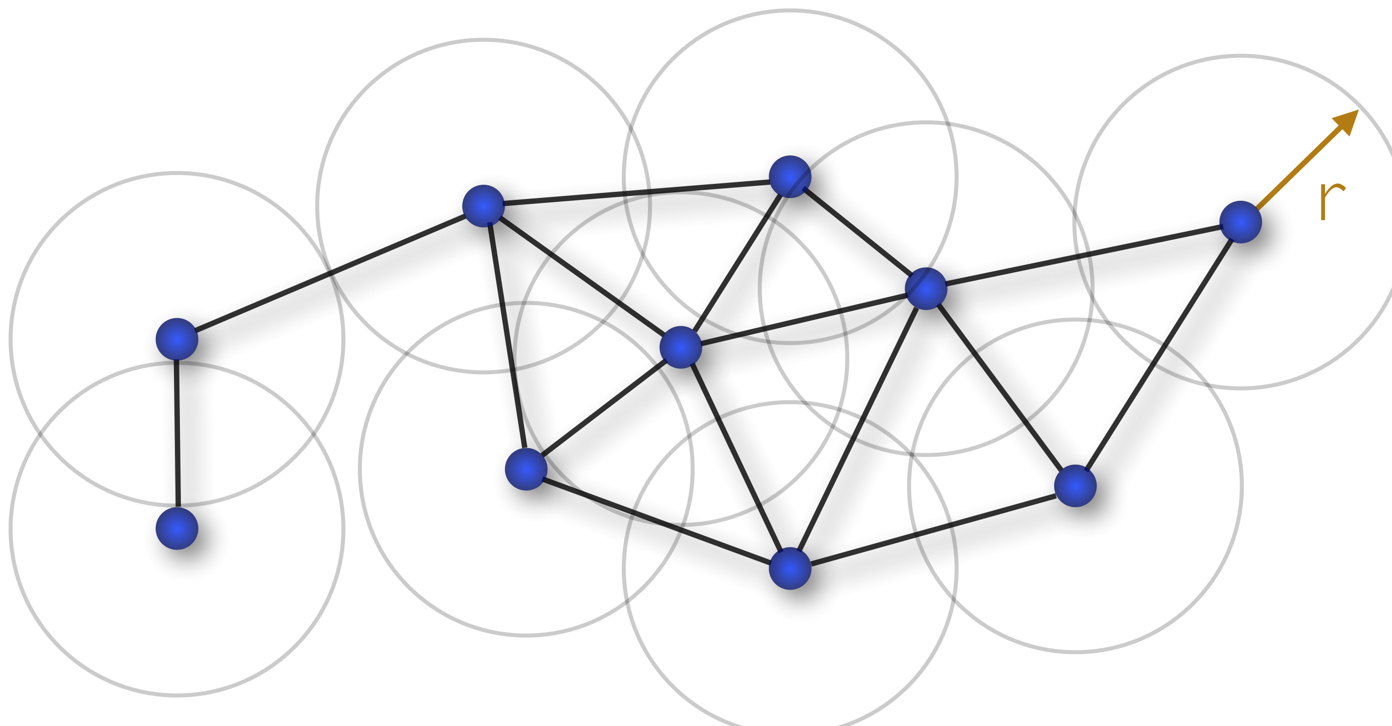
GROWING DISKS



Given a radius r, compute the connected components of the disk intersection graph with vertex set P.

→ Repeat this efficiently for different values of r.

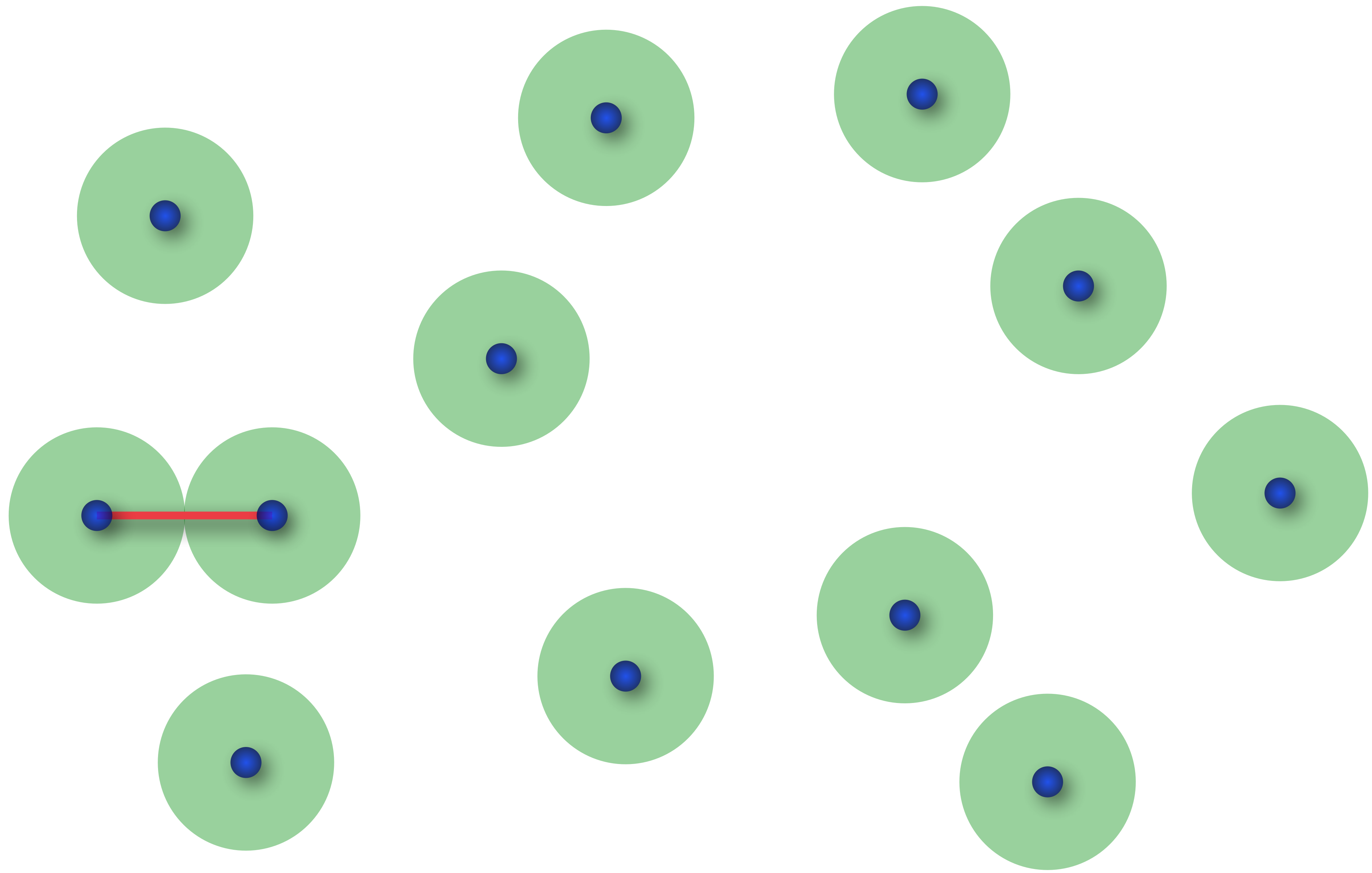
GROWING DISKS



Given a radius r, compute the connected components of the disk intersection graph with vertex set P.

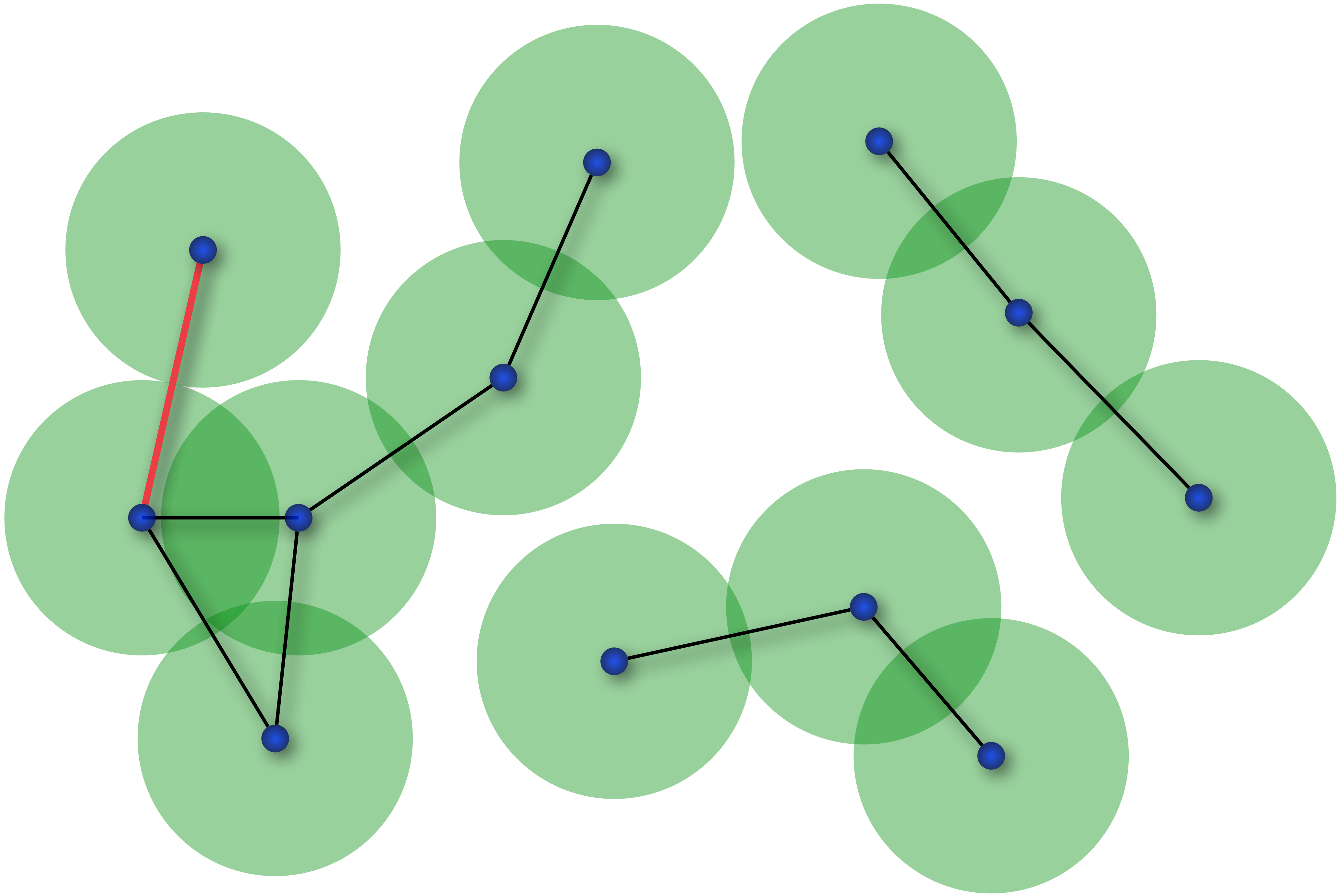
→ Compute special values of r (e.g., smallest r that makes the graph connected).

GROWING DISKS



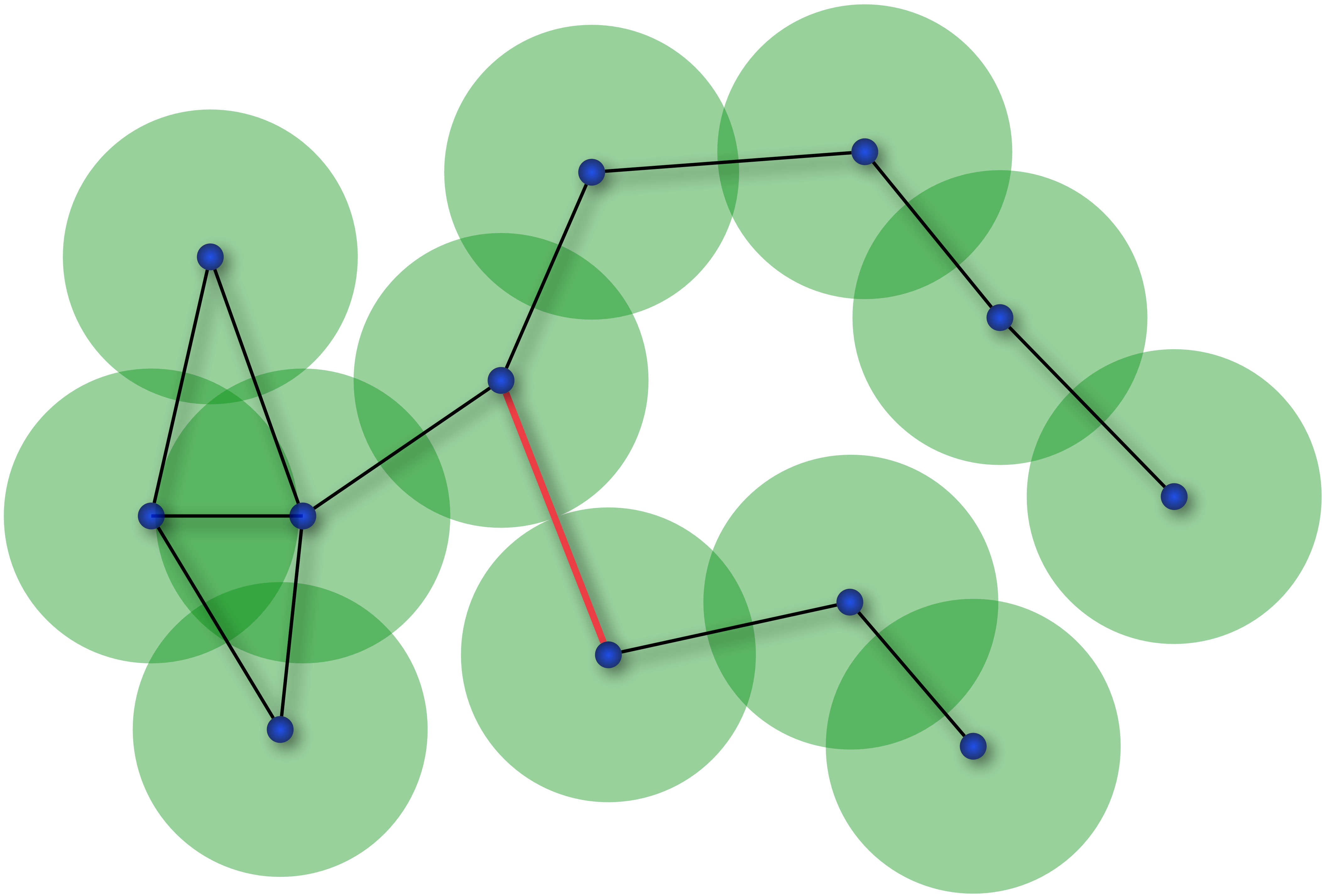
First pair(s) of disks hit: Closest pair.

GROWING DISKS



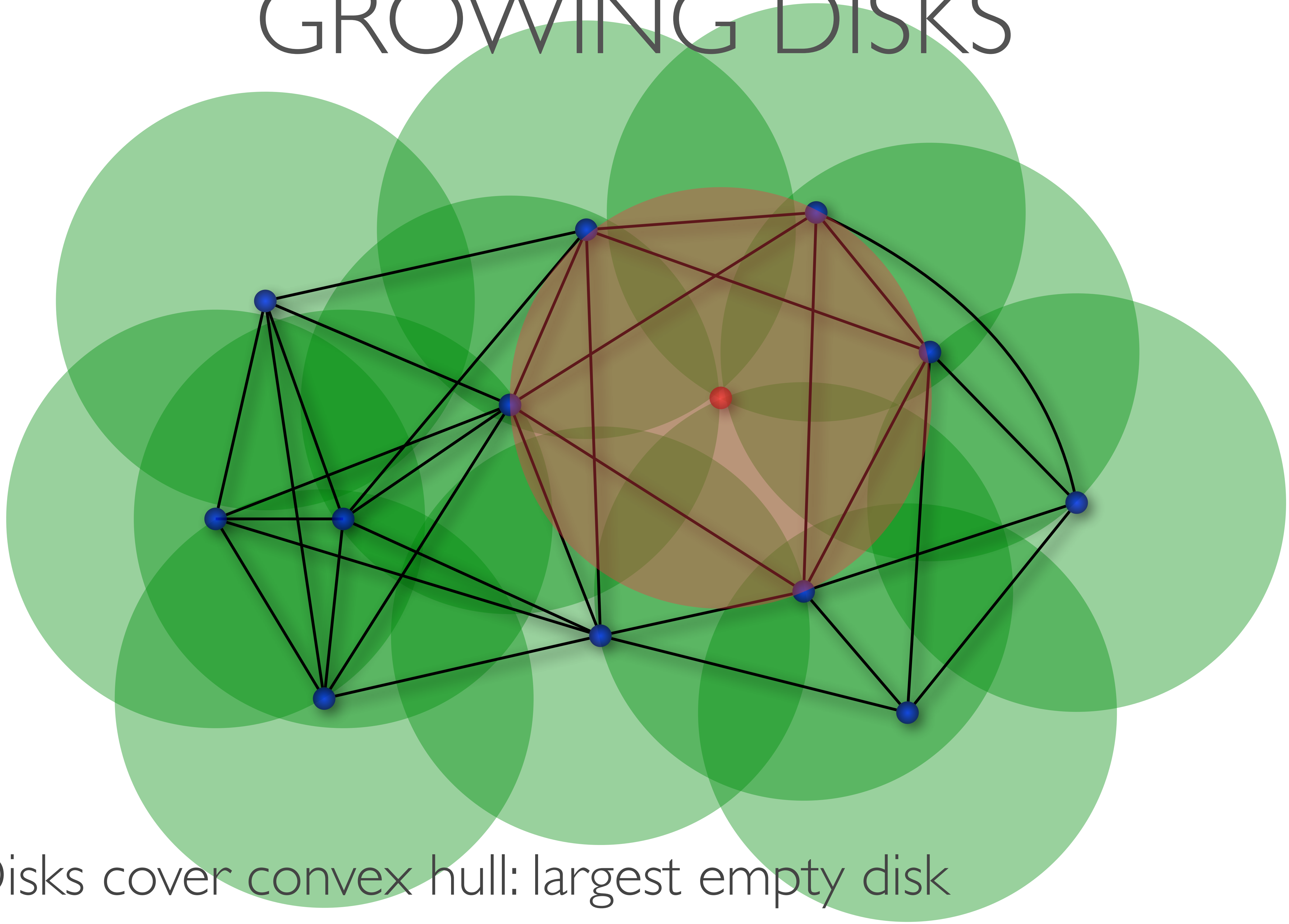
Last disk hits some neighbor: nearest neighbors known.

GROWING DISKS



Disks become connected: EMST known

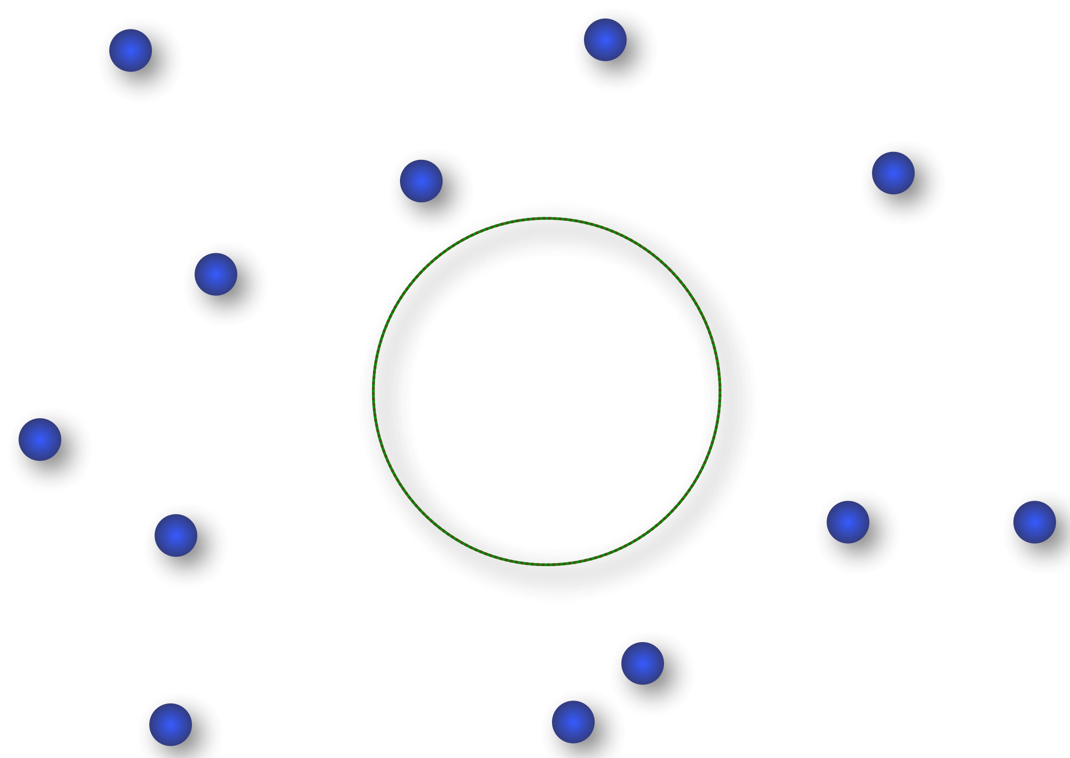
GROWING DISKS



Disks cover convex hull: largest empty disk

WARM-UP: EMPTY DISKS

Given n points, find a large disk that does not contain any.



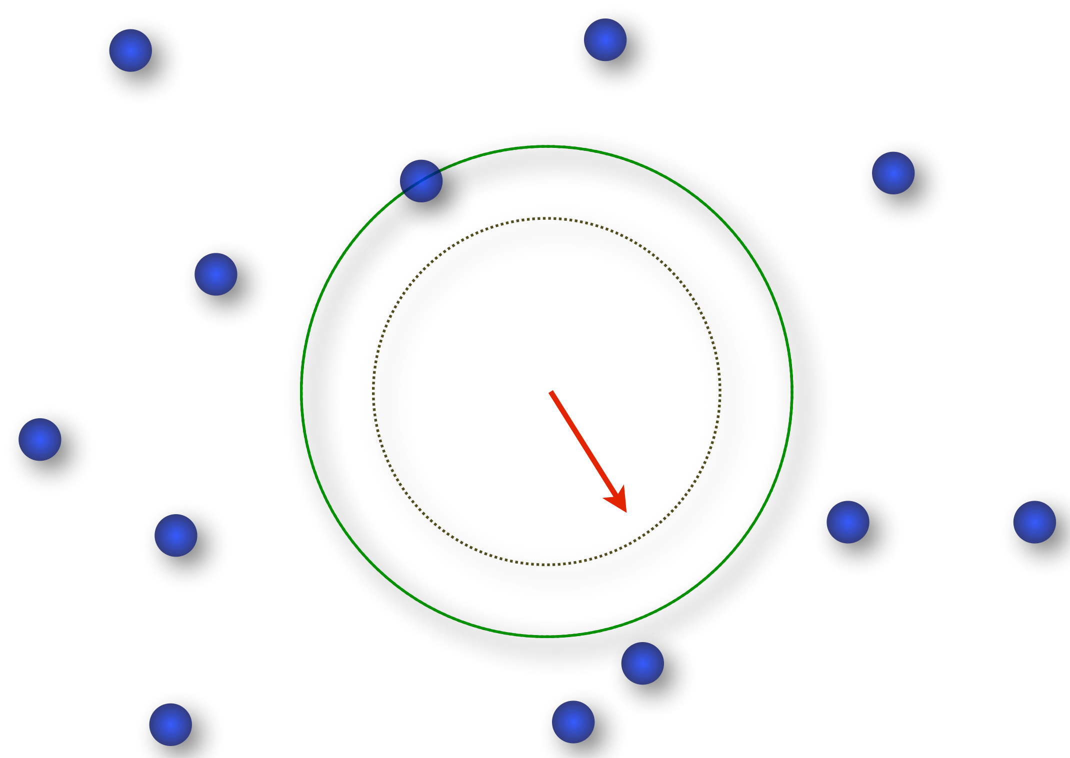
The new disk contains the original disk.

Start with some empty disk.

If there is no point on the boundary, increase the radius.

WARM-UP: EMPTY DISKS

Given n points, find a large disk that does not contain any.

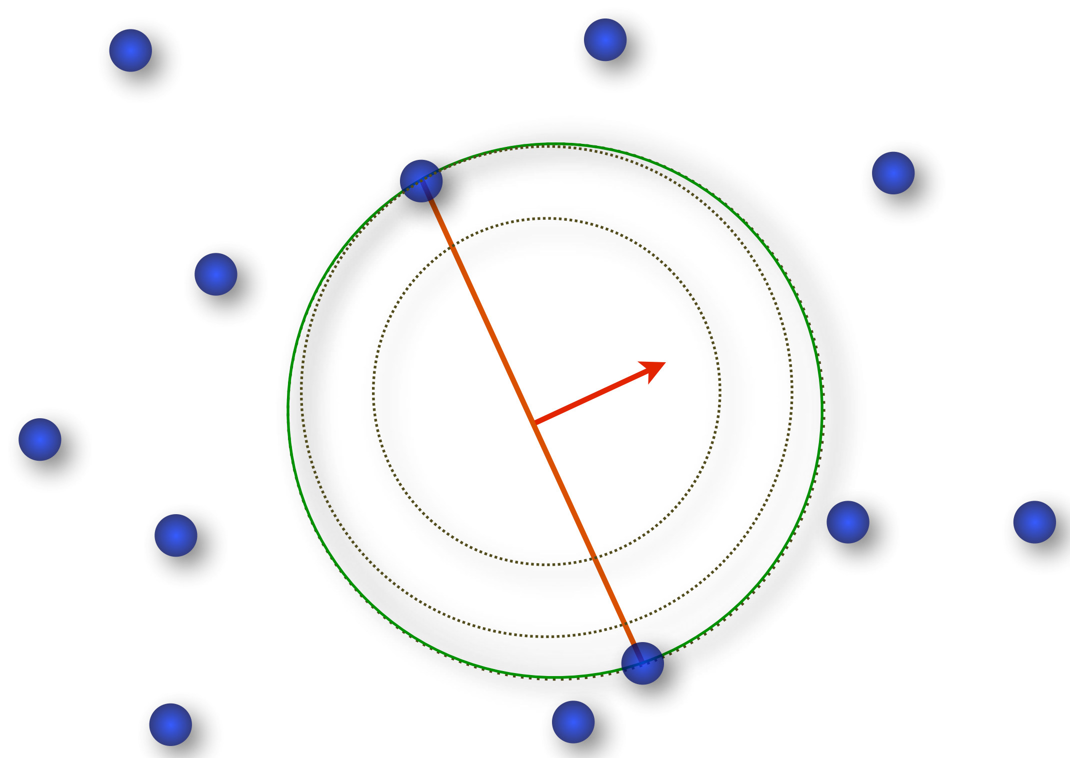


The new disk contains the original disk.

If there is only one point on the boundary, keep it there and move the center away to increase the radius.

WARM-UP: EMPTY DISKS

Given n points, find a large disk that does not contain any.

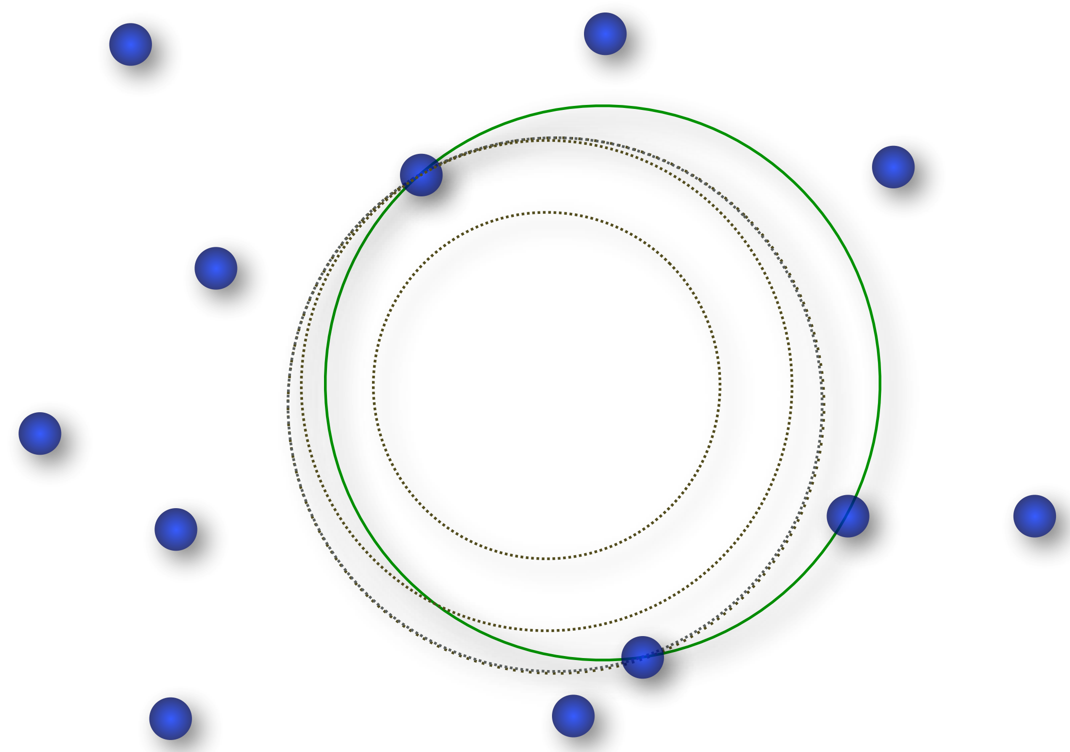


The new disk contains
the original disk.

If there are only two points on the boundary, keep them there and move the center away to increase the radius.

WARM-UP: EMPTY DISKS

Given n points, find a large disk that does not contain any.

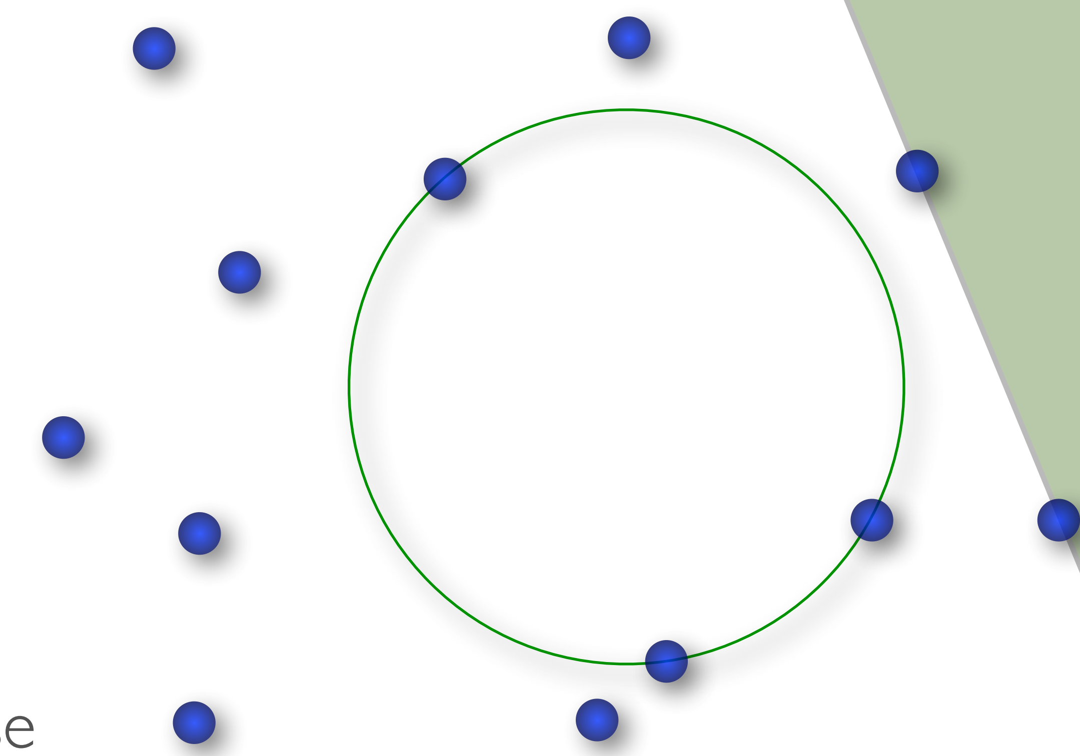


The new disk does not contain the original disk anymore.

If there are only two points on the boundary, keep them there and move the center away to increase the radius.

WARM-UP: EMPTY DISKS

Given n points, find a large disk that does not contain any.

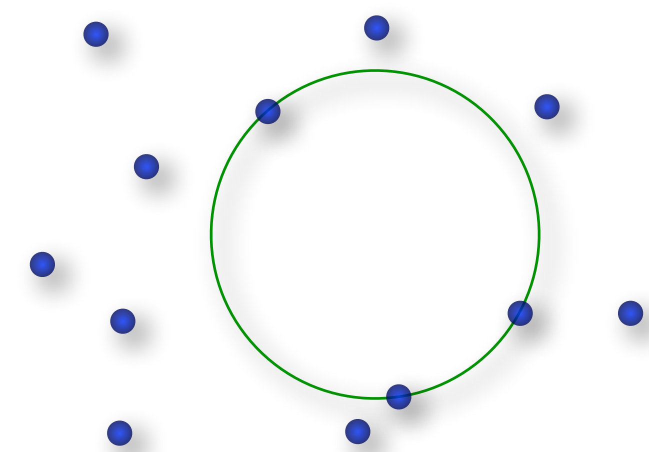


Outside of the convex hull we get huge disks that tend to halfplanes.

Cannot move center infinitesimally and increase radius while staying empty.

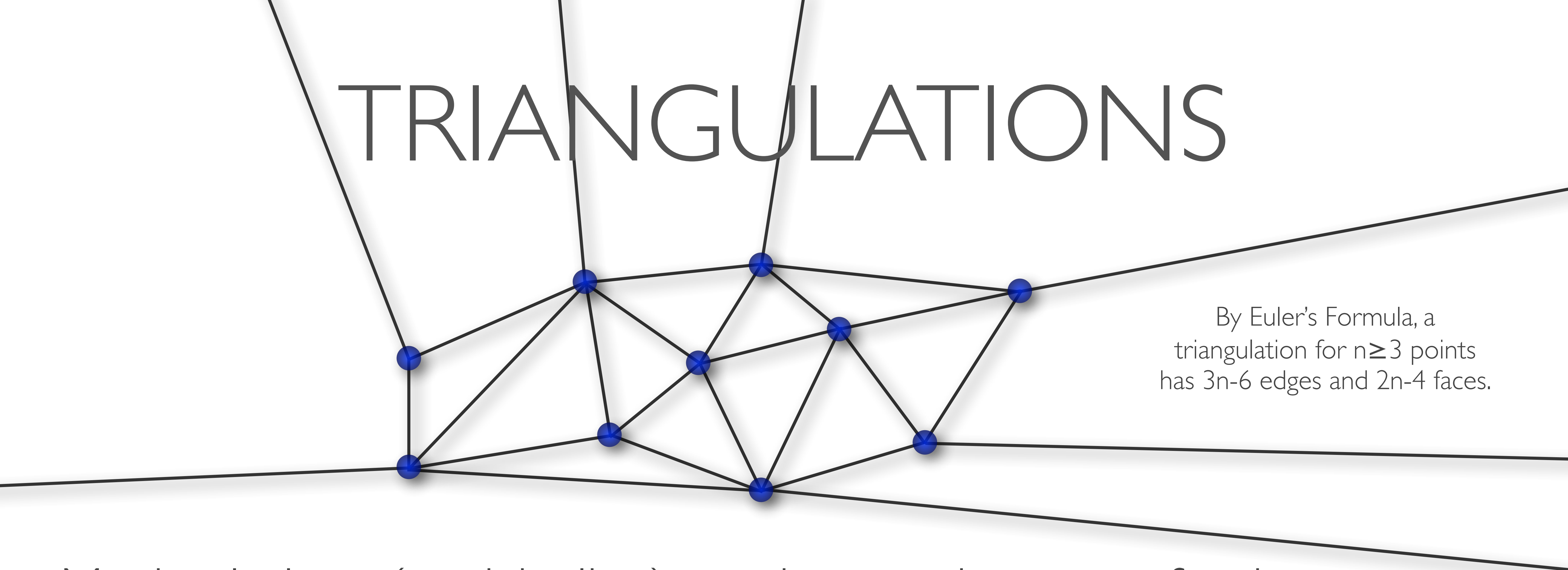
A maximal empty disk passes through three points, if its center is inside the convex hull of the points.

SUMMARY: EMPTY DISKS



- An empty disk of maximal radius passes through three points, if its center is inside the convex hull of the points.
- These maximal empty disks collectively define what is called the **Delaunay triangulation**.
- An inclusion-maximal empty disk passes through two points, if its center is inside the convex hull of the points.

TRIANGULATIONS

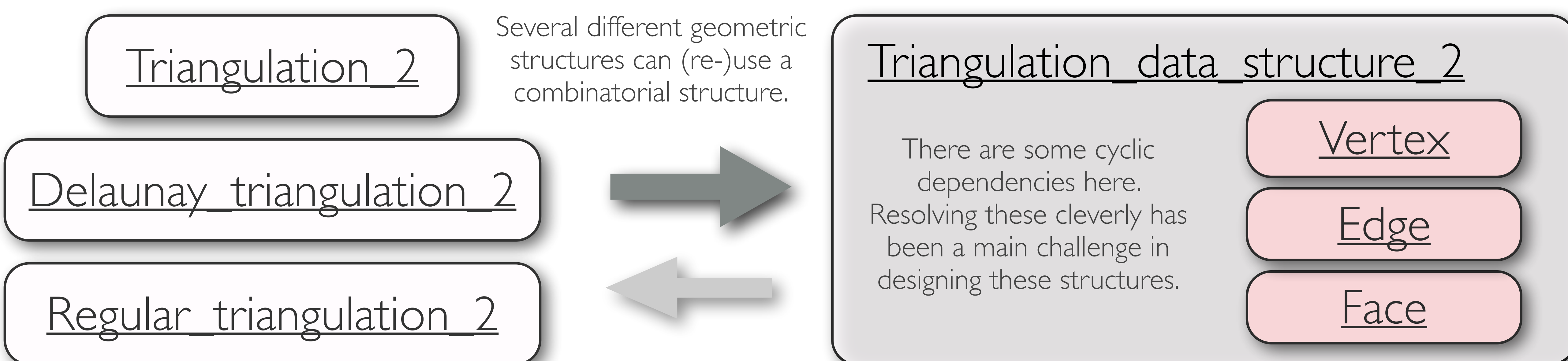


By Euler's Formula, a triangulation for $n \geq 3$ points has $3n-6$ edges and $2n-4$ faces.

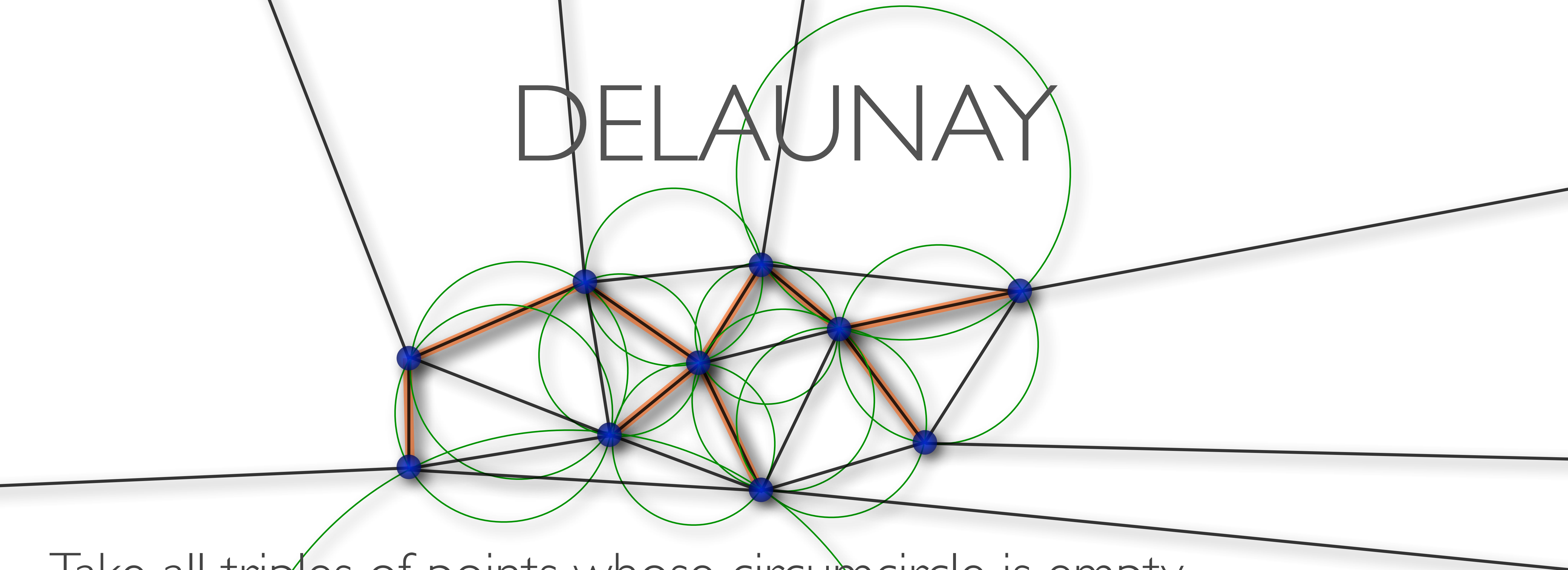
Maximal plane (straight line) graph on a given set of points.

An "infinite vertex" triangulates the exterior of the convex hull.

The combinatorial graph structure is separated from the geometry.



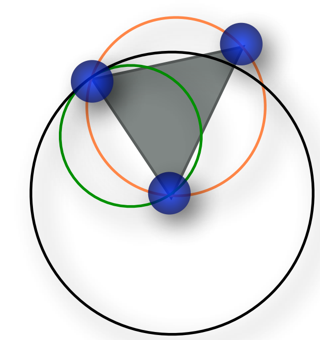
DELAUNAY



Take all triples of points whose circumcircle is empty.

By some magic, this gives a triangulation.

The Delaunay Triangulation has several nice properties:



- It maximizes the smallest angle. Among all triangulations of these points.
- It contains the Euclidean minimum spanning tree and the nearest neighbor graph. Each point has an edge to all closest other points.
- It is unique for points in general position. No three points collinear and no four points cocircular.
- It can be constructed efficiently. O(n log n) in 2D, O(n^2) in 3D.

EMST AND DELAUNAY

Thm. Let P be a set of n points in $\mathbb{R}^2$, and let $e = pq$ be an edge of an EMST for P. Then the circumdisk D_e of e is empty ($D_e \cap P = \{p,q\}$).

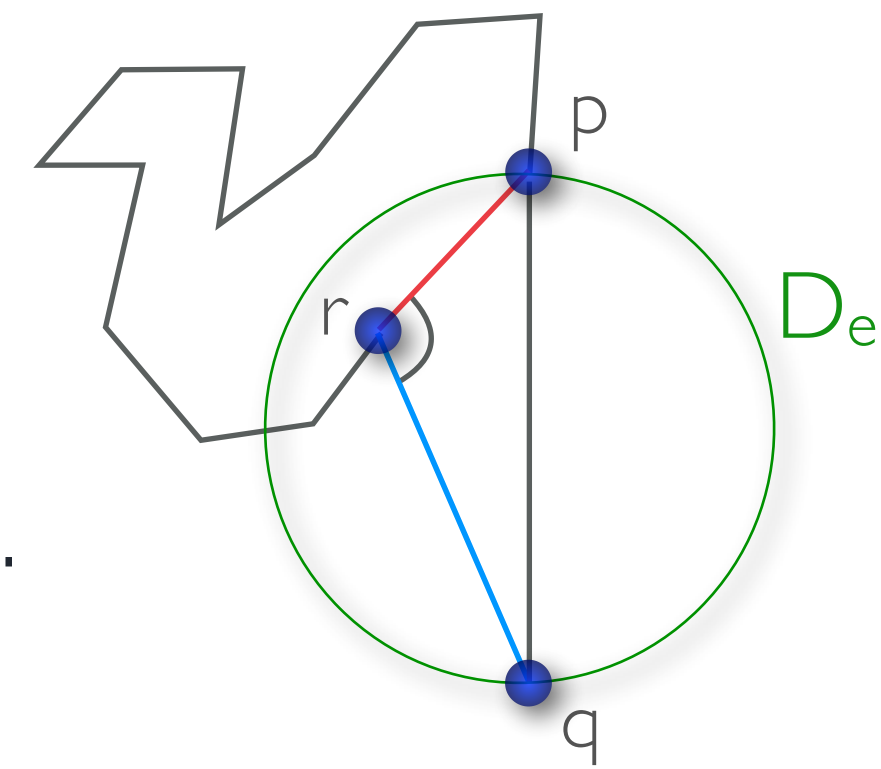
Proof. Suppose to the contrary there is a point $r \in D_e \cap P$.

Then $\measuredangle qrp \geq 90^\circ$ and so $\|rp\|, \|rq\| < \|e\|$.

Removal of e disconnects the tree into two components C_p and C_q s.t. $p \in C_p$ and $q \in C_q$.

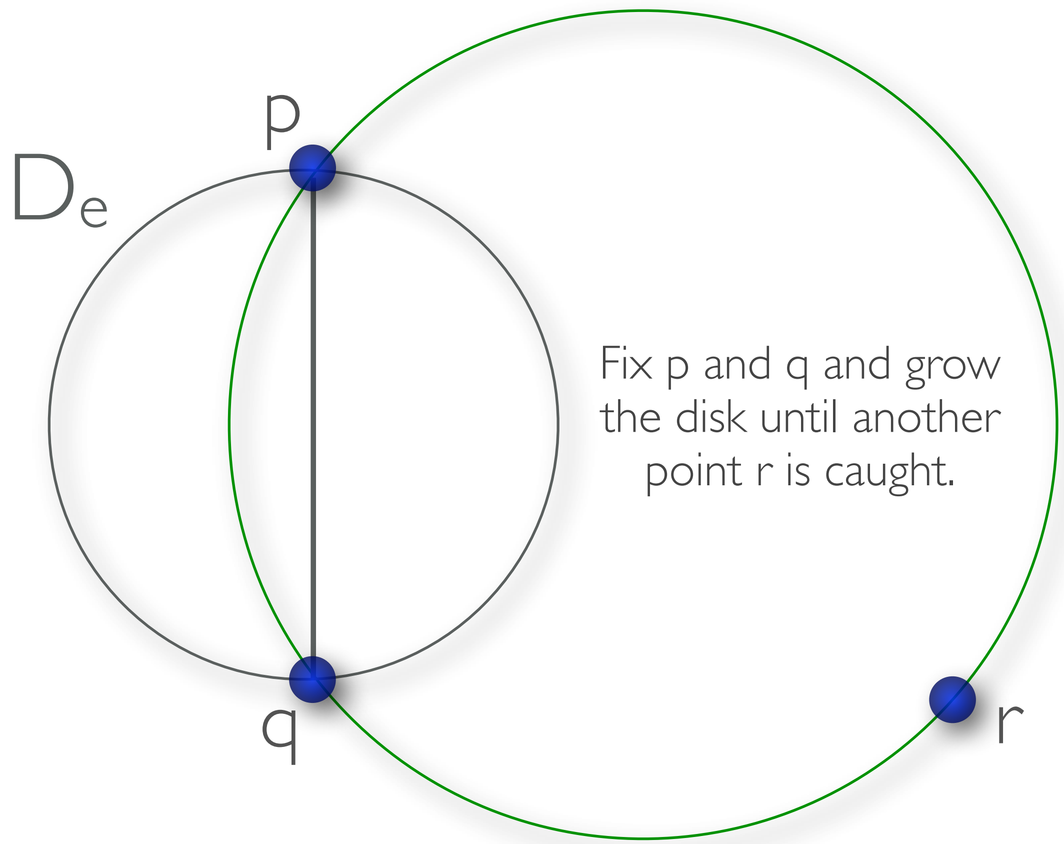
If $r \in C_p$, then add the edge rq to reconnect; else $r \in C_q$ and add the edge rp to reconnect.

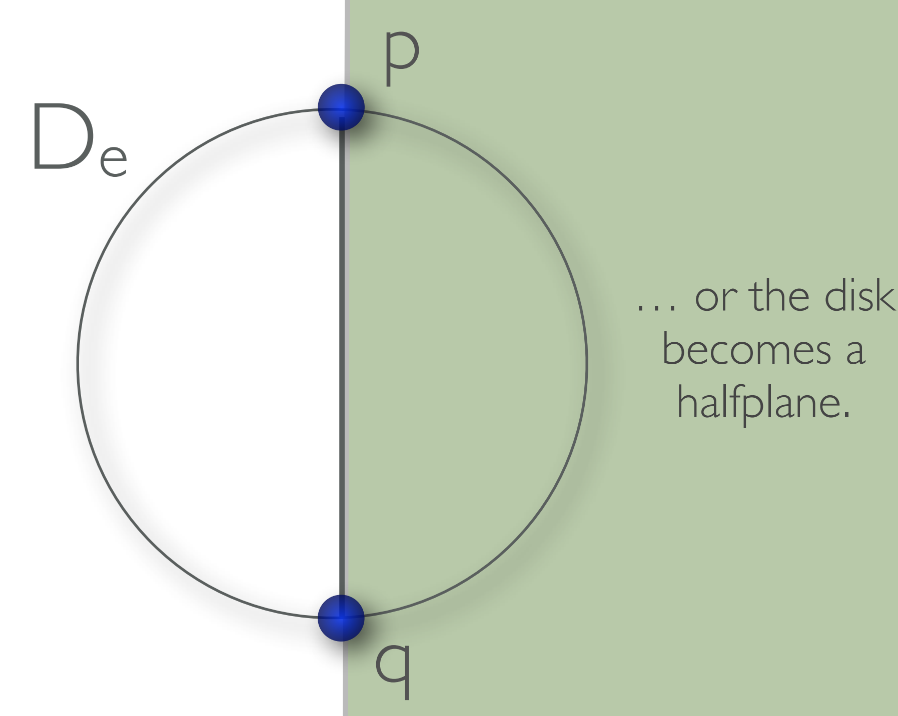
In either case the resulting tree T' is a spanning tree for P of smaller weight than the original. ↯



DELAUNAY EDGES

Obs. Let P be a set of n points in $\mathbb{R}^2$, and let $p, q \in P$ so that the line segment $e = pq$ has an empty circumdisk (that is, $D_e \cap P = \{p,q\}$). Then e is an edge of every Delaunay triangulation of P.





Cor. Let P be a set of n points in $\mathbb{R}^2$, and let e be an edge of an EMST for P. Then e is an edge of every Delaunay triangulation of P.

DELAUNAY TRIANGULATION

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_faces_iterator  Face_iterator;

int main()
{
  // read number of points
  std::size_t n;
  std::cin >> n;
  // construct triangulation
  Triangulation t;
  for (std::size_t i = 0; i < n; ++i) {
    int x, y;
    std::cin >> x >> y;
    t.insert(K::Point_2(x, y));
  }
  // output all triangles
  for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f)
   std::cout << t.triangle(f) << "\n";
}
```

Construction of `Segment_2` or `Triangle_2` from points is trivial. => No exact constructions needed.

We do not want to output the infinite faces outside the convex hull. Otw, use `All_faces_iterator`...

To get edges instead, replace `Face` by `Edge` and `faces` by `edges` everywhere, and use `t.segment(...)` instead of `t.triangle(...)`.

Here not `*f`! The triangulation interface is based on so-called *handles*. These are an abstraction of pointers. Think of them as something that can be dereferenced to yield (in this case) a `Triangulation::Face`. In particular, iterators (like f here) convert to the corresponding handles.

The corresponding type is called `Triangulation::Face_handle`.

DELAUNAY TRIANGULATION

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
typedef Triangulation::Finite_faces_iterator  Face_iterator;

int main()
{
  // read number of points
  std::size_t n;
  std::cin >> n;
  // construct triangulation
  Triangulation t;
  for (std::size_t i = 0; i < n; ++i) {
    int x, y;
    std::cin >> x >> y;
    t.insert(K::Point_2(x, y));
  }
  // output all triangles
  for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f)
   std::cout << t.triangle(f) << "\n";
}
```

This works, but inserting the points one by one is dangerous in terms of efficiency, as the performance of the triangulation depends on the insertion order. A (sufficiently uniform) random order yields an expected runtime of O(n log n), but there are point sets that have bad orders for which the runtime becomes quadratic…

DELAUNAY TRIANGULATION

```
...

int main()
{
  ...

  // read points
  std::vector<K::Point_2> pts;
  pts.reserve(n);
  for (std::size_t i = 0; i < n; ++i) {
    int x, y;
    std::cin >> x >> y;
    pts.push_back(K::Point_2(x, y));
  }
  // construct triangulation
  Triangulation t;
  t.insert(pts.begin(), pts.end());

  ...
}
```

A safe strategy is to let the triangulation choose a suitable insertion order: Instead of inserting points one by one using `t.insert(p)`, insert a whole (iterator) range [b,e) of points using `t.insert(b,e)`.

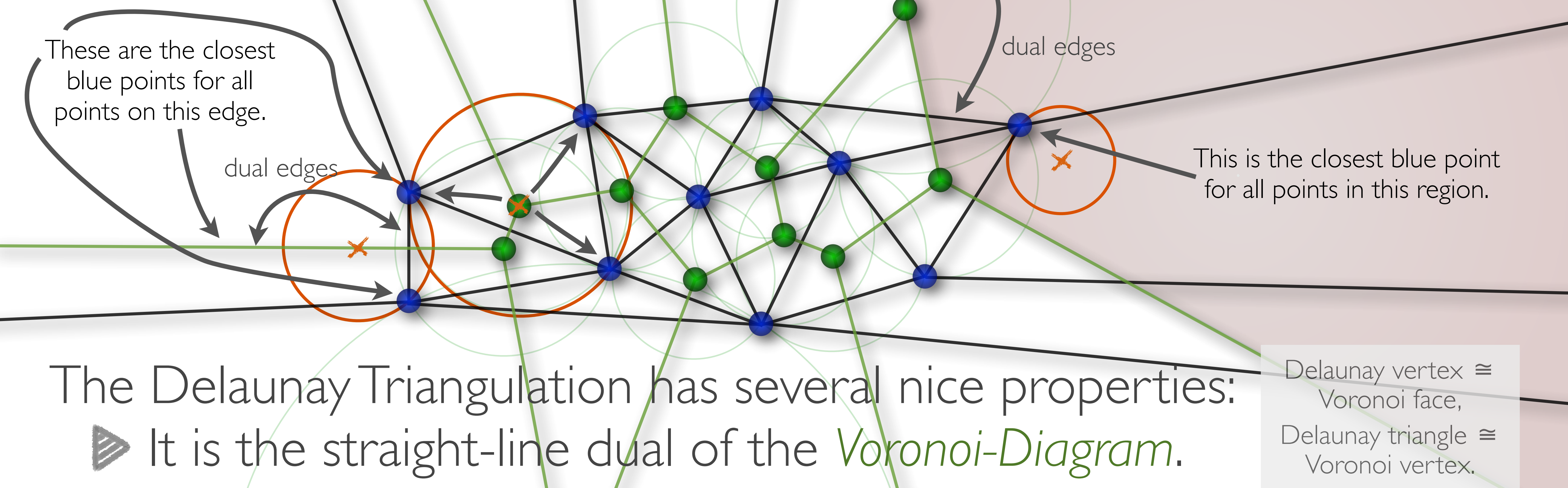
Here the input points are first read into a vector and then inserted as a whole into the triangulation.

Internally, the range insertion uses `CGAL::spatial_sort()` to determine a good insertion order.

This function is generally useful to speedup batch processing, for instance, when localizing many points in a triangulation...

NB: Watch out in case of duplicate input points: These are inserted once only. (The points of a triangulation form a set, not a multiset.)

DELAUNAY / VORONOI



The Delaunay Triangulation has several nice properties:

- It is the straight-line dual of the *Voronoi-Diagram*.

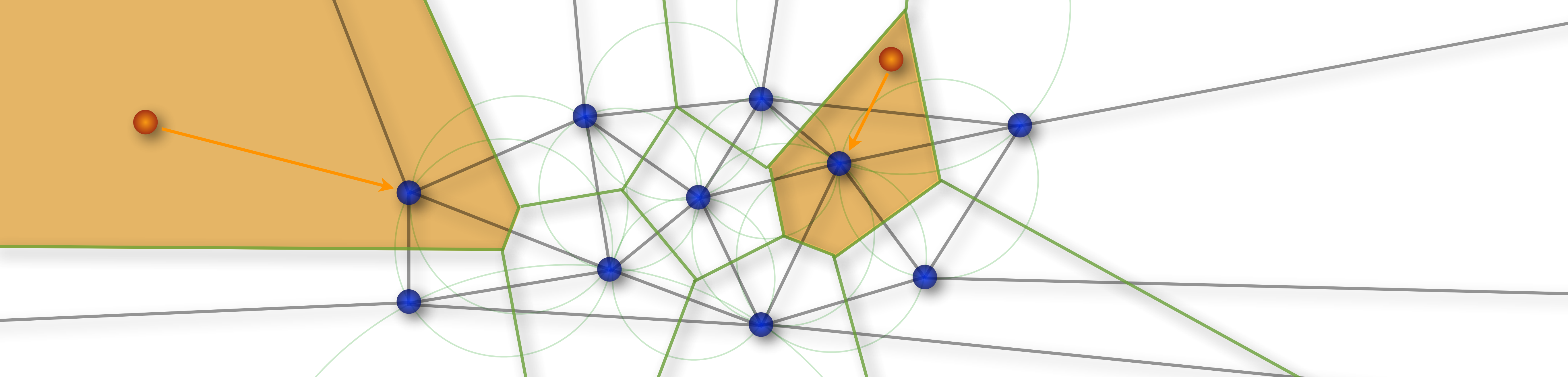
The Voronoi-Diagram for a set P of points partitions the plane into regions for which the closest point from P is the same.

For points ...

- in the interior of a Voronoi region, there is one closest point from P;
- in the relative interior of a Voronoi edge, there are two closest points from P;
- on a Voronoi vertex, there are three (or more) closest points from P.

A Delaunay edge is a convex hull edge iff its dual Voronoi edge is a ray.

DELAUNAY / VORONOI



Post Office Problem:
Process a set P of n points, s.t. for any given query point q (not necessarily from P) the closest point from P can be found quickly.

→ Find Voronoi region that contains q.

Consider this as an operation of complexity O(log n), where n = #vertices in the Delaunay triangulation.

The Delaunay triangulation offers `t.nearest_vertex()`, which often is much more efficient than computing the Voronoi diagram.

Why? Because it uses predicates only...

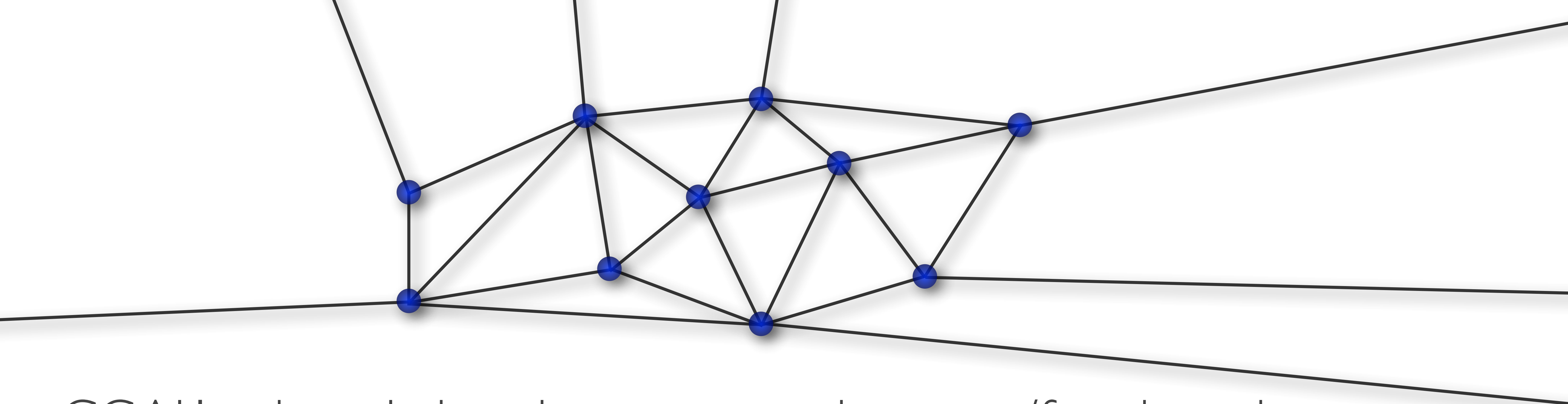
VORONOI DIAGRAMS

There is an explicit Voronoi adaptor in CGAL.
But for our purposes, we can extract all information needed from the Delaunay triangulation.

```
...
// process all Voronoi vertices
for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f) {
  K::Point_2 p = t.dual(f);
  ...
}
...
```

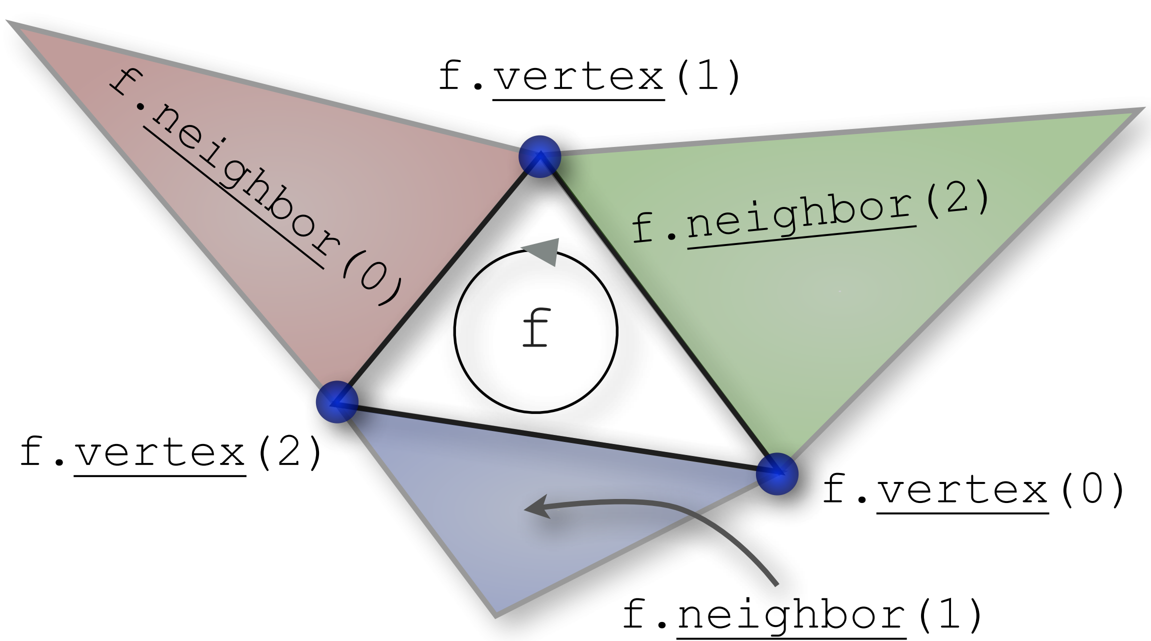
Attention! Construction...

TRIANGULATION DATA STRUCTURE



CGAL's triangulation data structure is vertex/face based.
Edges are represented implicitly only.

Similarly in 3D it is vertex/cell based.



Space consumption is ~12n.

Geometric information is stored at vertices: each vertex has a `.point()` member function.

EDGE REPRESENTATION

Edges in CGAL::Triangulation_data_structure_2 are represented as a std::pair<Face_handle,int>.

A pair (f,i) represents the i-th edge along the boundary of *f.

$0 <= i < 3$

The edge connects the vertices (i+1)%3 and (i+2)%3 of *f.

Therefore, we can obtain the vertices of an edge as follows:

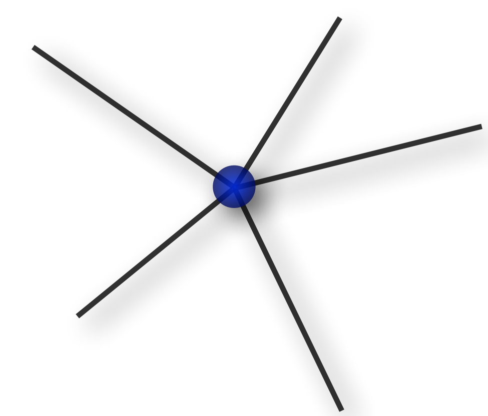
```
...
Triangulation::Edge e;
...
// get the vertices of e
Triangulation::Vertex_handle v1 = e.first->vertex((e.second + 1) % 3);
Triangulation::Vertex_handle v2 = e.first->vertex((e.second + 2) % 3);
std::cout << "e = " << v1->point() << " <-> " << v2->point() << std::endl;
...
```

If we wanted these points only, t.segment(e) would have done it. But if we need the vertices...

CIRCULATORS

… are like iterators, but for circular rather than linear structures.

For instance, the circular sequence of edges incident to a vertex in a triangulation.



For a circulator c, the range [c,c) denotes the full circular sequence.

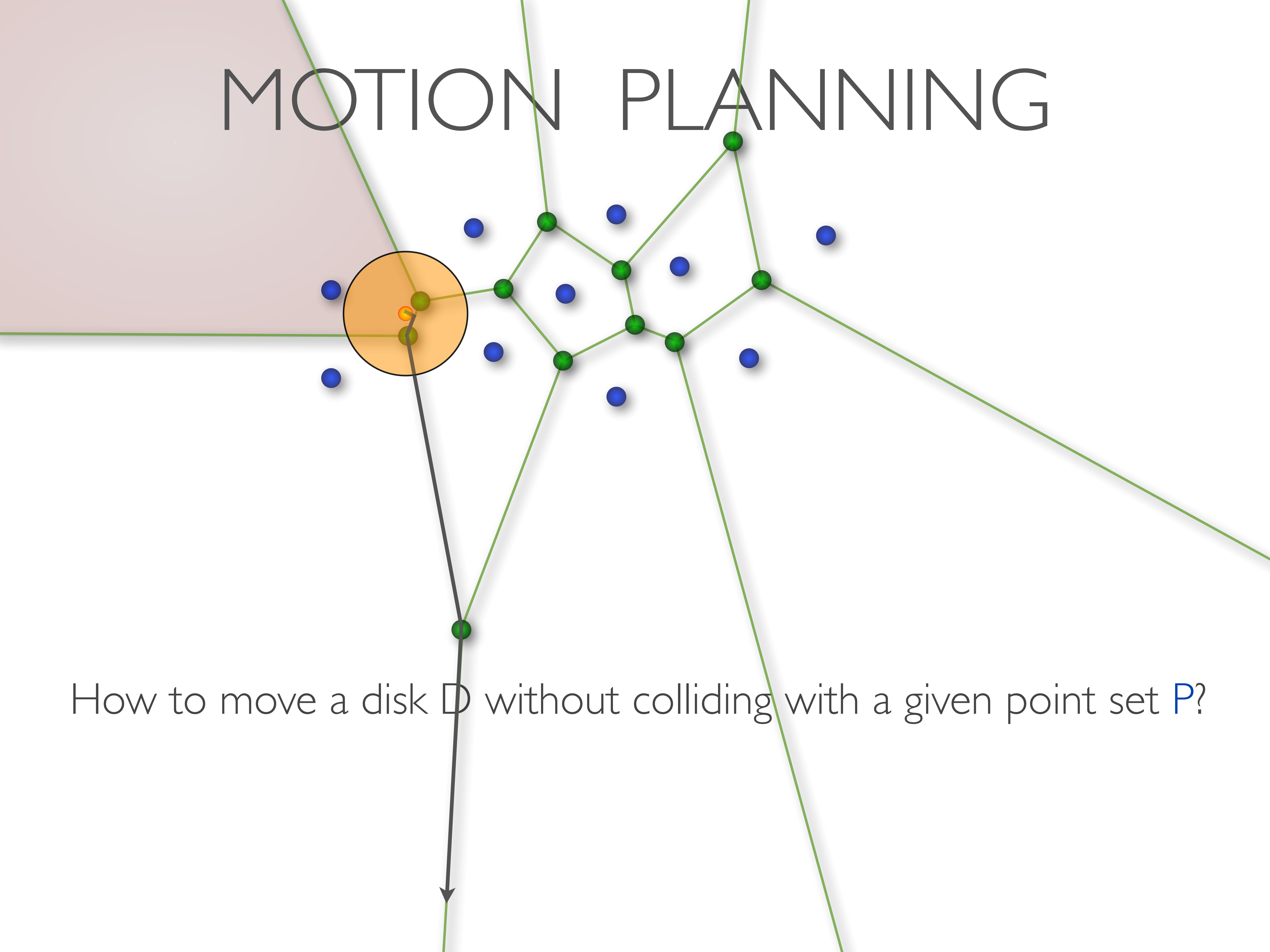
In contrast to iterators, where such a range is empty.

```
Triangulation t;
...
Triangulation::Vertex_handle v = ...;
// find all infinite edges incident to v
Triangulation::Edge_circulator c = t.incident_edges(v);
do {
  if (t.is_infinite(c)) { ... }
  ...
} while (++c != t.incident_edges(v));
```

The usual loop construct to circulate is `do … while`. It ensures at least one iteration and the following increment and therefore works as desired for full circular ranges.

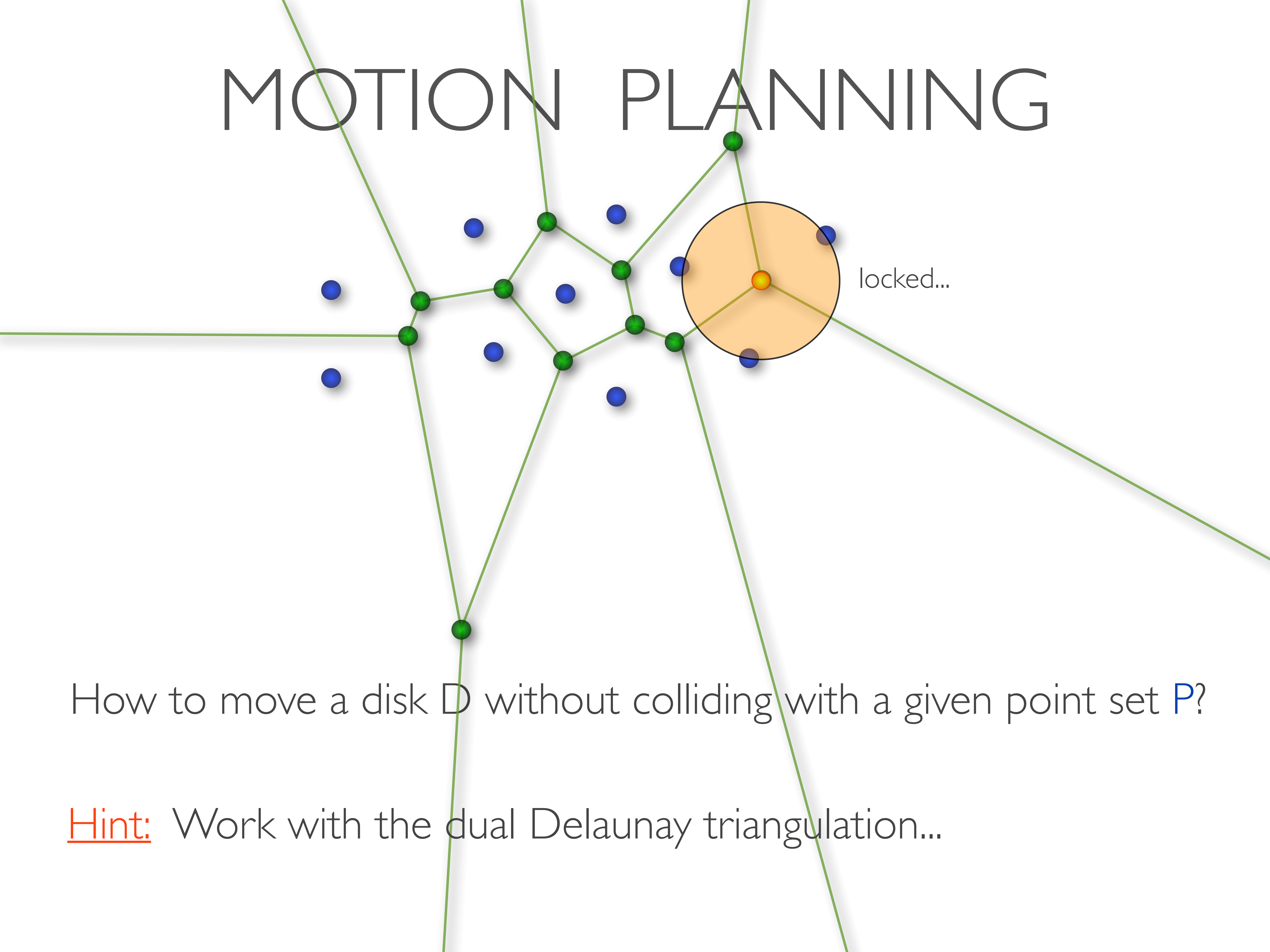
There are no isolated vertices in a triangulation. Otherwise, we would have to test `c != 0` first. (This is the way to describe an empty circular range.)

MOTION PLANNING



How to move a disk D without colliding with a given point set P?

MOTION PLANNING



How to move a disk D without colliding with a given point set P?

Hint: Work with the dual Delaunay triangulation...

ENHANCING FACES I

Add information (e.g., color) to a face using an external map.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <map>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K>  Triangulation;
enum Color { Black = 0, White = 1, Red = 2 };
typedef std::map<Triangulation::Face_handle,Color> Colormap;
```

Can be done in the same way for vertices and edges. (For edges, there are no handles, but the edge type can be used directly.)

```
...
Triangulation t;
Colormap colors;
...
// color all finite faces white
for (Face_iterator f = t.finite_faces_begin(); f != t.finite_faces_end(); ++f)
  colors[f] = White;
...
```

Lookup in a map costs O(log n), where n is the number of entries.

ENHANCING FACES II

Store information in the face directly => O(1) time access.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>

enum Color { Black = 0, White = 1, Red = 2 };
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<Color,K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>  Triangulation;

...
Triangulation t;
...
// color all infinite faces black
Triangulation::Face_circulator f = t.incident_faces(t.infinite_vertex());
do {
  f->info() = Black;
} while (++f != t.incident_faces(t.infinite_vertex()));
...
```

Info parameter. Here: each face gets a `Color`.

New face class, vertex class stays the same.

Change the underlying triangulation data structure (so far we've used the default).

Can be done in the same way for vertices. But for edges this does not work because they are represented implicitly only.