2.7 Geometrische Algorithmen

Nachbarschaften

- Gegeben sei eine Punktmenge $p_1,\ldots,p_n$
- Problemstellungen
  - geringster Abstand zwischen zwei p_i
  - k nächste Punkte zu p_i
  - k nächste Punkte zu (x,y)
  - alle Punkte innerhalb eines Kreises (x,y,r)
  - …

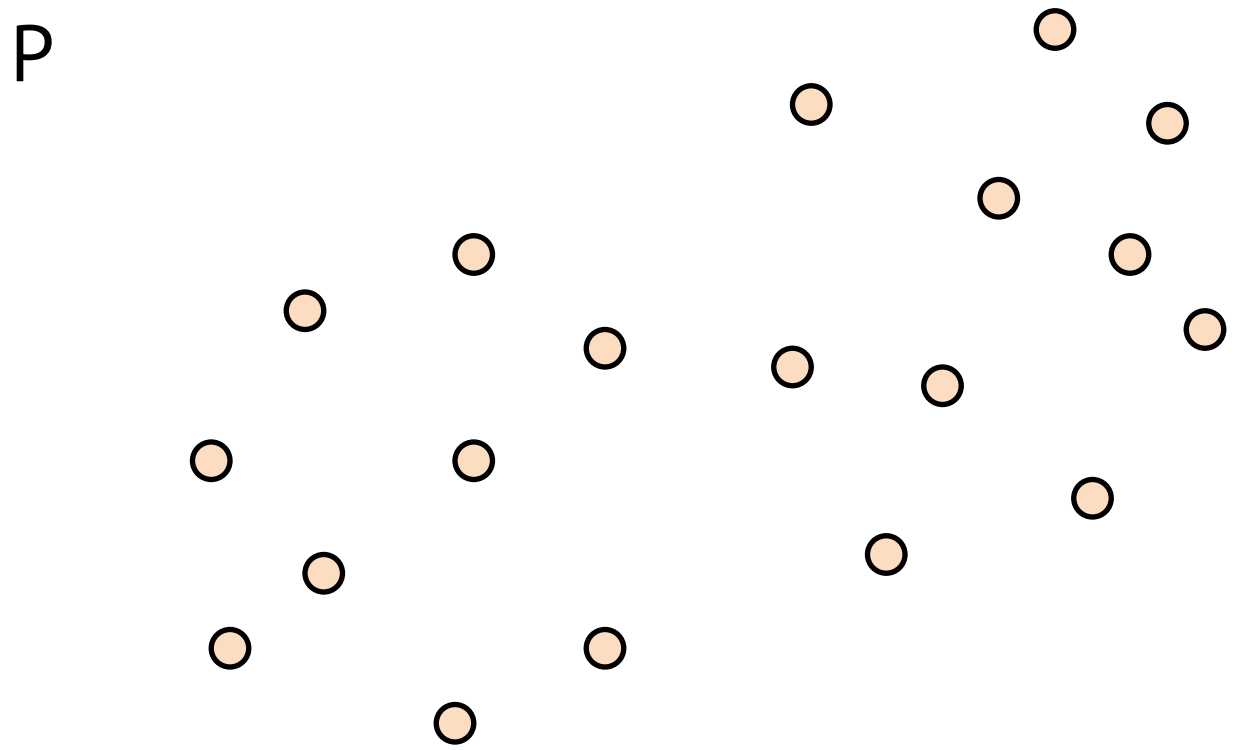
Maximale Punktdichte

- Gegeben: Punktmenge $p_1,\ldots,p_n$
- $d_{ij} = \|p_j - p_i\|$
- finde $\min_{ij} \{d_{ij}\}$
- triviale Lösung in $O(n^2)$
- schnellerer Algorithmus?

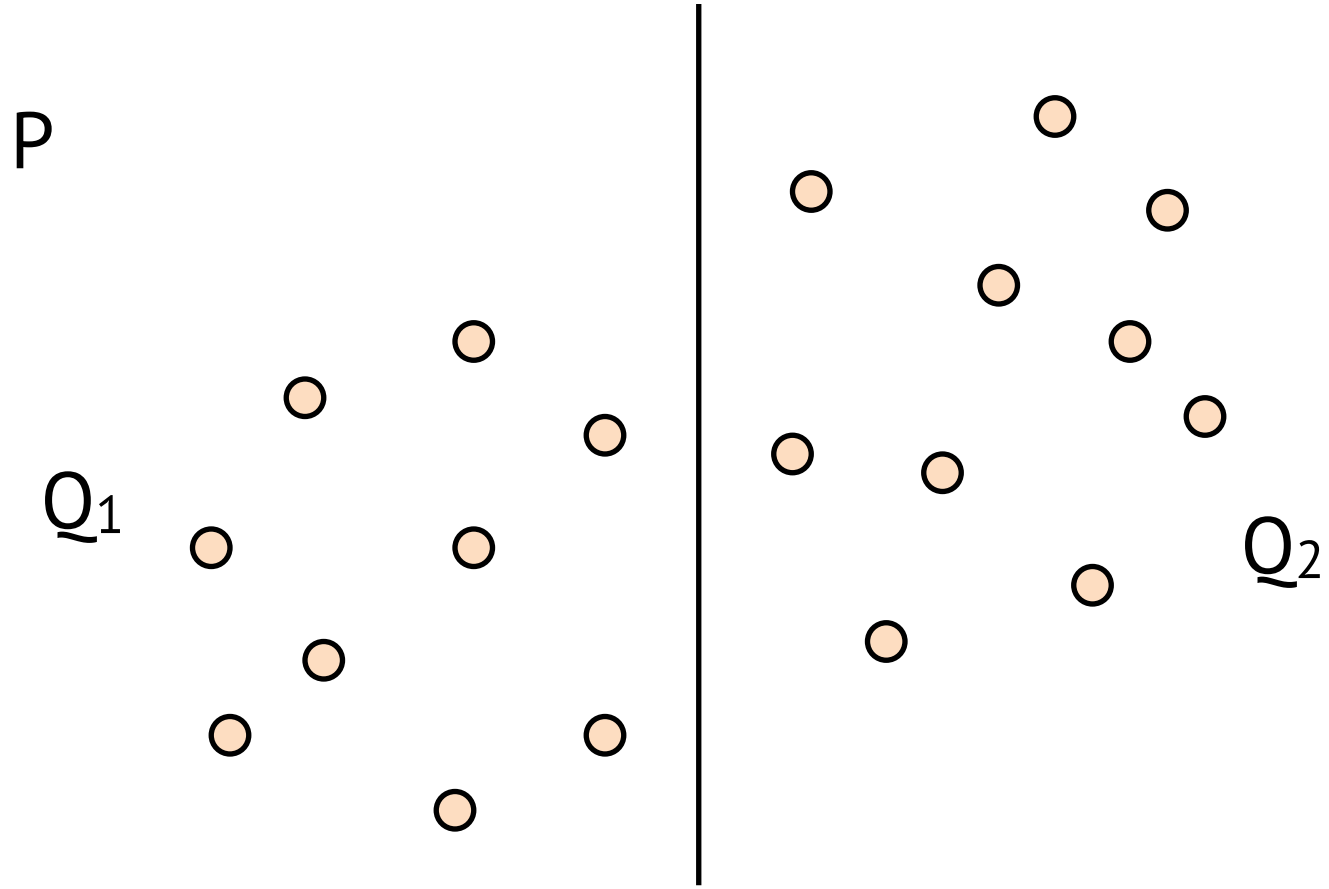
Maximale Punktdichte

- Versuche bestimmte Punktpaare p_i,p_j ohne weitere Berechnung auszuschliessen
- Ansatz: Divide & Conquer
  - Teile die Menge $P=\{p_1,\ldots,p_n\}$ in $Q_1=\{p_1,\ldots,p_{n/2}\}$ und $Q_2=\{p_{n/2+1},\ldots,p_n\}$
  - minimaler Abstand innerhalb Q_1
  - minimaler Abstand innerhalb Q_2
  - minimaler Abstand zwischen Q_1 und Q_2

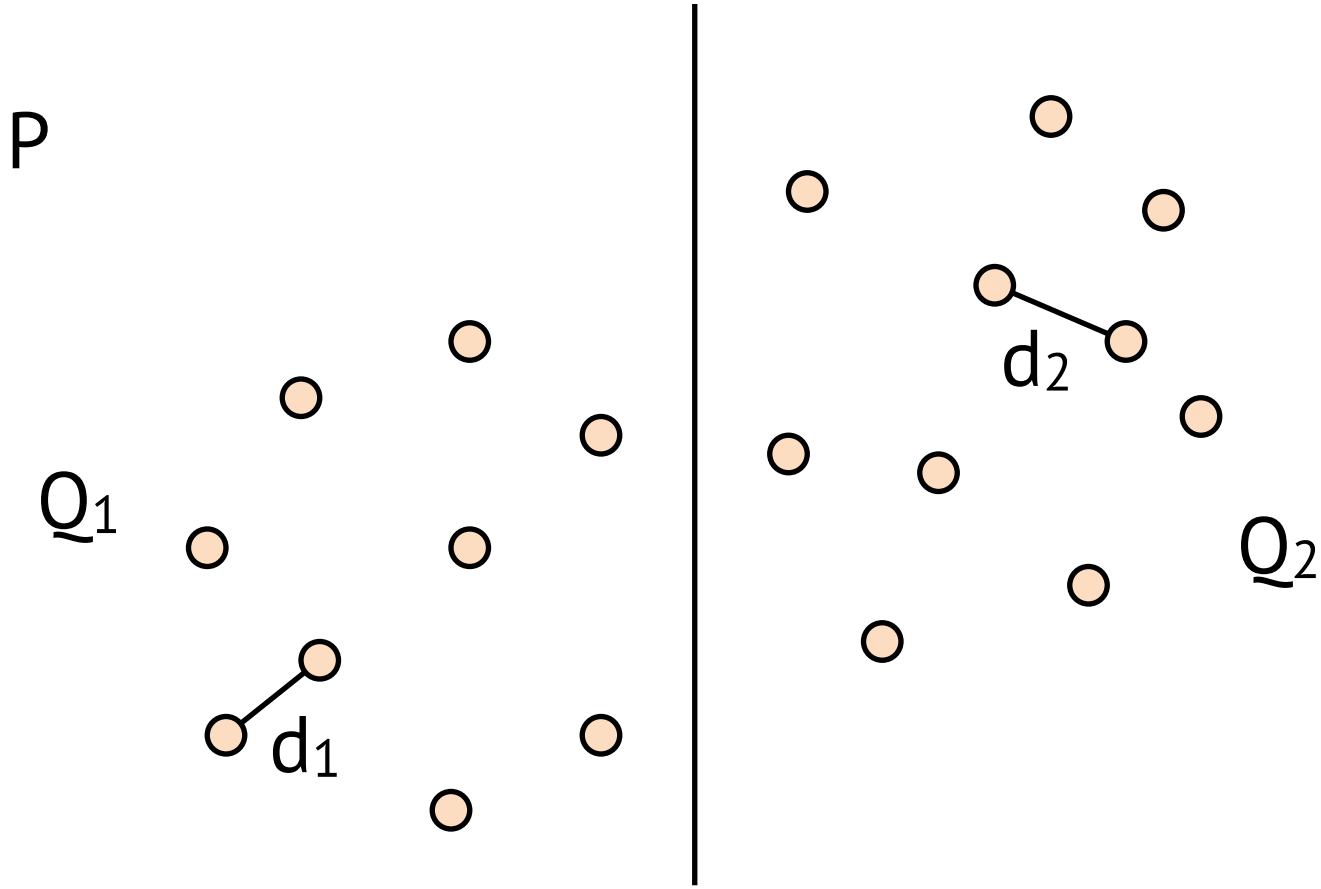
Divide & Conquer



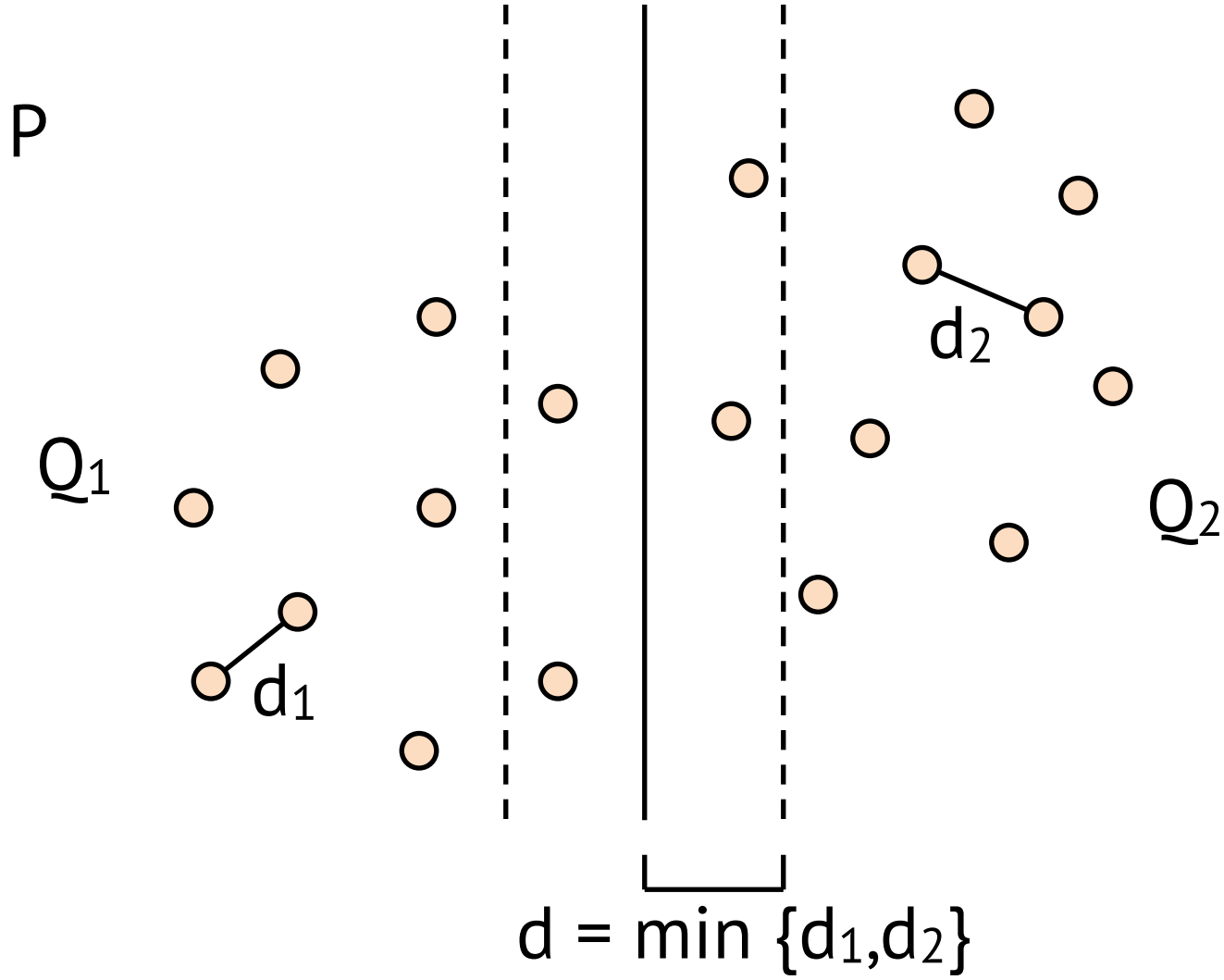
Divide & Conquer



Divide & Conquer



Divide & Conquer



1. Algorithmus

- MinDist(p[1..n])

  sort p[i] by increasing x-coordinate

  d ← MinDistRec(p[1..n])

  return d

1. Algorithmus

- MinDistRec(p[1..n])

  $d_1 \leftarrow$ MinDistRec(p[1..n/2])

  $d_2 \leftarrow$ MinDistRec(p[n/2+1..n])

  $d \leftarrow \min(d_1, d_2)$

  $x \leftarrow (p[n/2].x + p[n/2+1].x) / 2$

  $Q_1 \leftarrow$ select $p[i \leqslant n/2]$ where $|p[i].x - x| \leqslant d$

  $Q_2 \leftarrow$ select $p[i > n/2]$ where $|p[i].x - x| \leqslant d$

  $d_3 \leftarrow$ MinDistCross(Q_1,Q_2)

  return $\min(d,d_3)$

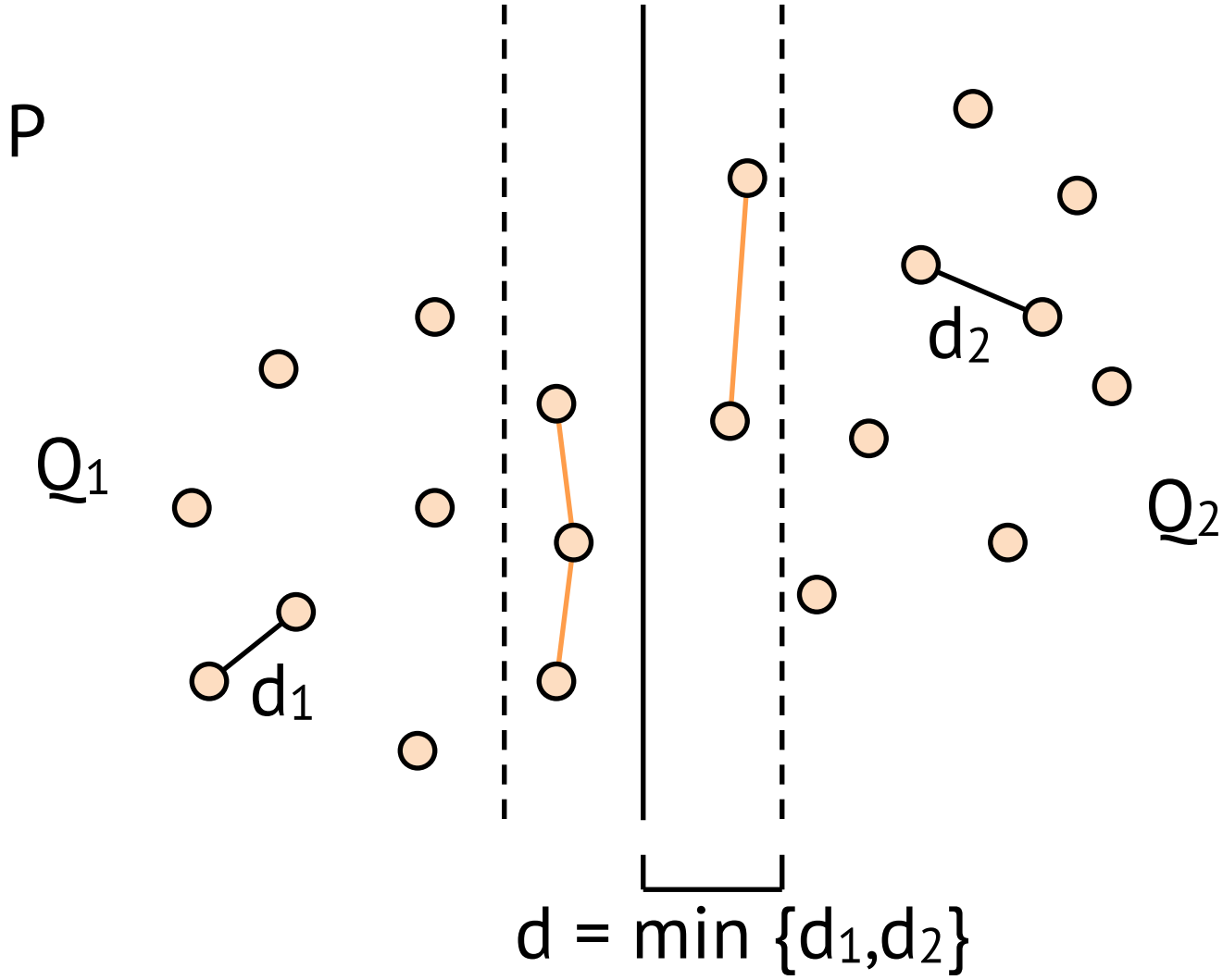
1. Algorithmus

- $T(n) = 2\,T(n/2) + O(n^2)$
- $T(n) = O(n^2)$
- … nicht besser als die triviale Lösung

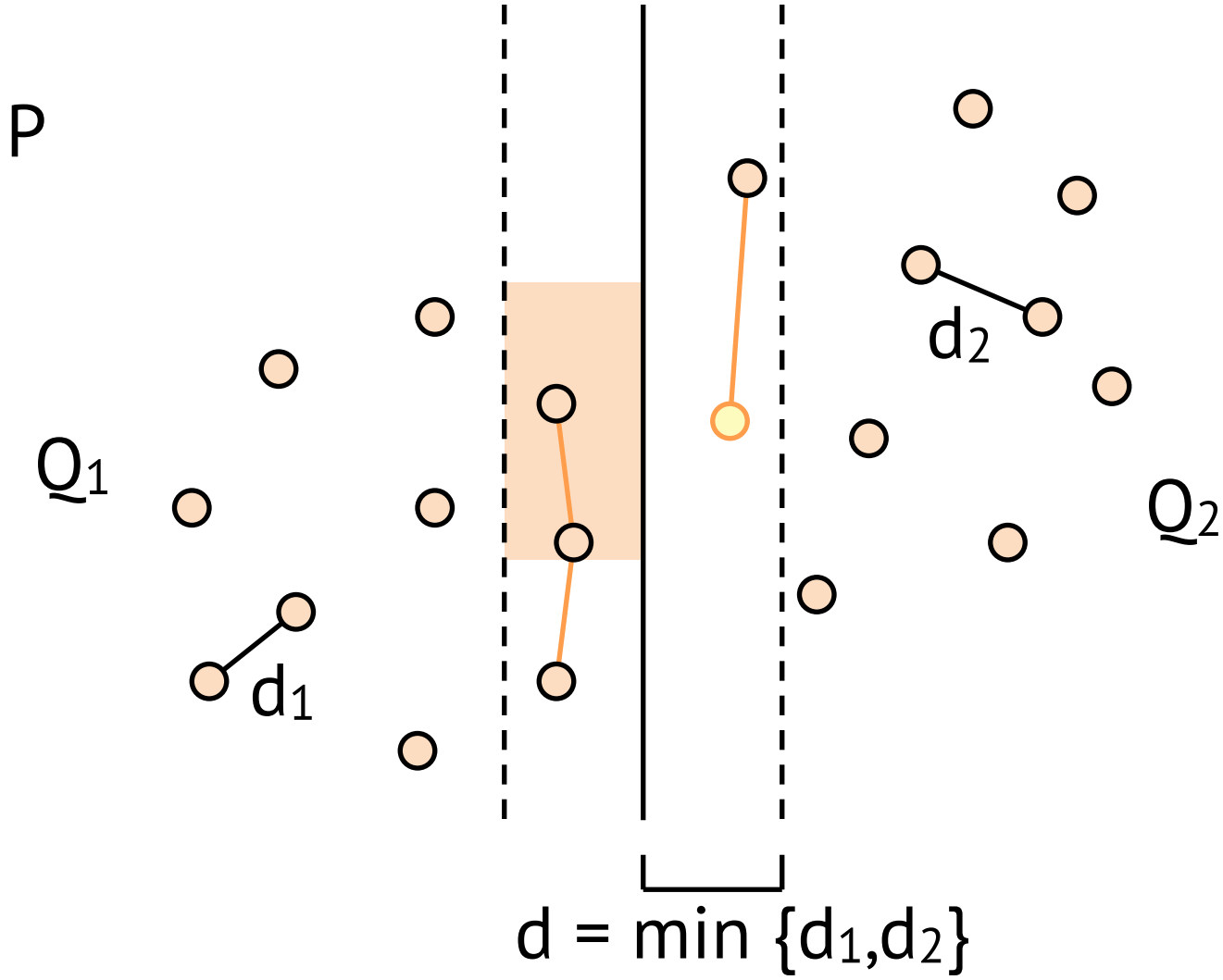
2. Algorithmus

- Die Kandidaten für minimale Distanzen in der Menge Q können weiter eingeschränkt werden …

Divide & Conquer



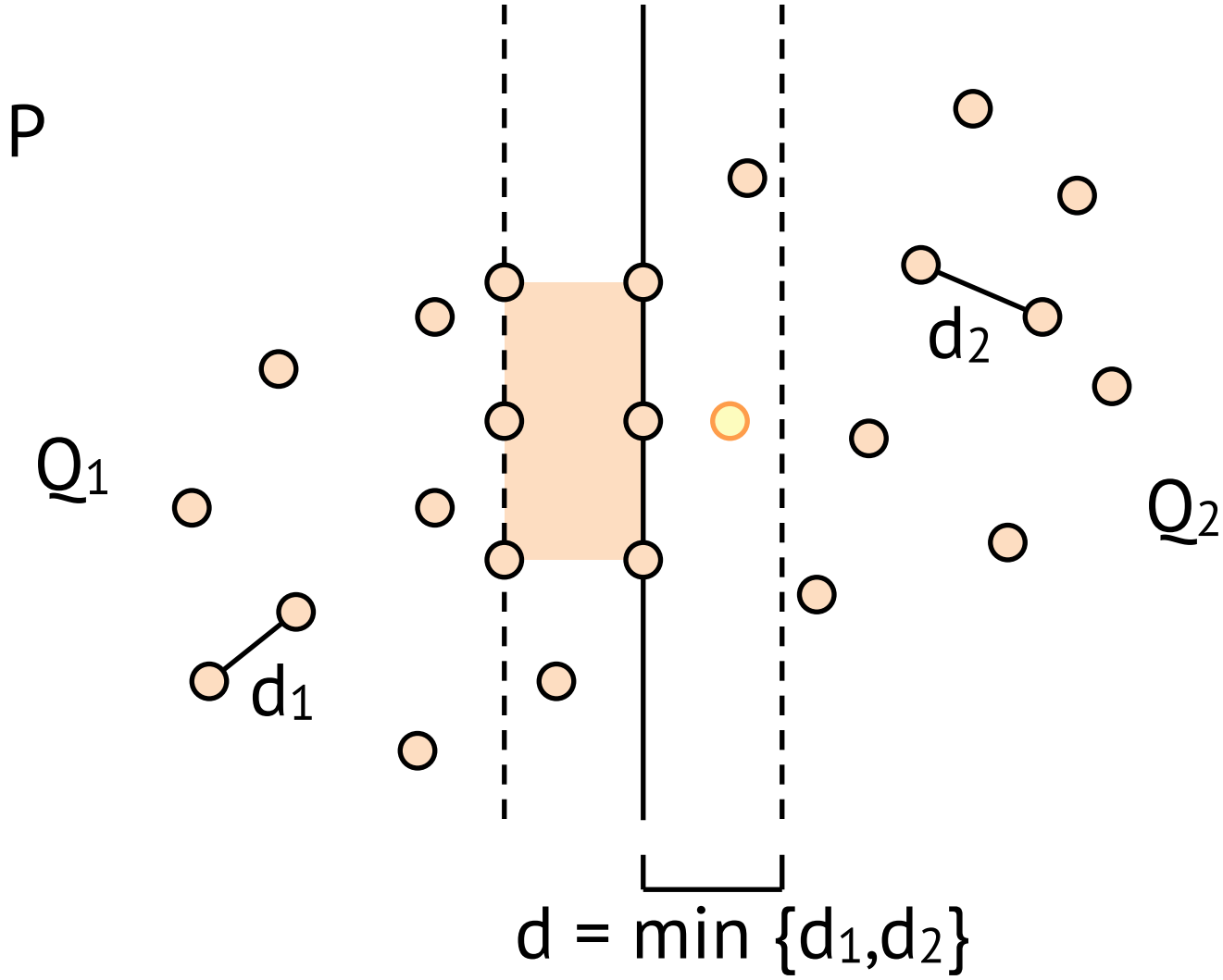
Divide & Conquer



2. Algorithmus

- Die Kandidaten für minimale Distanzen in der Menge Q können weiter eingeschränkt werden …
- … jeder Punkt muss mit maximal sechs Kandidaten verglichen werden

Divide & Conquer



2. Algorithmus

- MinDistRec(p[1..n])

  $d_1 \leftarrow$ MinDistRec(p[1..n/2])

  $d_2 \leftarrow$ MinDistRec(p[n/2+1..n])

  $d \leftarrow \min(d_1, d_2)$

  $x \leftarrow (p[n/2].x + p[n/2+1].x) / 2$

  $Q_1 \leftarrow$ select $p[i \leqslant n/2]$ where $|p[i].x - x| \leqslant d$

  $Q_2 \leftarrow$ select $p[i > n/2]$ where $|p[i].x - x| \leqslant d$

  sort Q_1, Q_2 by increasing y-coordinate

  $d_3 \leftarrow$ MinDistRestrict(Q_1,Q_2,d)

  return $\min(d,d_3)$

2. Algorithmus

- MinDistRestrict(q[1..k], q'[1..k'], d)

```
dist ← d
l ← 1, r ← 1
for i ← 1 to k do
    while q'[l].y < q[i].y−d and l < k' do
        l ← l + 1
    while q'[r].y < q[i].y+d and r ⩽ k' do
        r ← r + 1
    for j ← l to r − 1 do
        dist ← min(dist, dist(q[i],q'[j]) )
return dist
```

2. Algorithmus

- MinDistRestrict(q[1..k], q'[1..k'], d)

```
dist ← d
l ← 1, r ← 1
for i ← 1 to k do                                   O(n)
    while q'[l].y < q[i].y−d and l < k' do          O(1)
        l ← l + 1
    while q'[r].y < q[i].y+d and r ⩽ k' do
        r ← r + 1
    for j ← l to r − 1 do                           O(1)
        dist ← min(dist, dist(q[i],q'[j]) )
return dist
```

2. Algorithmus

- MinDistRec(p[1..n])

  $d_1 \leftarrow$ MinDistRec(p[1..n/2])

  $d_2 \leftarrow$ MinDistRec(p[n/2+1..n])

  $d \leftarrow \min(d_1, d_2)$

  $x \leftarrow (p[n/2].x + p[n/2+1].x) / 2$

  O(n) $Q_1 \leftarrow$ select p[i ⩽ n/2] where |p[i].x - x| ⩽ d

  $Q_2 \leftarrow$ select p[i > n/2] where |p[i].x - x| ⩽ d

  O(nlogn) sort Q_1, Q_2 by increasing y-coordinate

  O(n) $d_3 \leftarrow$ MinDistRestrict(Q_1,Q_2,d)

  return min(d,d_3)

2. Algorithmus

- $T(n) = 2 \times T(n/2) + O(n) + O(n \times \log n)$
- $T(n) = 2 \times T(n/2) + O(n \times \log n)$
- $T(n) = O(n \times \log^2 n)$

3. Algorithmus

- Beobachtung: der MinDist-Algorithmus hat dieselbe Struktur wie Merge-Sort
  - Splitting in zwei gleichgroße Teile
  - Sortierung beim Zusammenfügen
- Merging von sortierten Folgen kostet nur O(n)
- MinDistRec() gibt nach der y-Koordinate sortierte Folgen zurück …

3. Algorithmus

- MinDistRec(p[1..n])

  $x \leftarrow (p[n/2].x + p[n/2+1].x) / 2$

  $d_1 \leftarrow$ MinDistRec(p[1..n/2])

  $d_2 \leftarrow$ MinDistRec(p[n/2+1..n])

  $d \leftarrow \min(d_1, d_2)$

  $Q_1 \leftarrow$ select $p[i \leqslant n/2]$ where $|p[i].x - x| \leqslant d$

  $Q_2 \leftarrow$ select $p[i > n/2]$ where $|p[i].x - x| \leqslant d$

  merge pre-sorted lists Q_1, Q_2

  $d_3 \leftarrow$ MinDistRestrict(Q_1,Q_2,d)

  merge pre-sorted lists $p[i \leqslant n/2], p[i > n/2]$

  return $\min(d,d_3)$

3. Algorithmus

- MinDistRec(p[1..n])

  $x \leftarrow (p[n/2].x + p[n/2+1].x) / 2$

  $d_1 \leftarrow$ MinDistRec(p[1..n/2])

  $d_2 \leftarrow$ MinDistRec(p[n/2+1..n])

  $d \leftarrow \min(d_1, d_2)$

  O(n) $Q_1 \leftarrow$ select $p[i \leqslant n/2]$ where $|p[i].x - x| \leqslant d$

  O(n) $Q_2 \leftarrow$ select $p[i > n/2]$ where $|p[i].x - x| \leqslant d$

  O(n) merge pre-sorted lists Q_1, Q_2

  O(n) $d_3 \leftarrow$ MinDistRestrict(Q_1,Q_2,d)

  O(n) merge pre-sorted lists $p[i \leqslant n/2], p[i > n/2]$

  return $\min(d,d_3)$

3. Algorithmus

- $T(n) = 2 \times T(n/2) + O(n)$
- $T(n) = O(n \times \log n)$

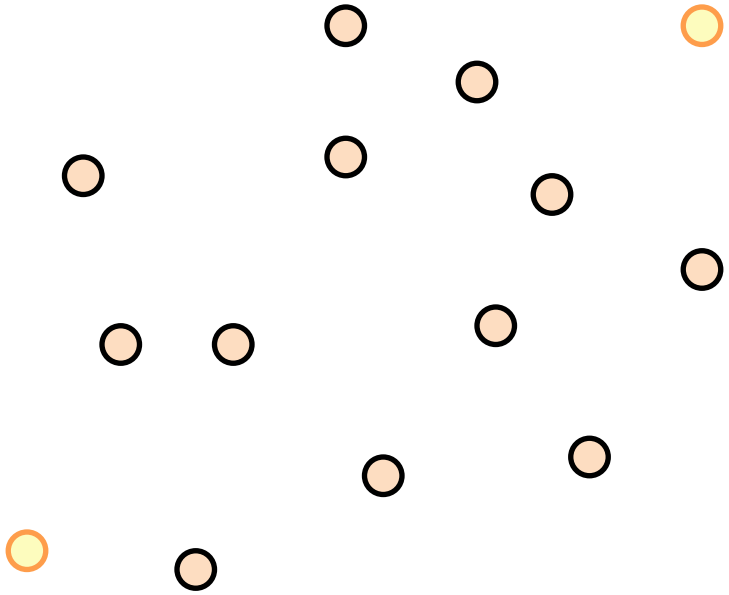
Weitester Abstand

- Gegeben: Punktmenge $p_1,\ldots,p_n$
- $d_{ij} = ||p_j - p_i||$
- finde $\max_{ij} \{d_{ij}\}$
- Durchmesser der Punktmenge
- triviale Lösung in $O(n^2)$
- schnellerer Algorithmus?

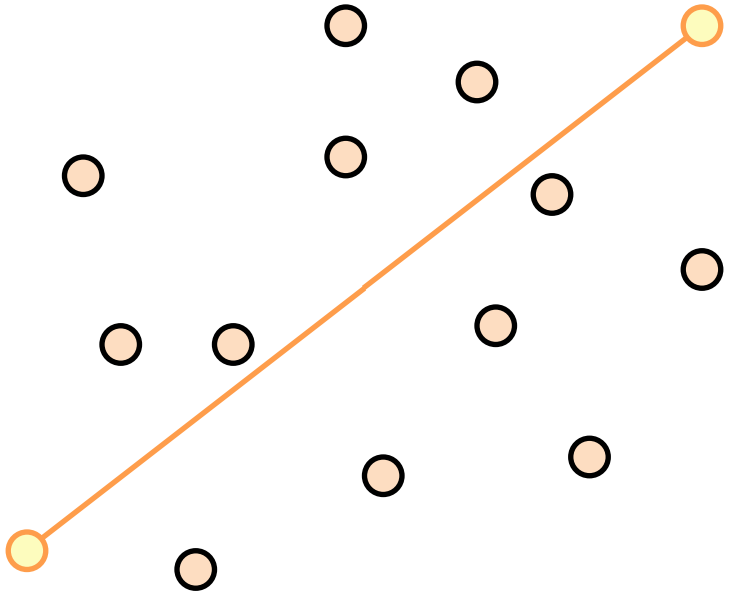
Weitester Abstand

- Versuche wieder bestimmte Punktpaare auszuschließen.
- Der maximale Abstand muss zwischen Extremalpunkten auftreten.
- Beschränke die Suche auf die konvexe Hülle.

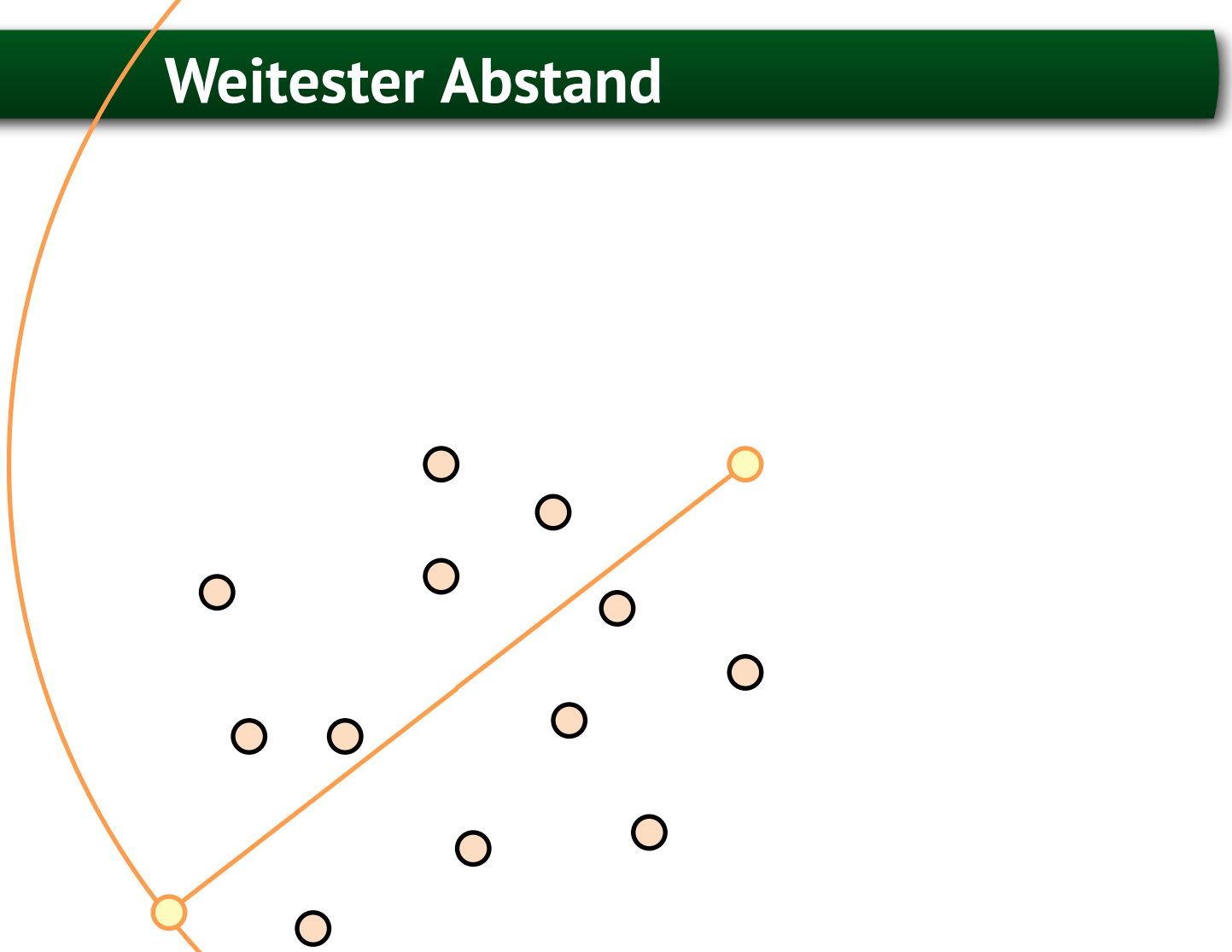
Weitester Abstand



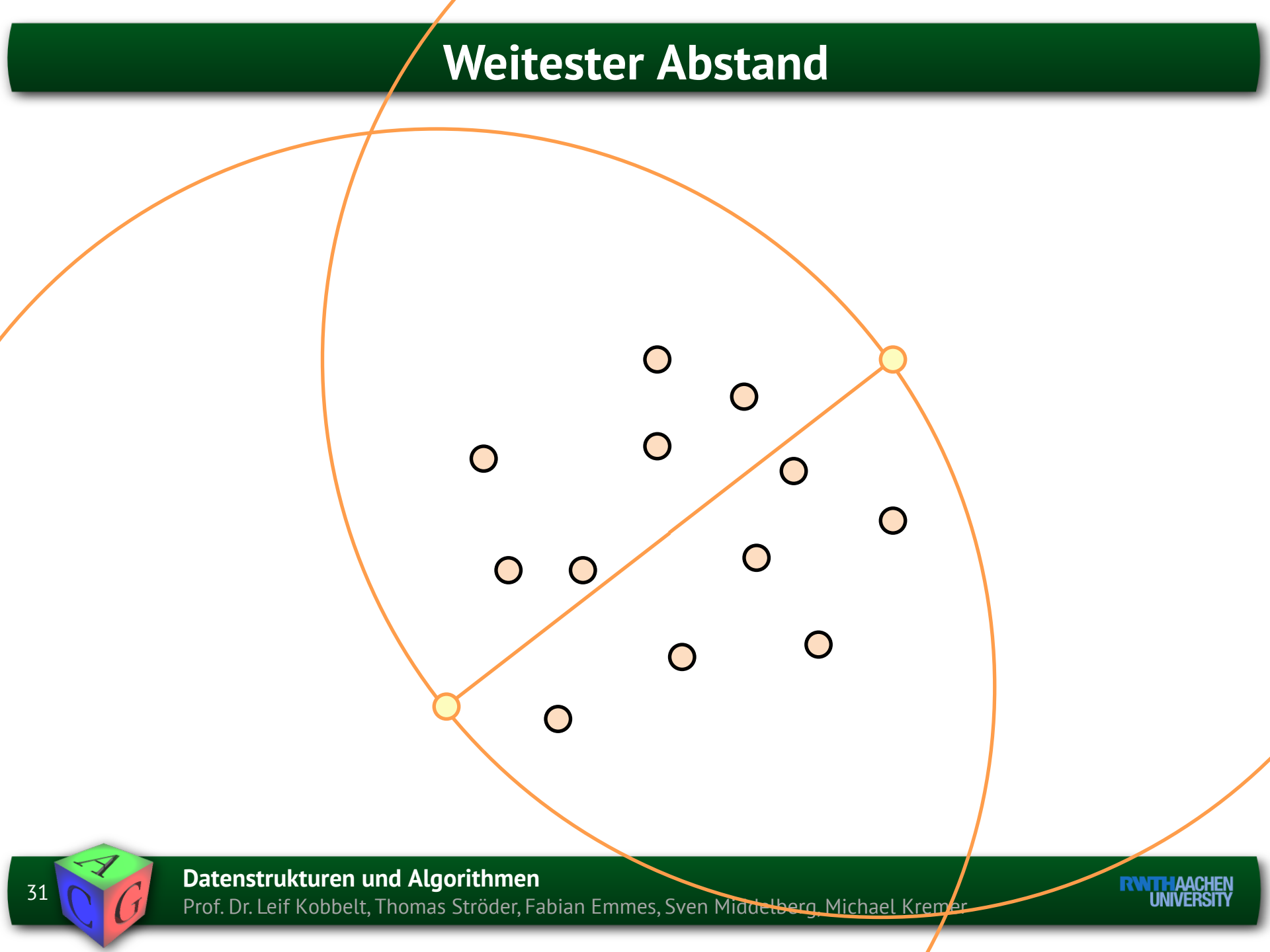
Weitester Abstand



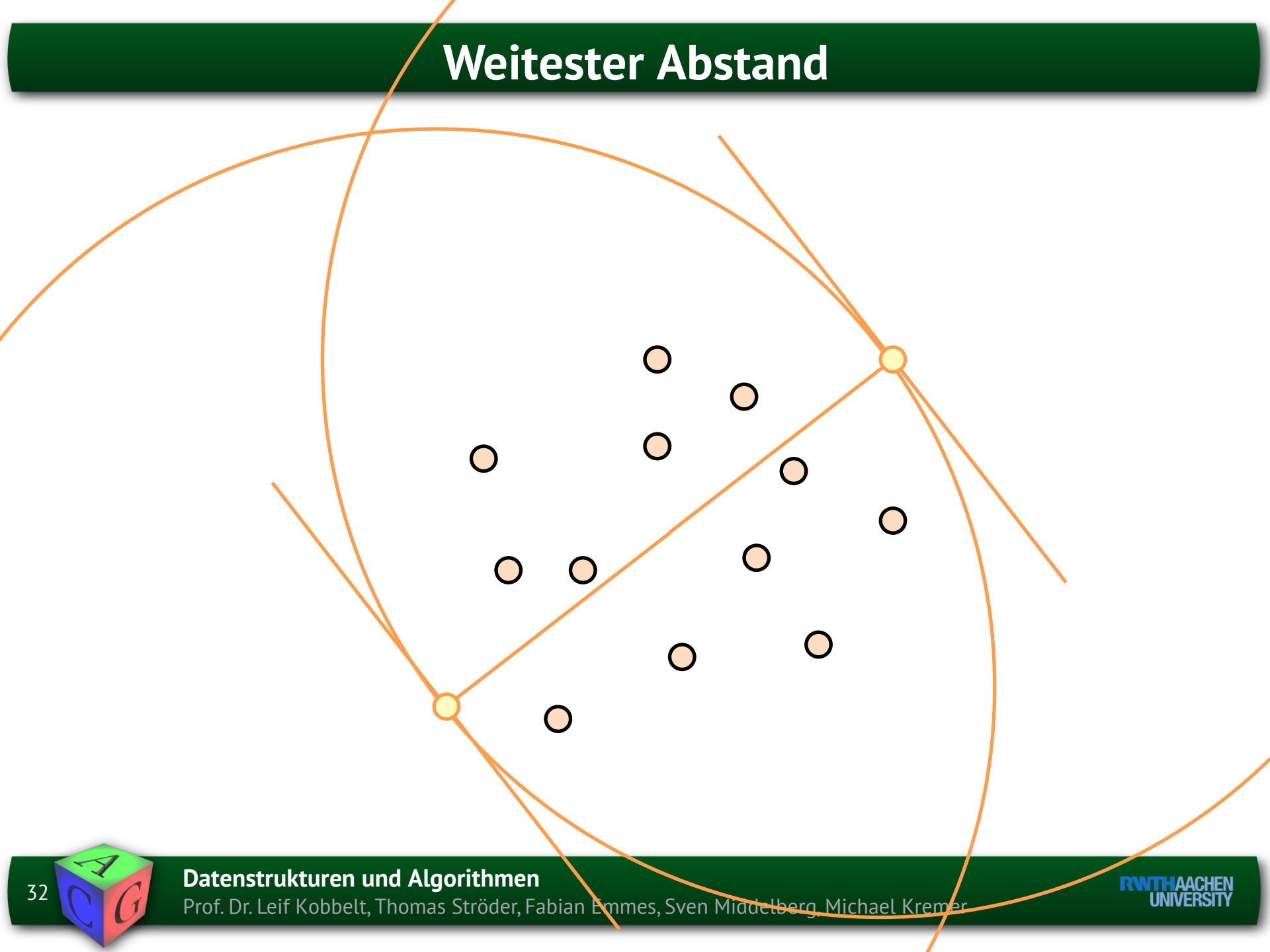
Weitester Abstand



Weitester Abstand



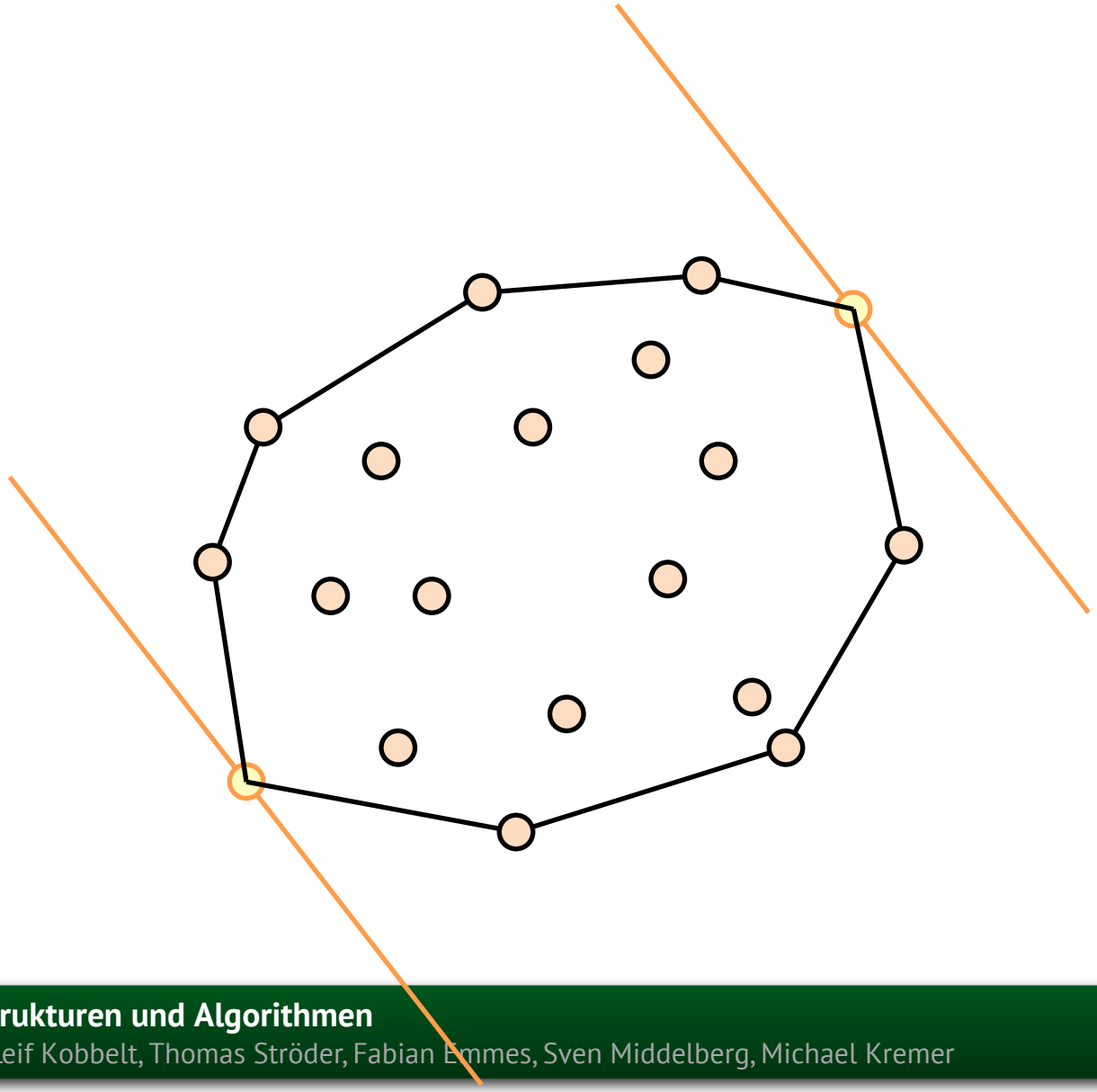
Weitester Abstand



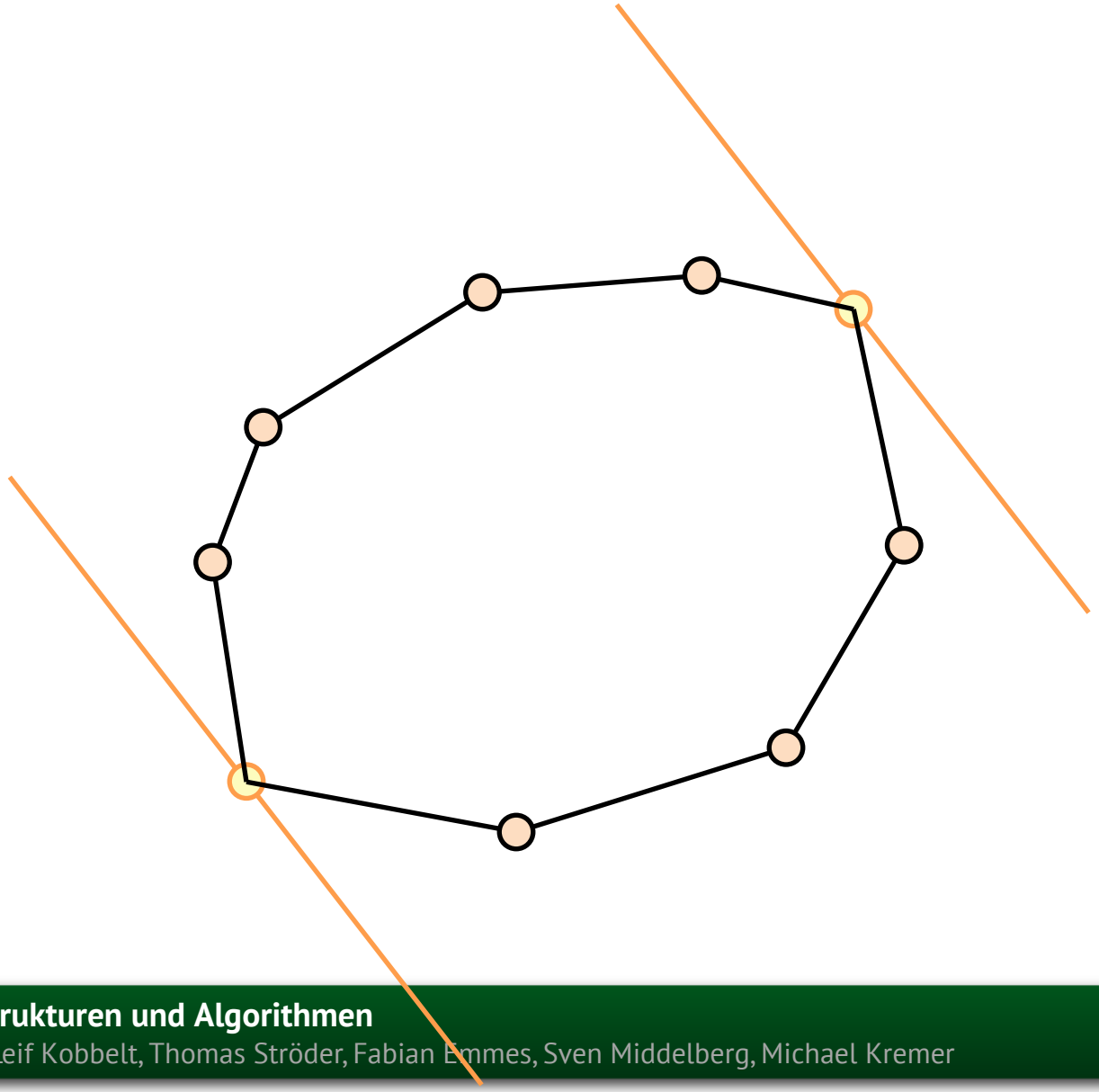
Weitester Abstand

- Geg. Punktmenge $p_1,\ldots,p_n$
  1. Berechne die konvexe Hülle
  2. Suche Extremalpunkte mit parallelen Supporting Lines
  3. Bestimme den maximalen Abstand zwischen diesen Kandidaten

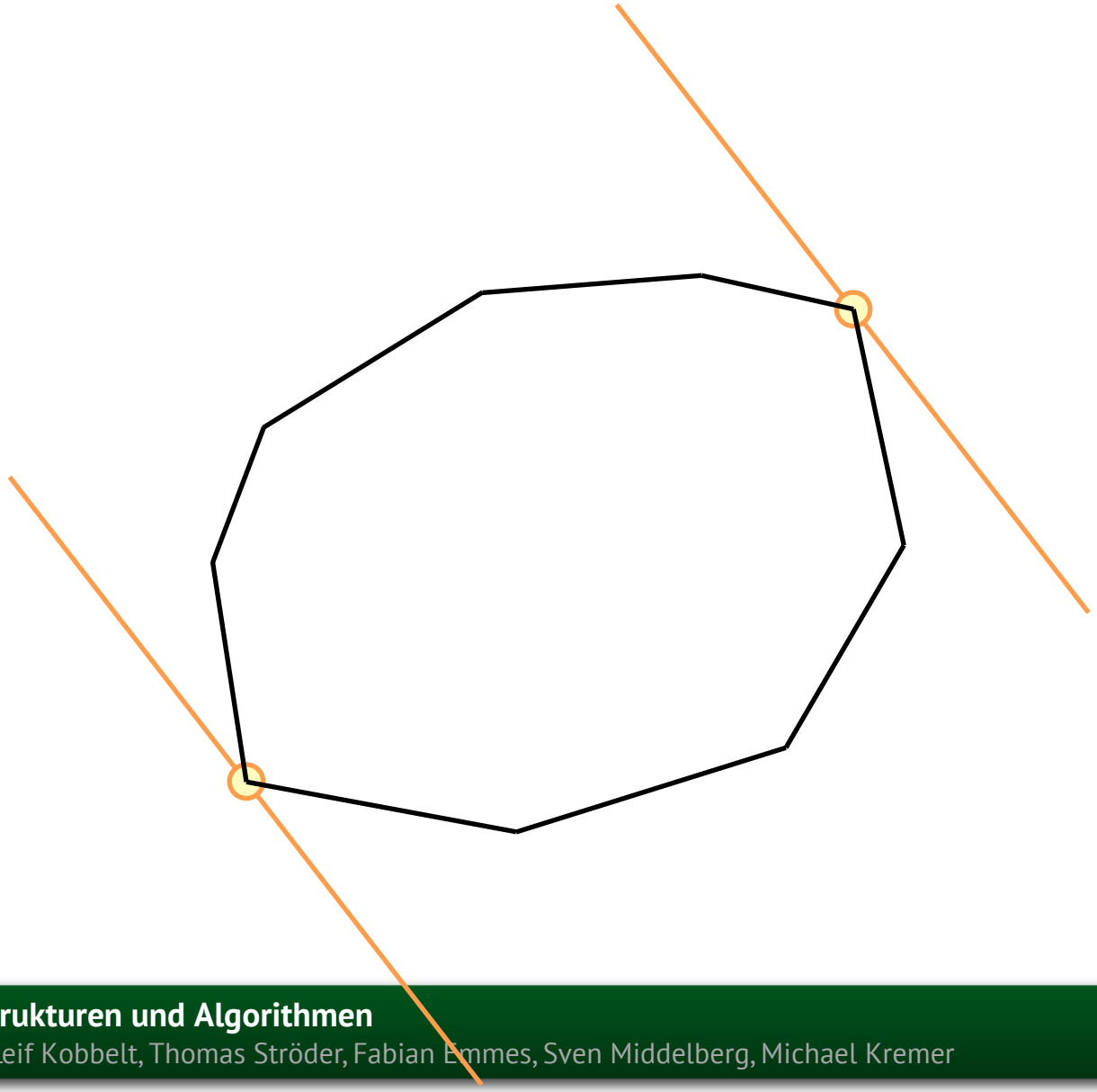
Weitester Abstand



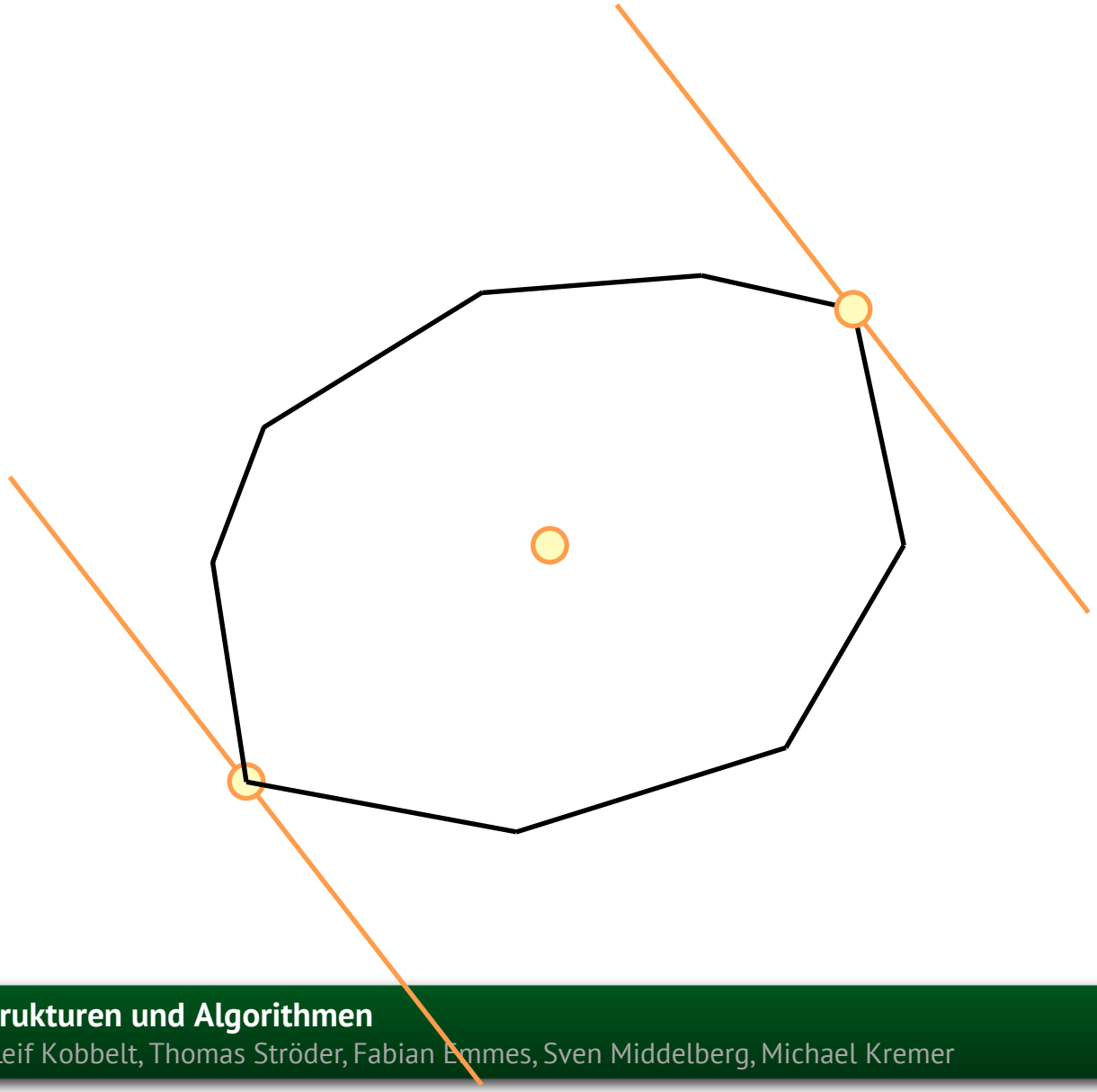
Weitester Abstand



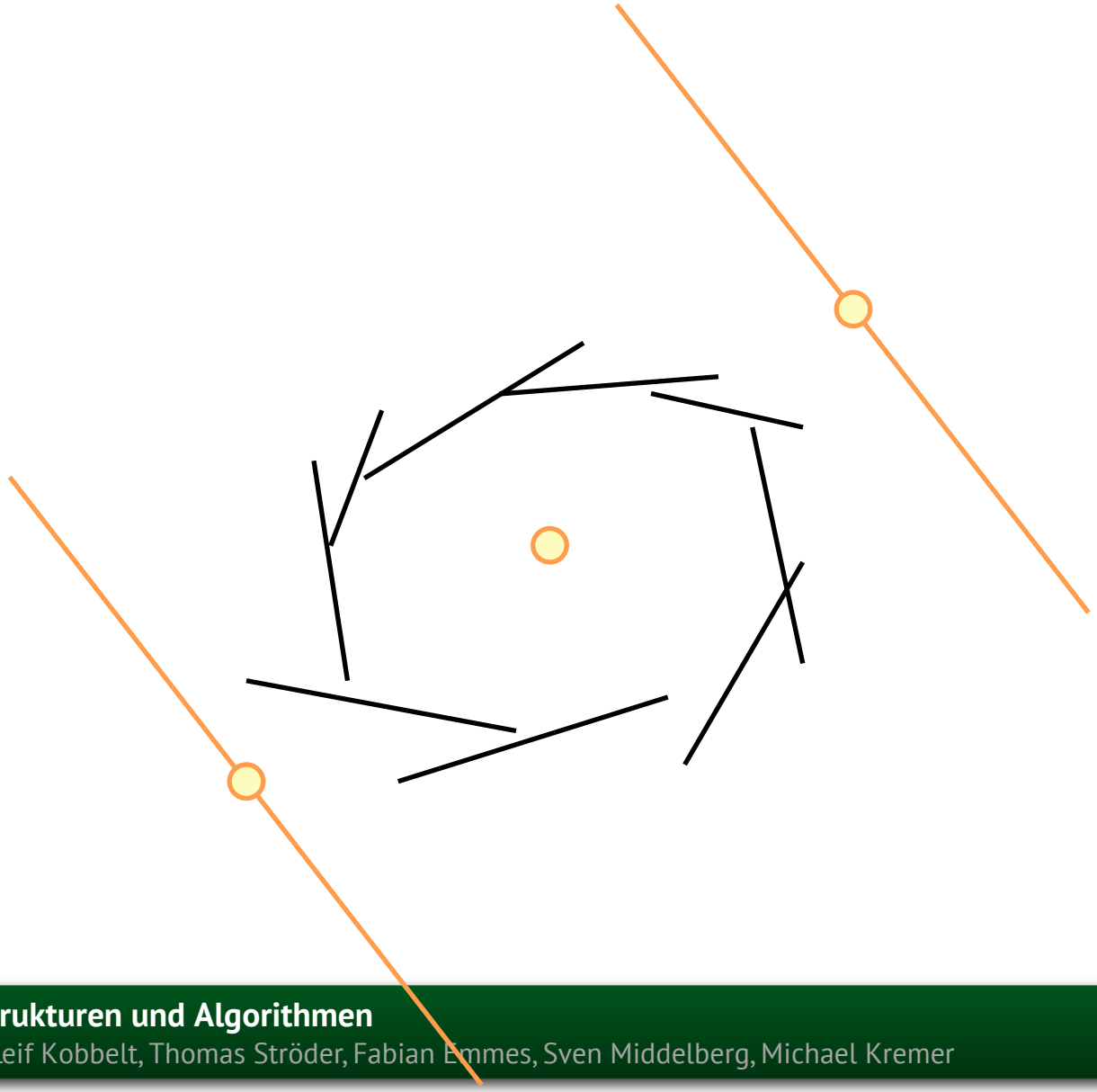
Weitester Abstand



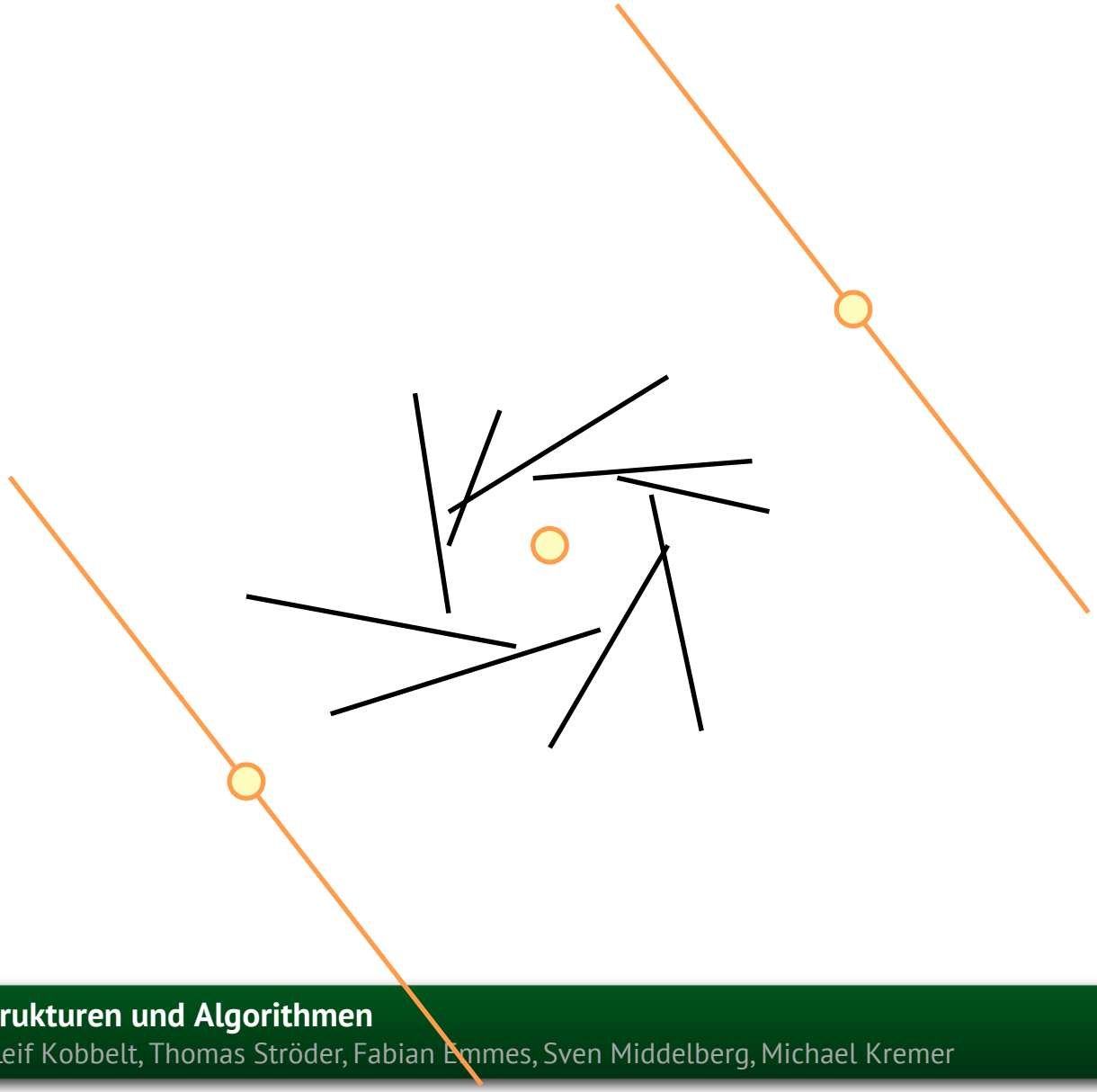
Weitester Abstand



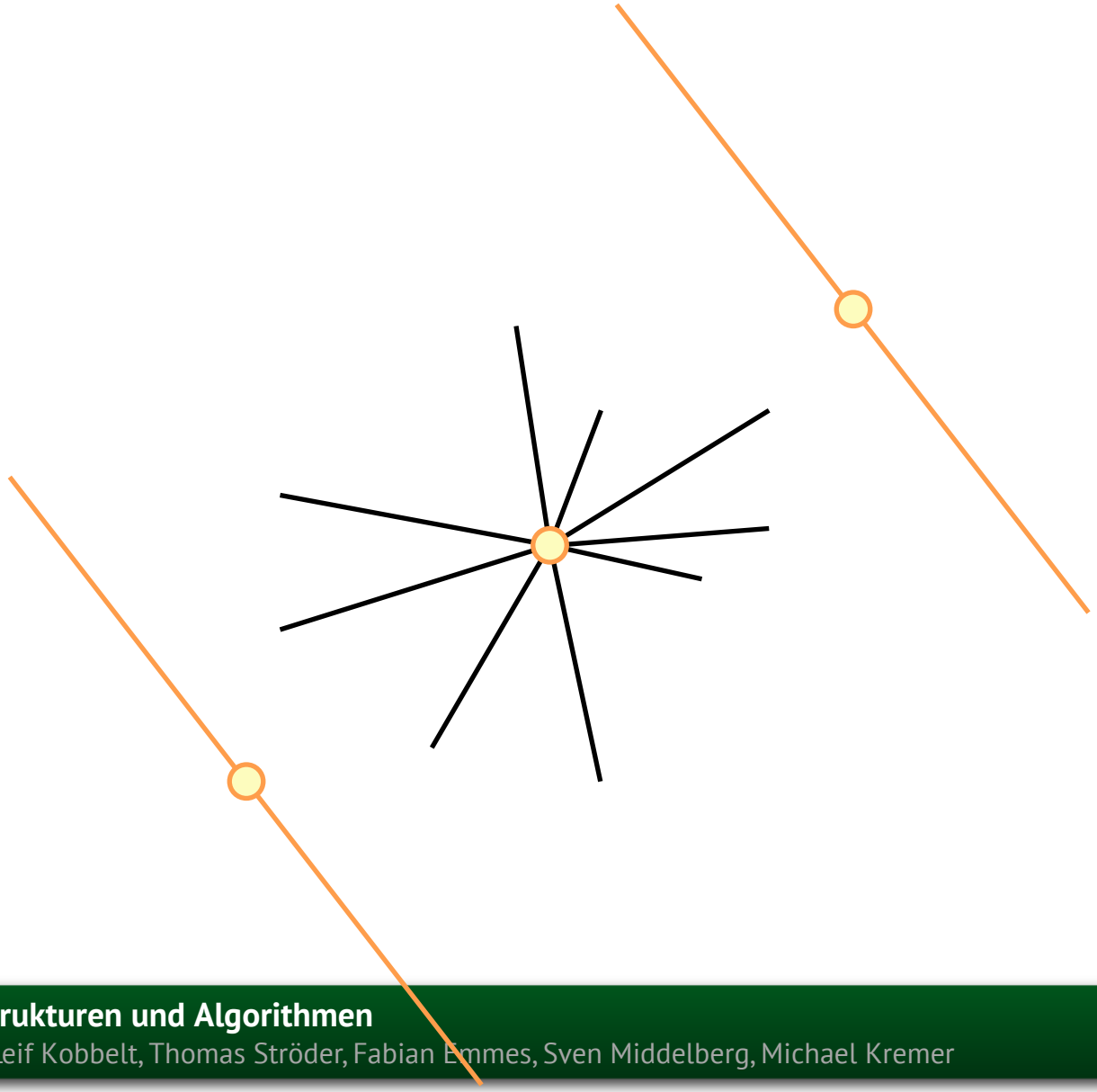
Weitester Abstand



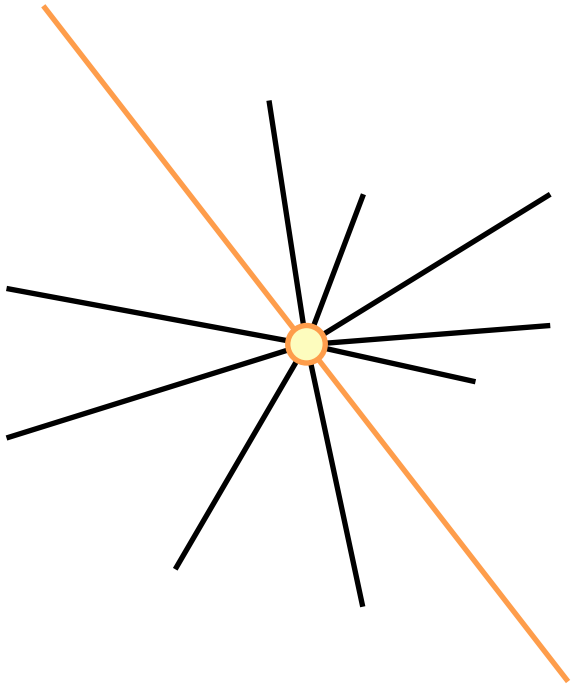
Weitester Abstand



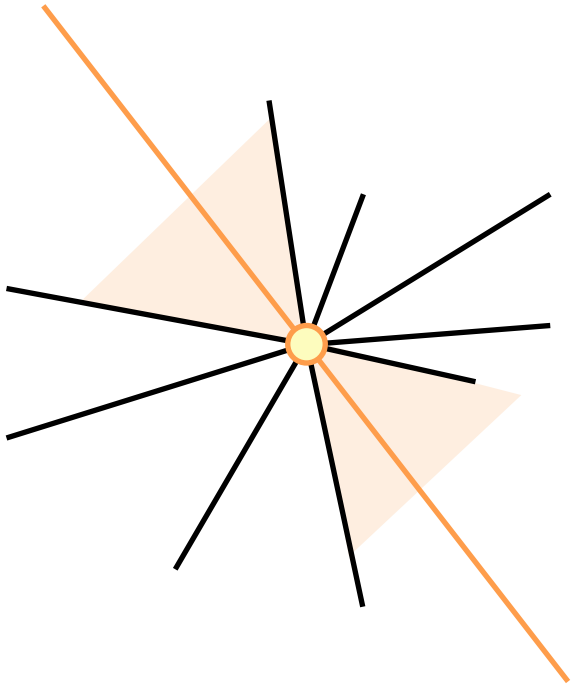
Weitester Abstand



Weitester Abstand



Weitester Abstand



Weitester Abstand

1. Berechne die konvexe Hülle] O(nlogn)
2. Suche Extremalpunkte mit parallelen Supporting Lines
   a. die Kanten der konvexen Hülle bilden Sektoren
   b. gegenüberliegende Sektoren enthalten gemeinsame Gerade] O(n)
3. Finde den maximalen Abstand zwischen diesen Kandidaten] O(n)

Nachbarschaften

- Geg. Punktmenge $p_1, \ldots, p_n$
- Typische Problemstellungen :
  - min. / max. Abstand zwischen zwei p_i
  - k nächste Punkte zu p_i
  - k nächste Punkte zu (x,y)
  - alle Punkte innerhalb eines Kreises (x,y,r)
  - ...

Nächste Punkte

- Beispiele:
  - Datenbanksuche mit Unsicherheit
  - Finde nächste Mobilfunkstation
  - Globale Beleuchtungssimulation
  - …

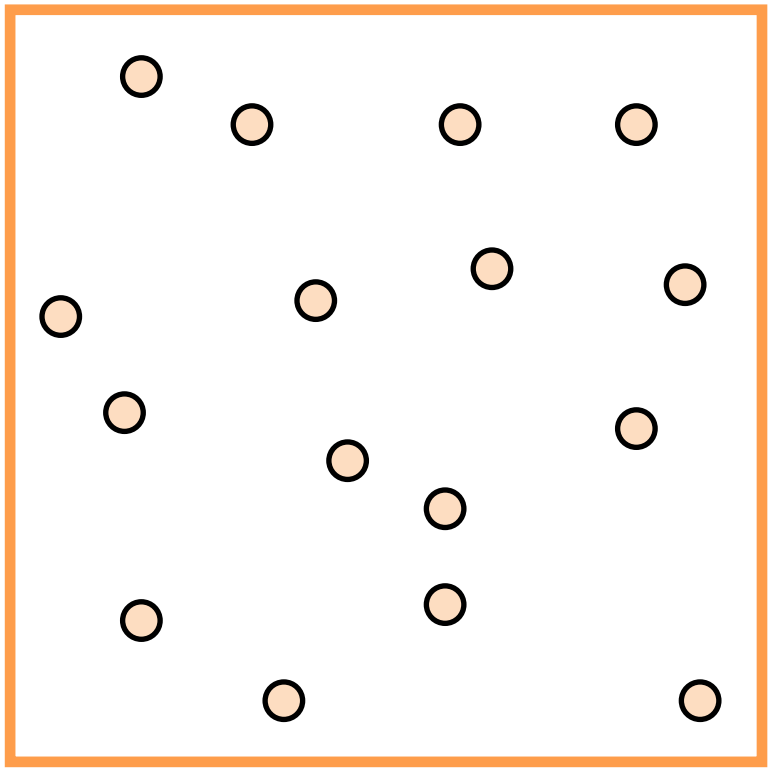
Nächste Punkte

- Geg. Punktmenge $p_1, \ldots, p_n$
- Anfrage: (x,y) [hier: $\mathbb{R}^2$, allg.: $\mathbb{R}^k$]
- Optimale Datenstruktur zum Suchen

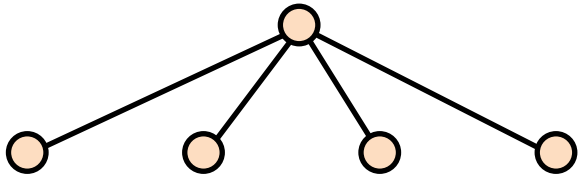
  ⇒ Bäume
- Mehrdimensionale Schlüssel

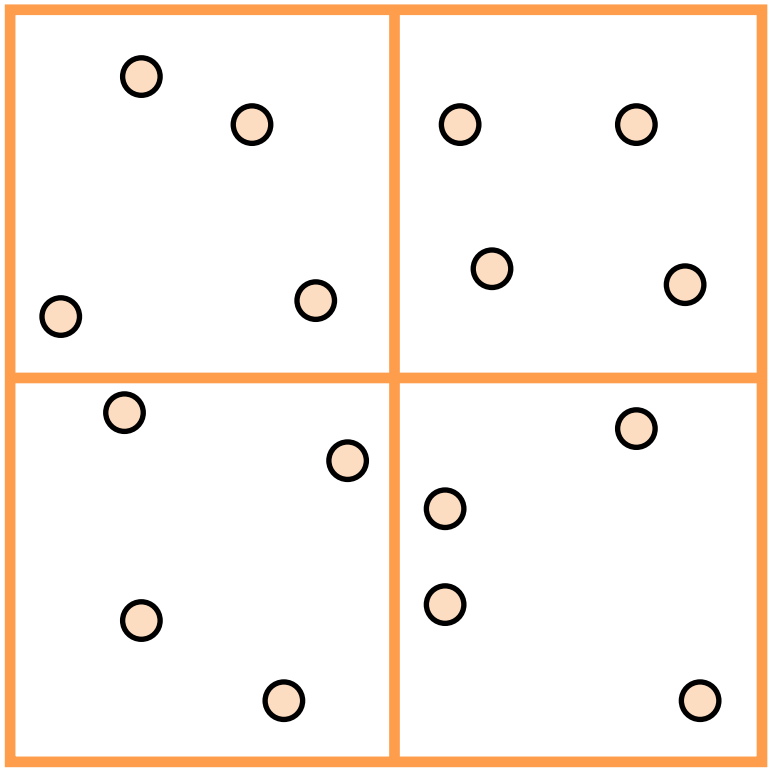
  ⇒ z.B. kD-Bäume

Quadtree

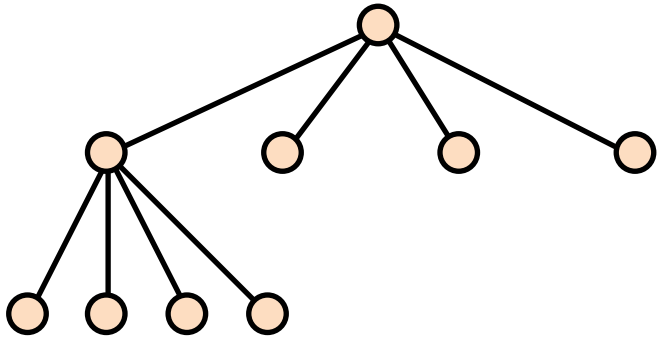


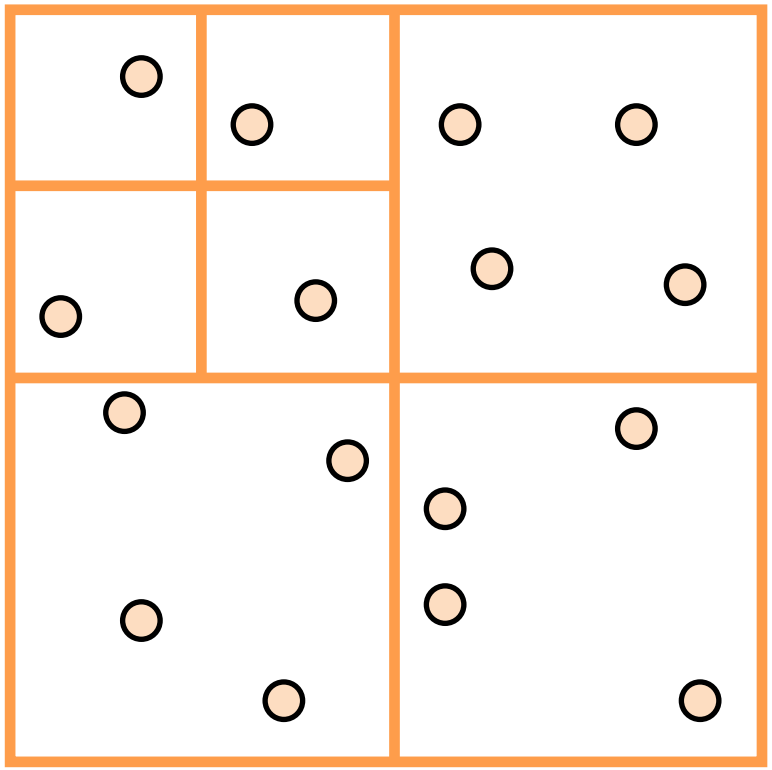
Quadtree



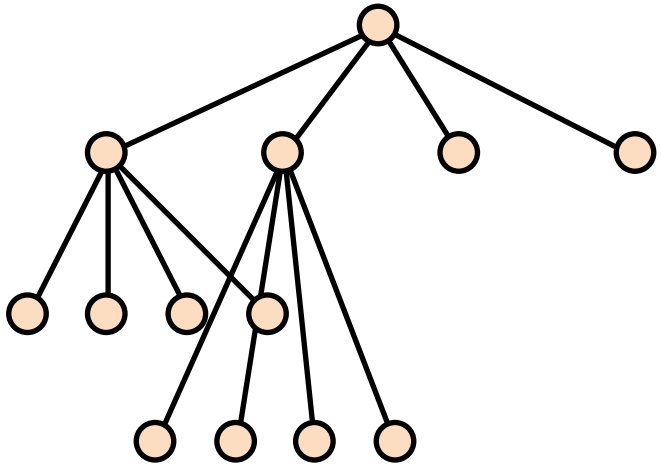


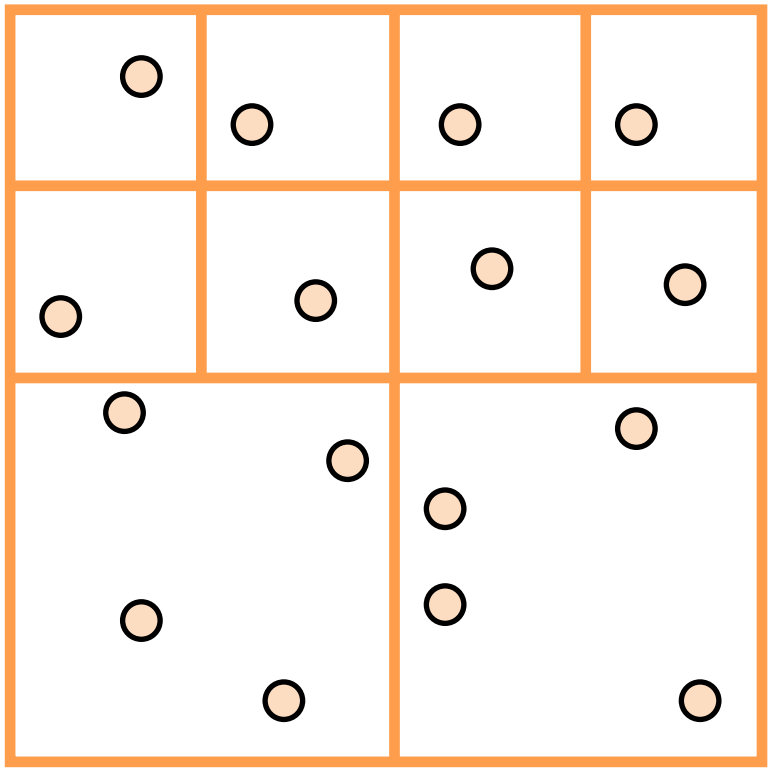
Quadtree



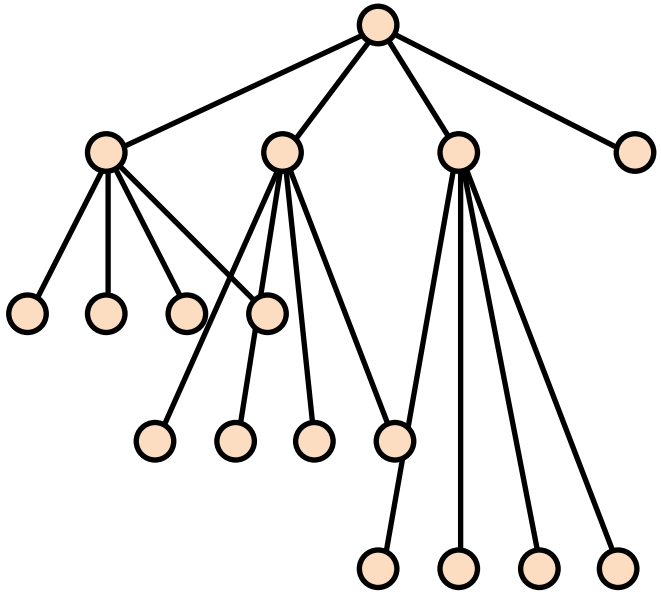


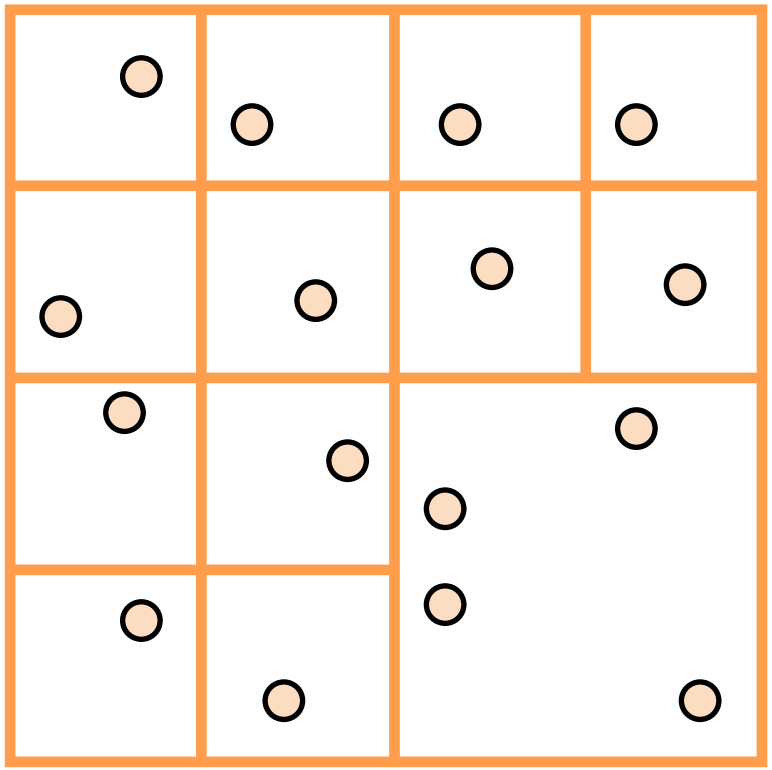
Quadtree



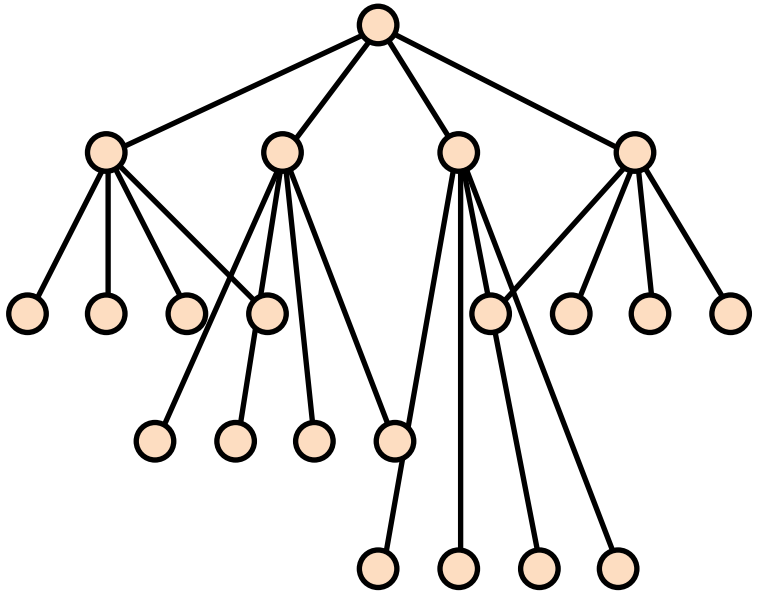


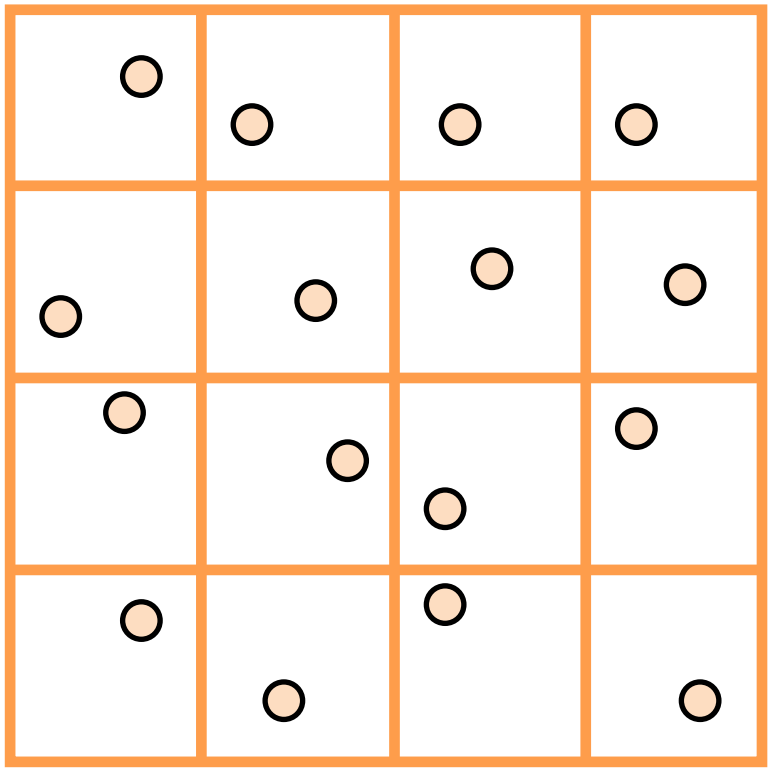
Quadtree



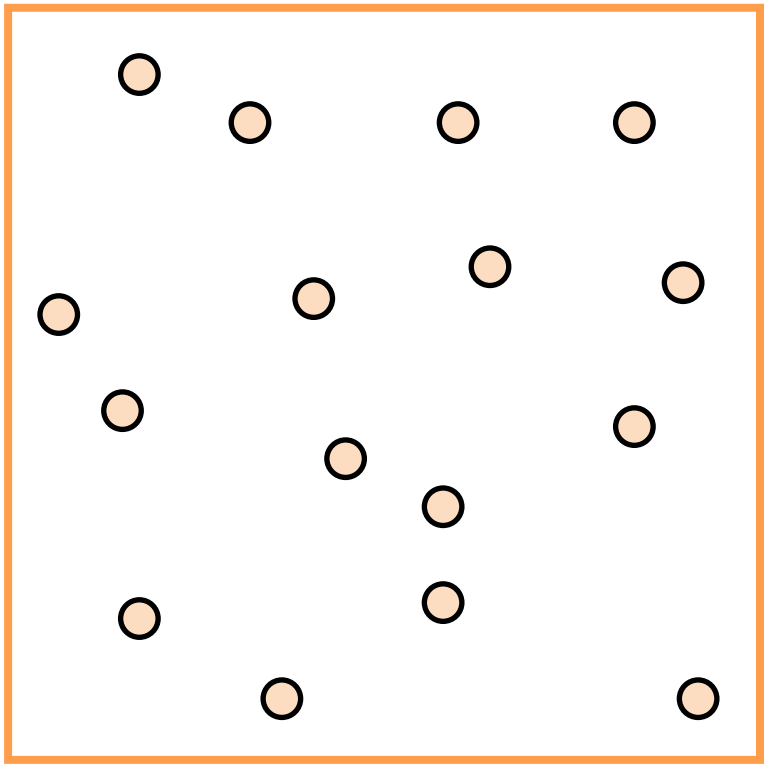


Quadtree

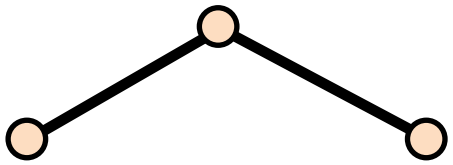


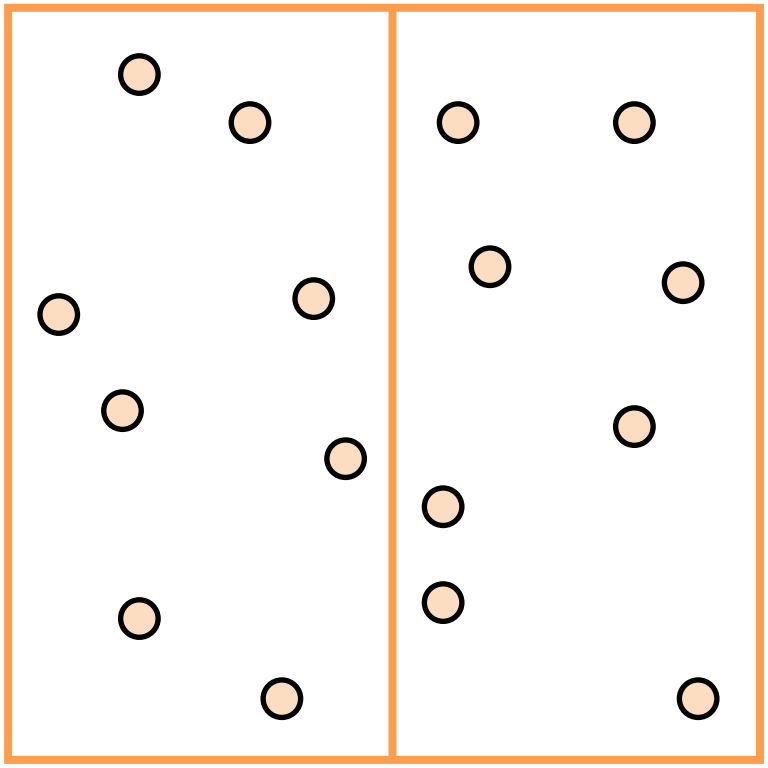


kD-Bäume

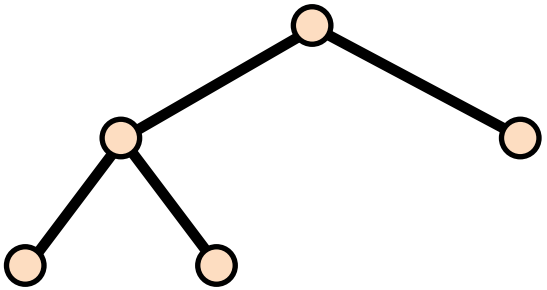


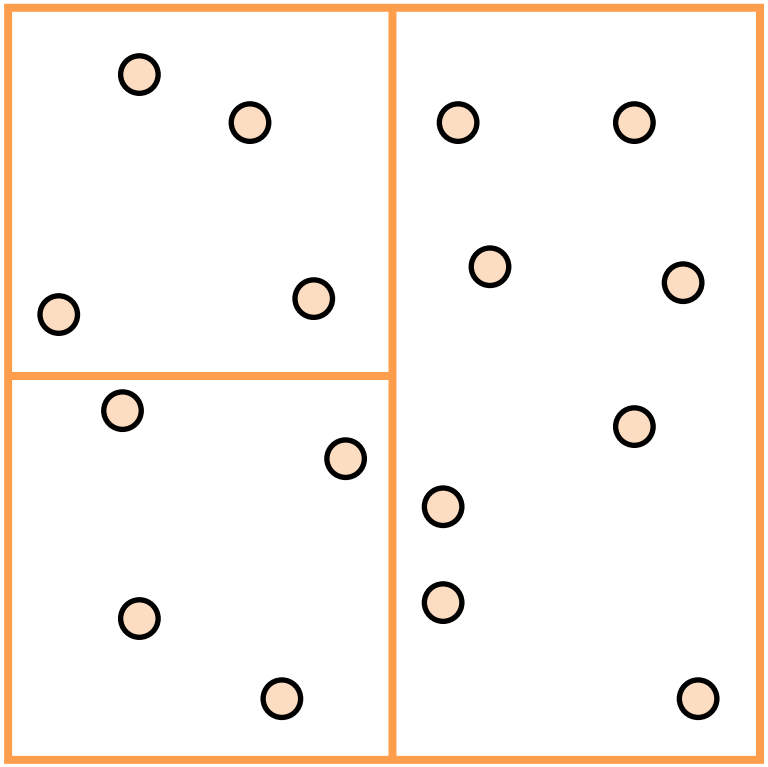
kD-Bäume



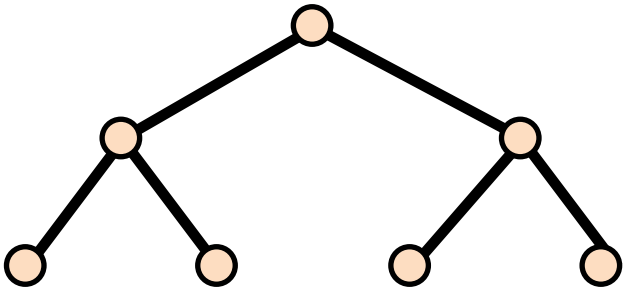


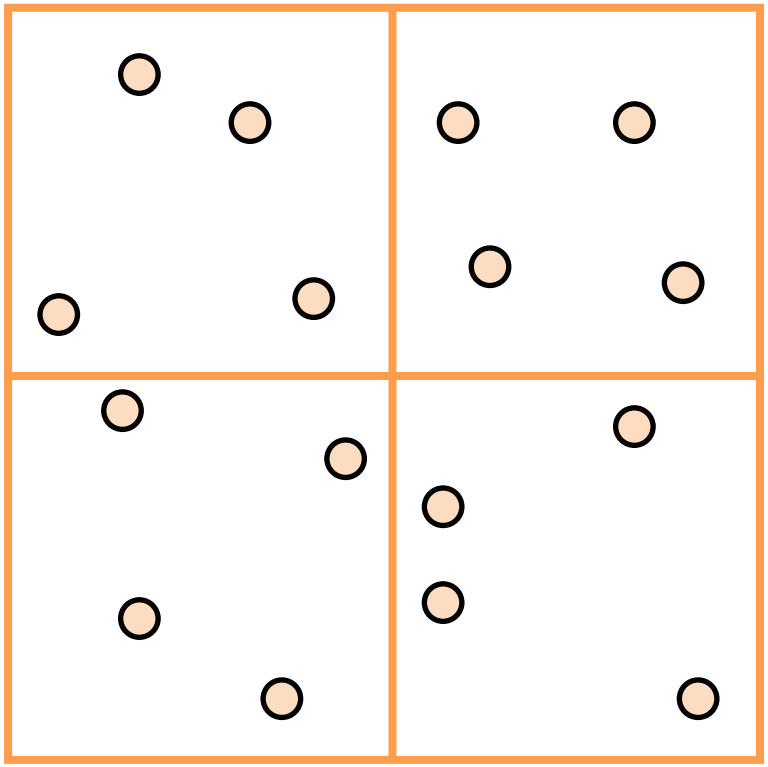
kD-Bäume



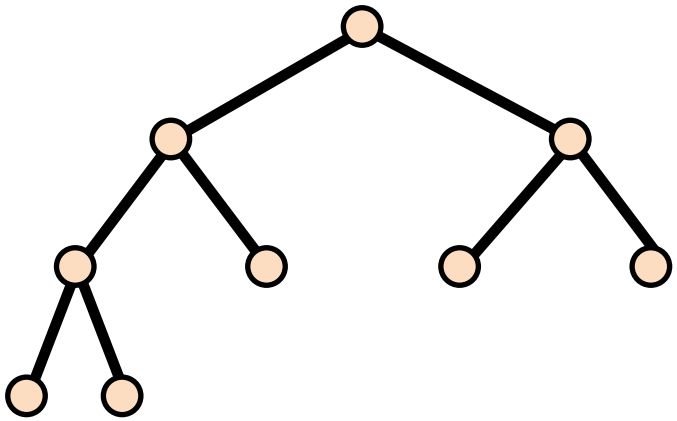


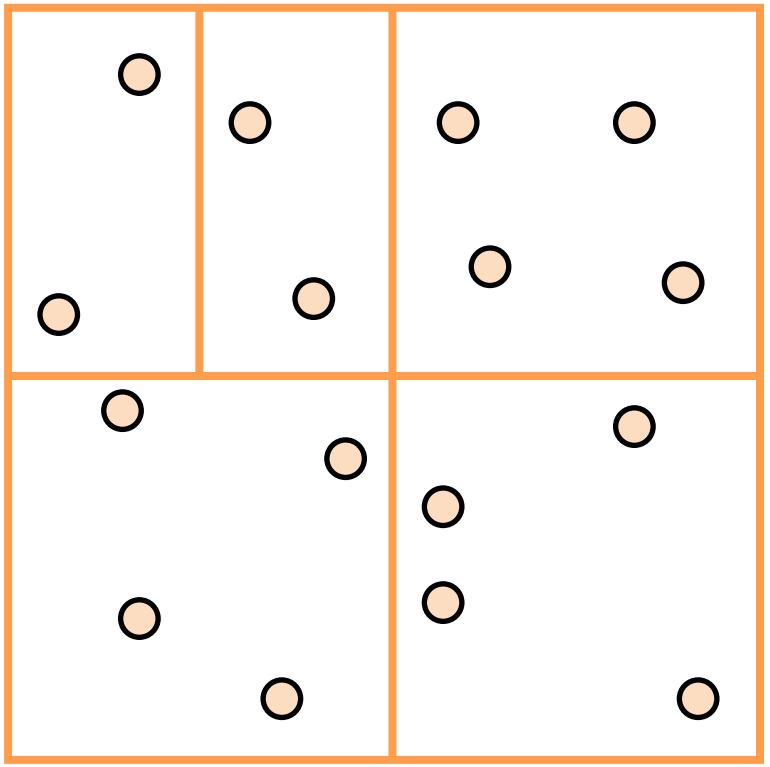
kD-Bäume



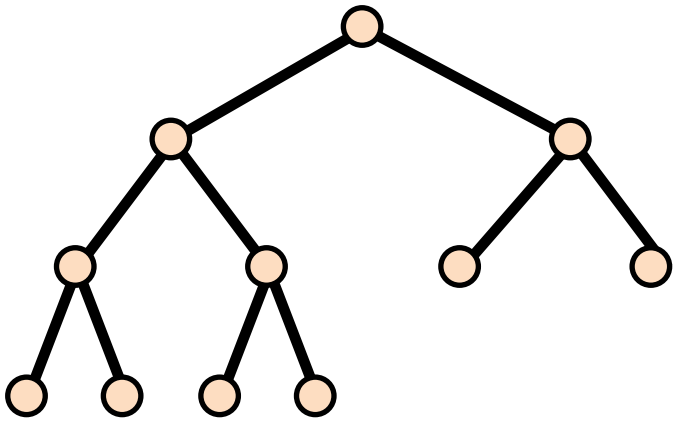


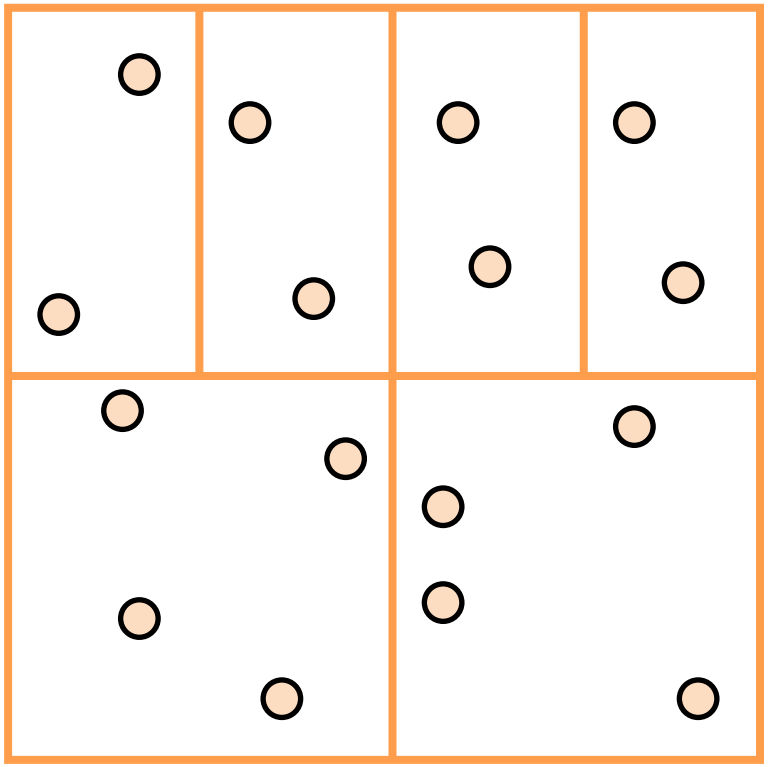
kD-Bäume



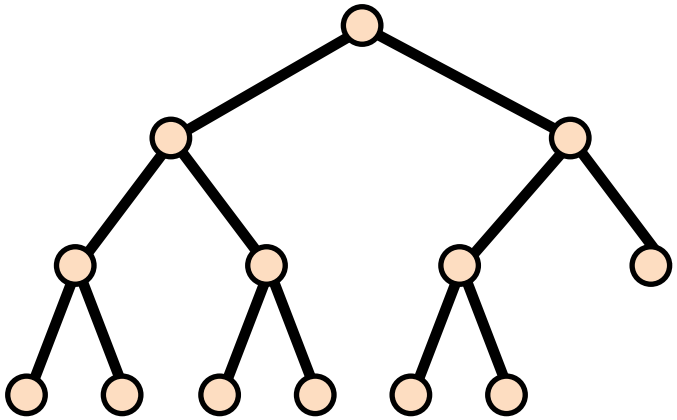


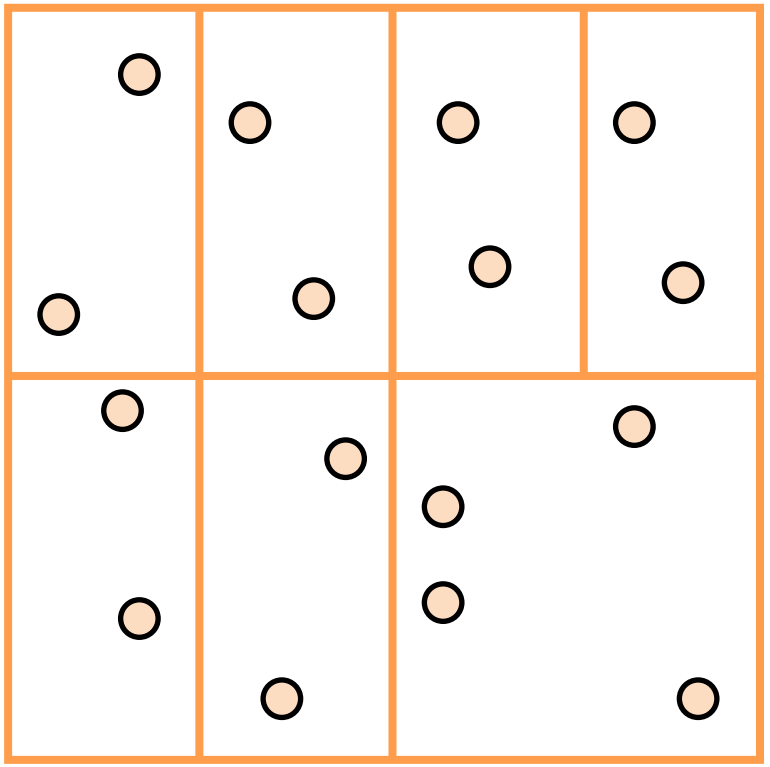
kD-Bäume



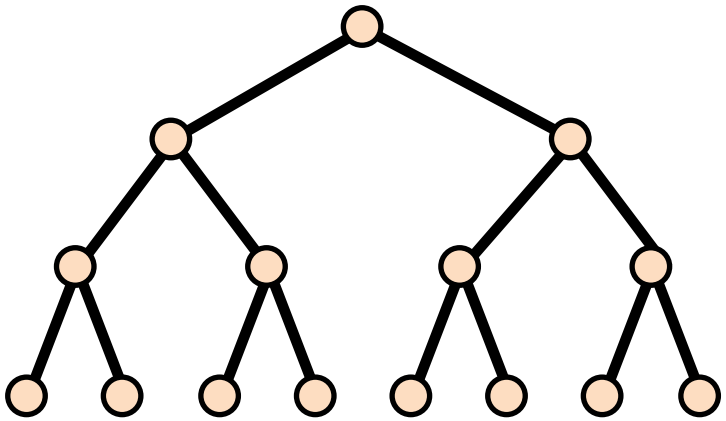


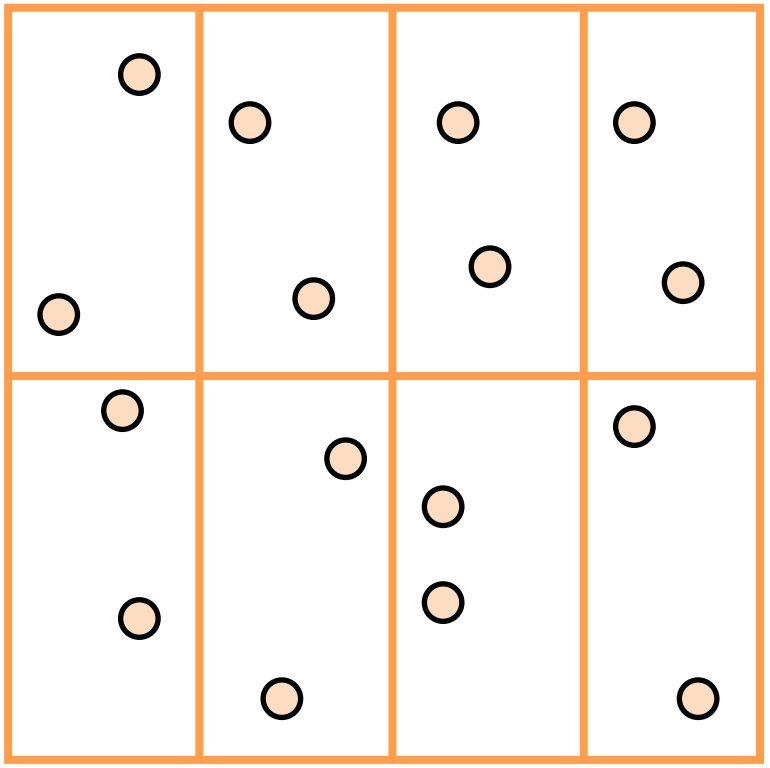
kD-Bäume



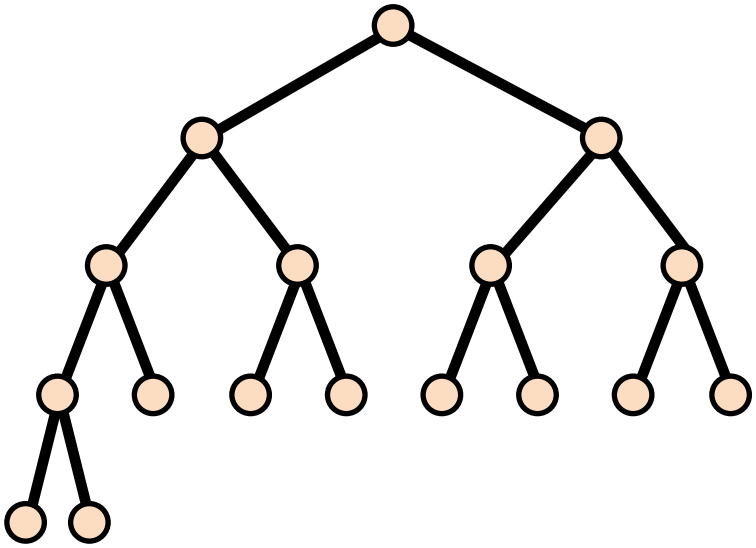


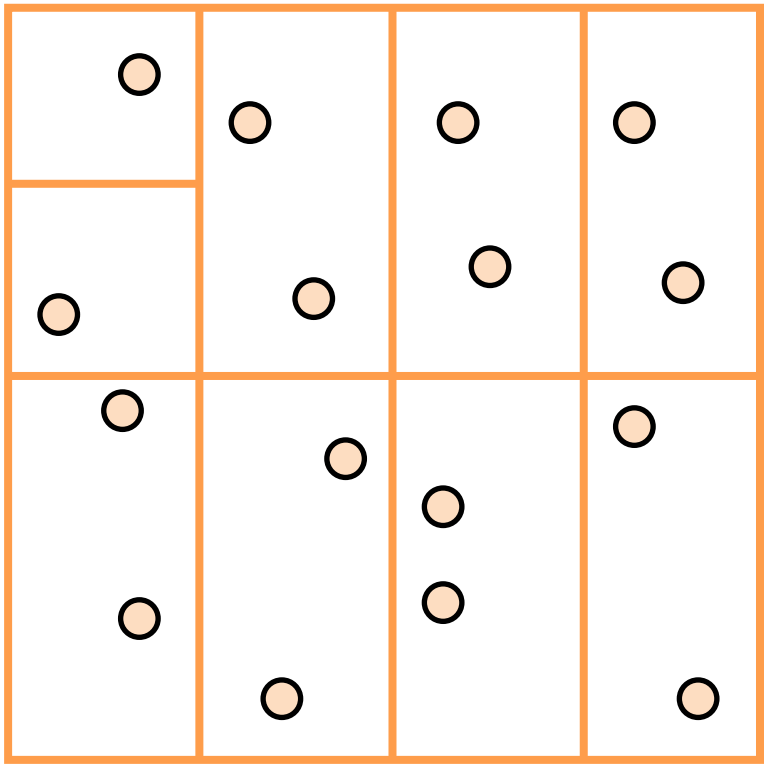
kD-Bäume



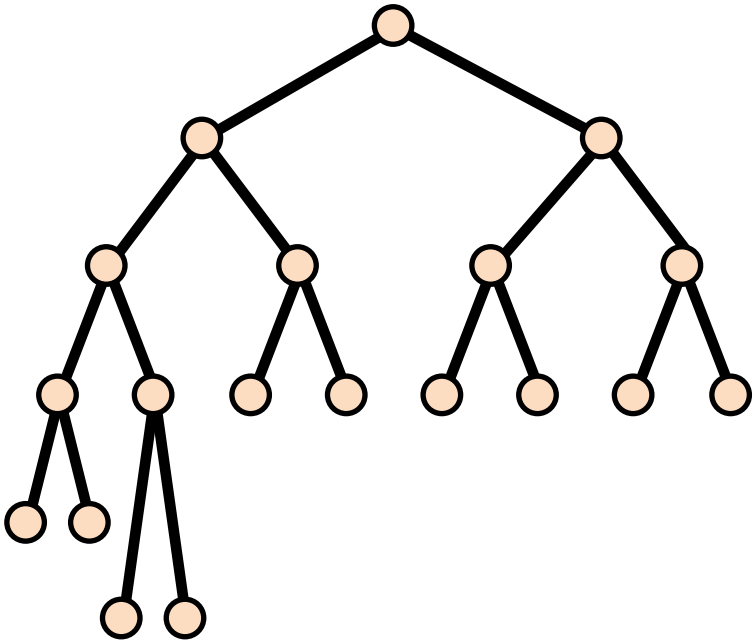


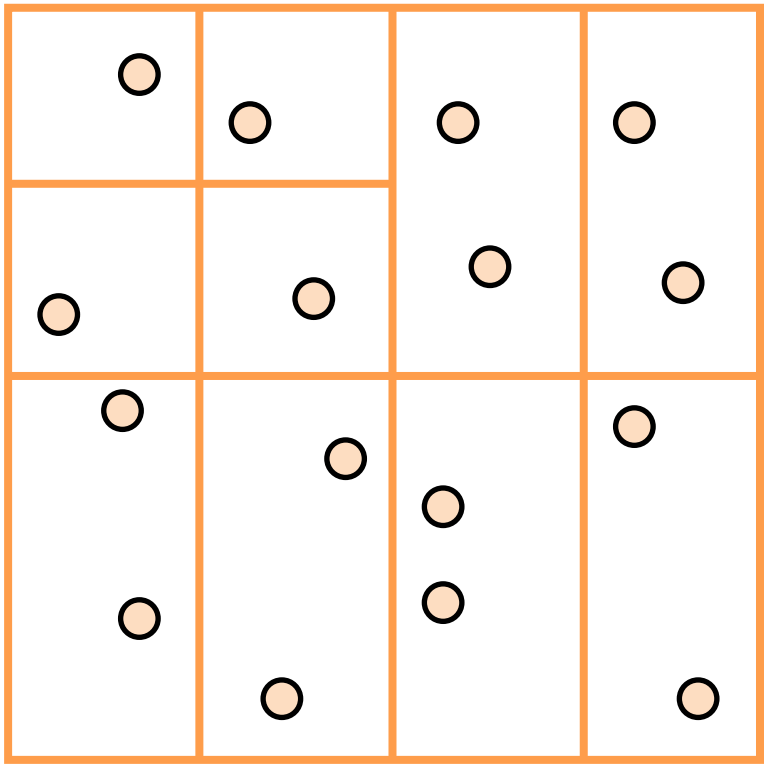
kD-Bäume



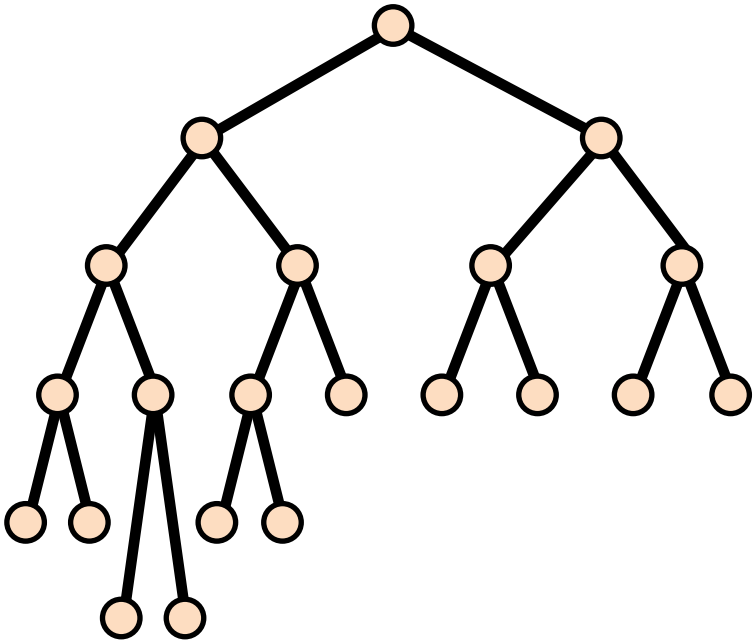


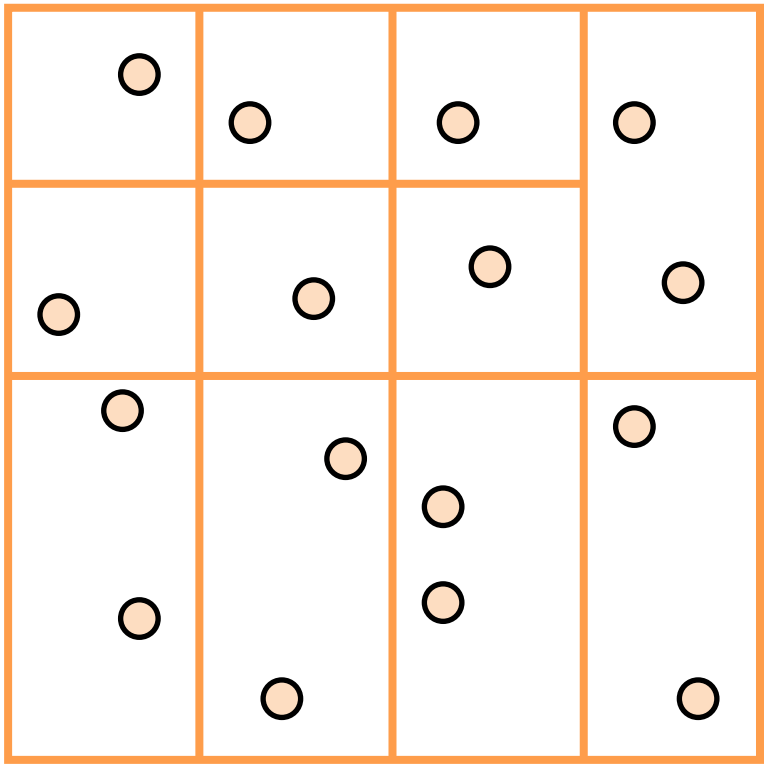
kD-Bäume



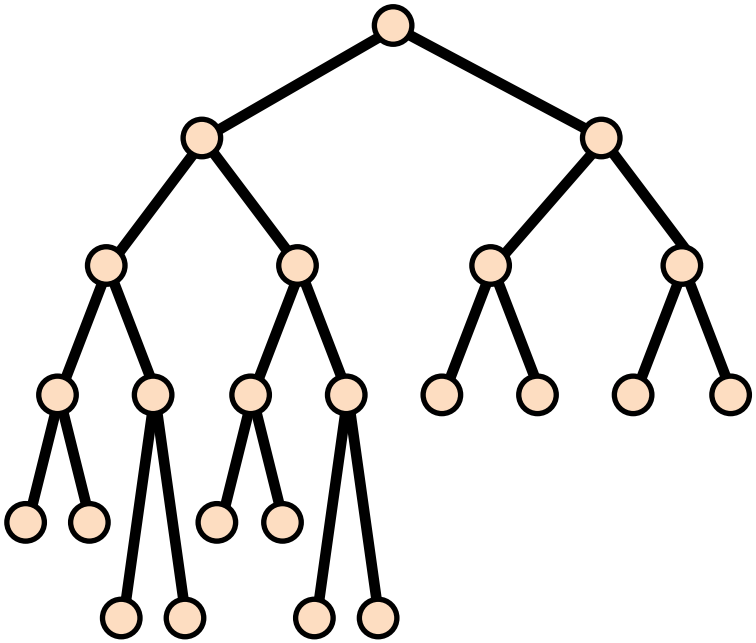


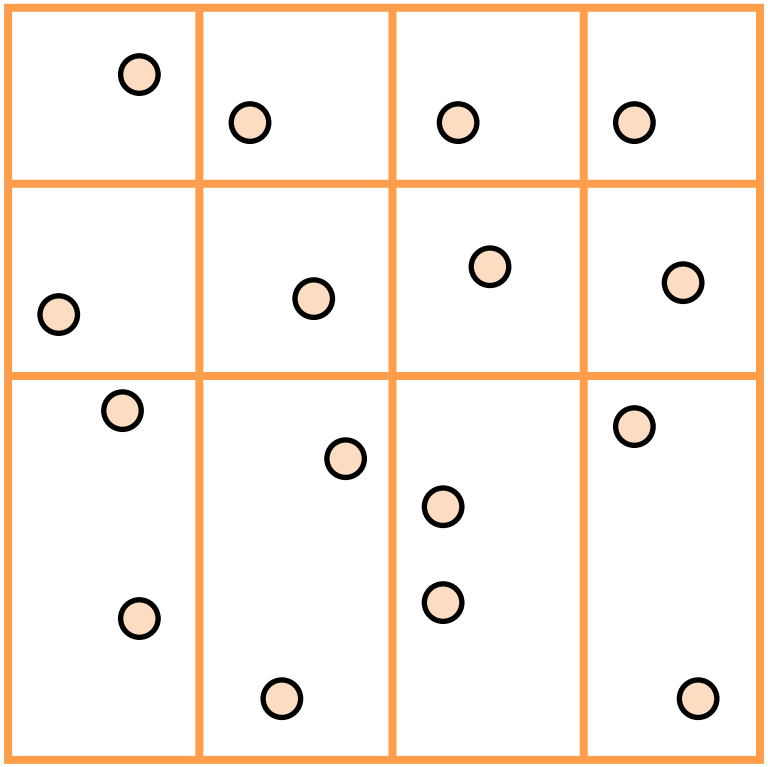
kD-Bäume



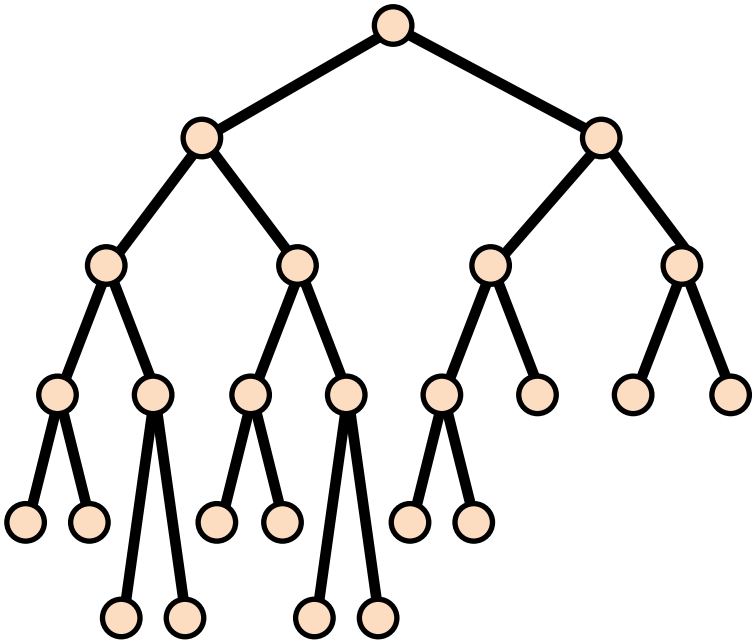


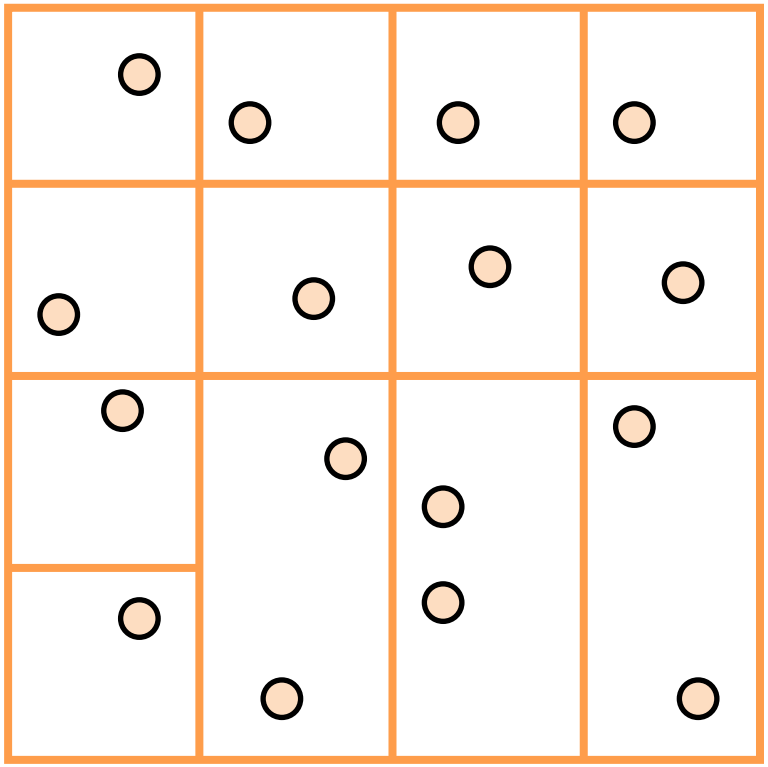
kD-Bäume



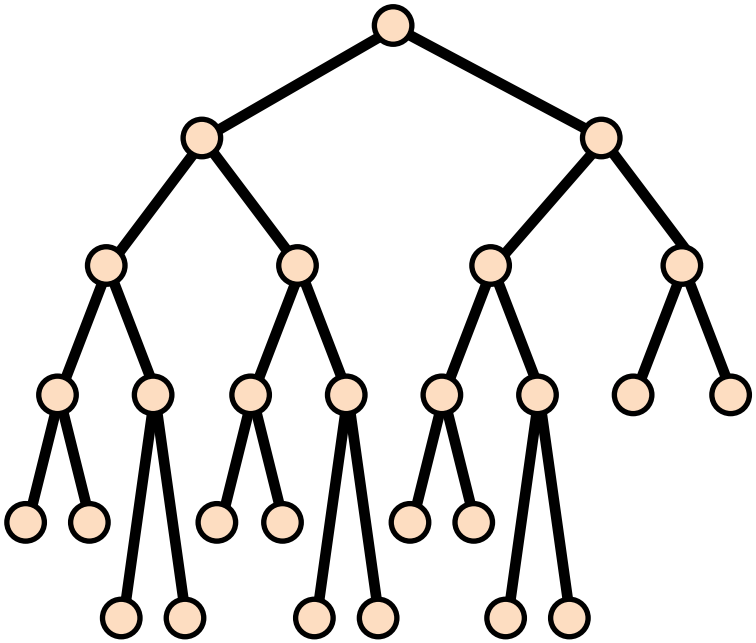


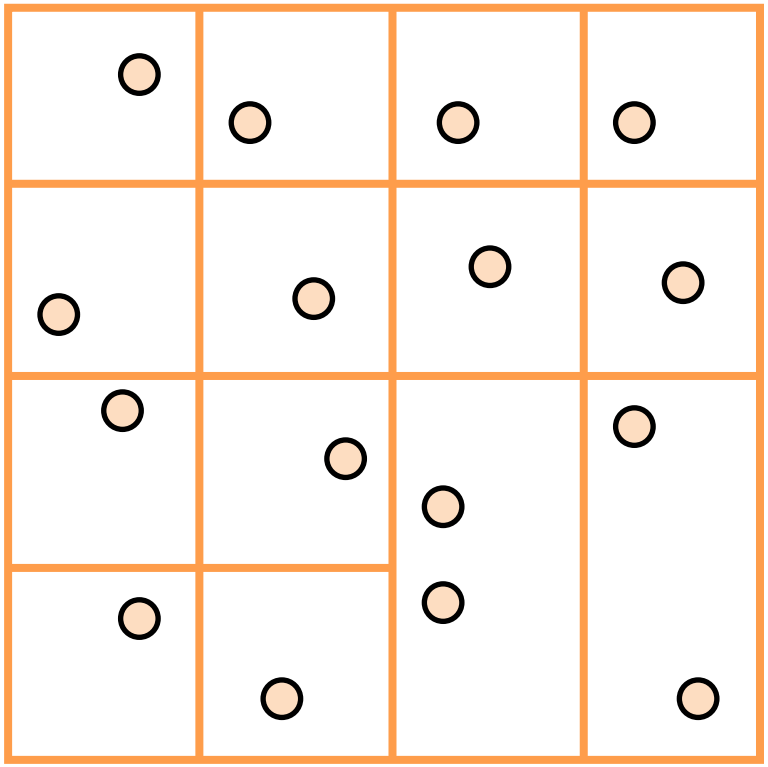
kD-Bäume



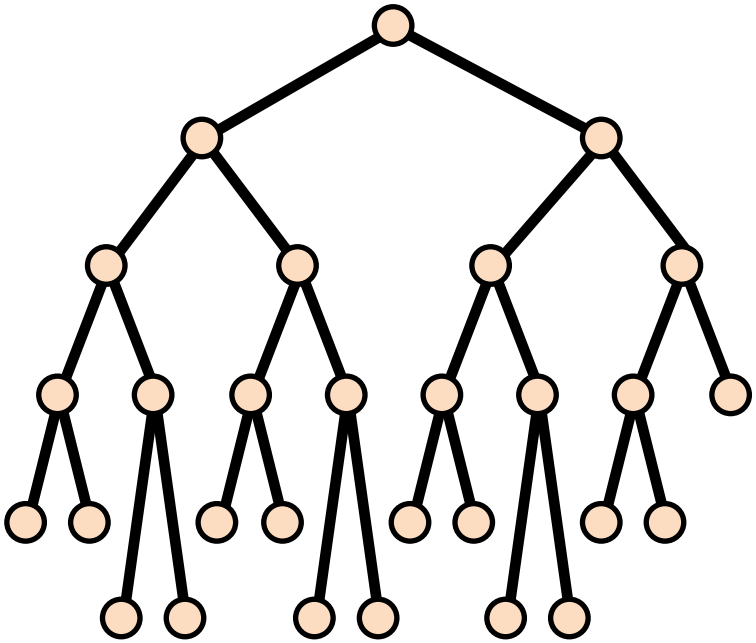


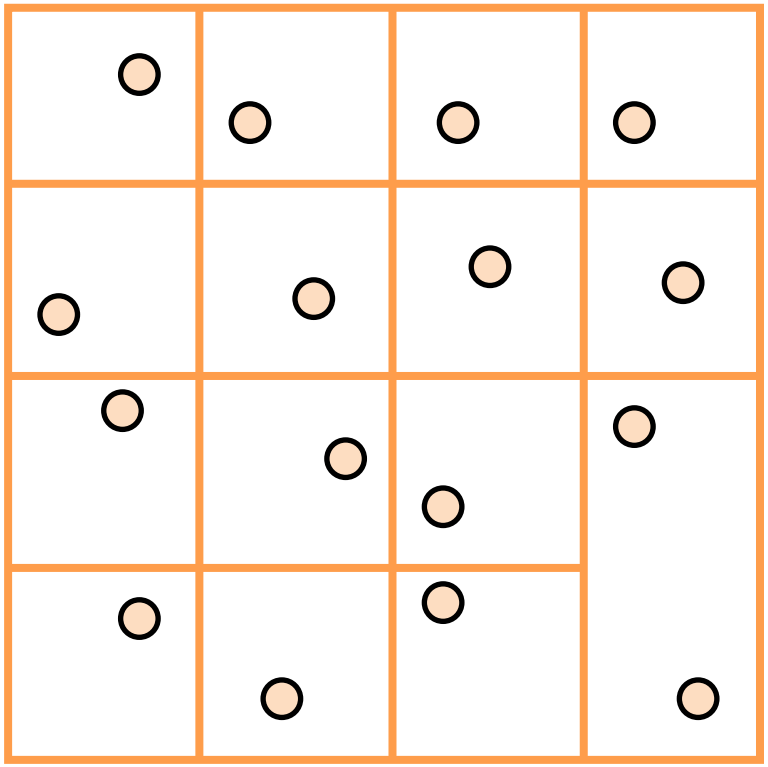
kD-Bäume



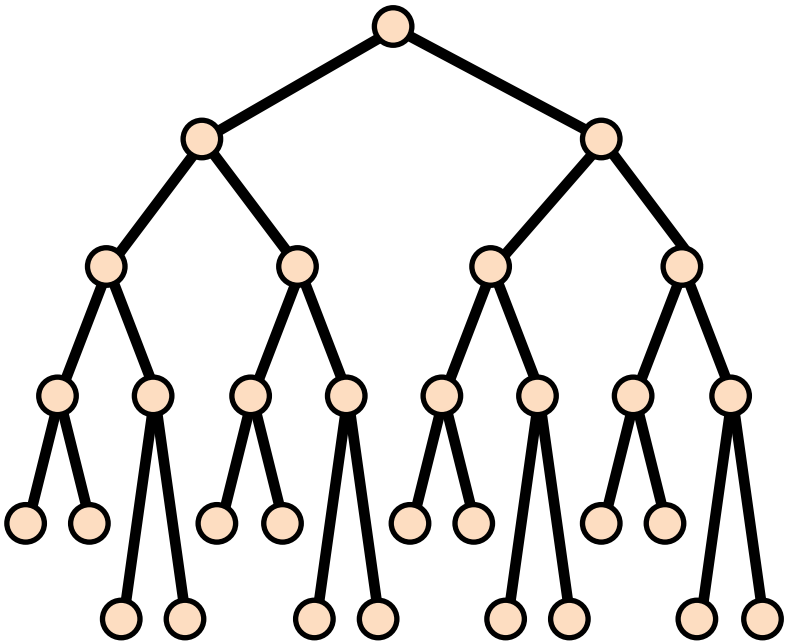


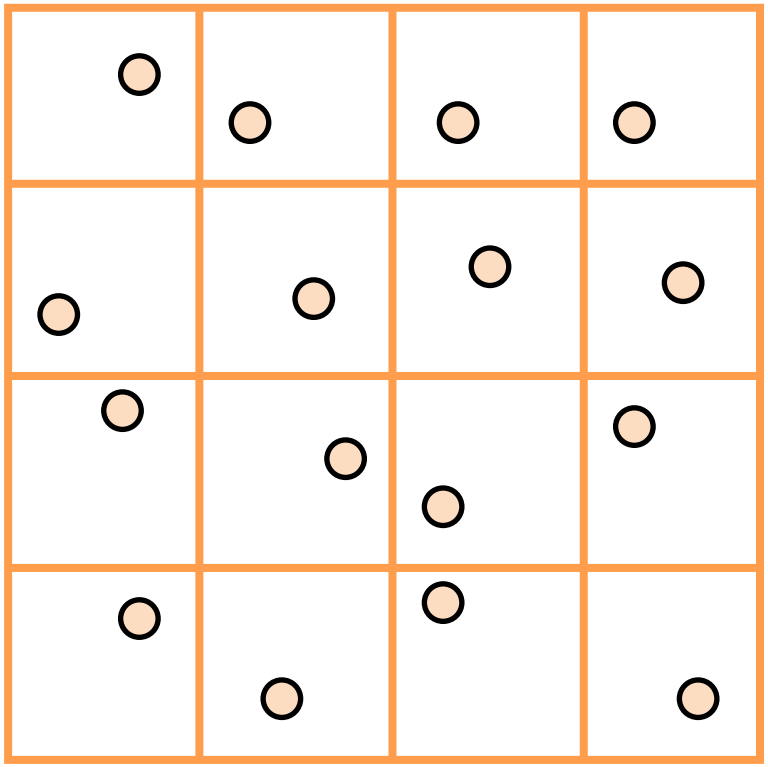
kD-Bäume





kD-Bäume





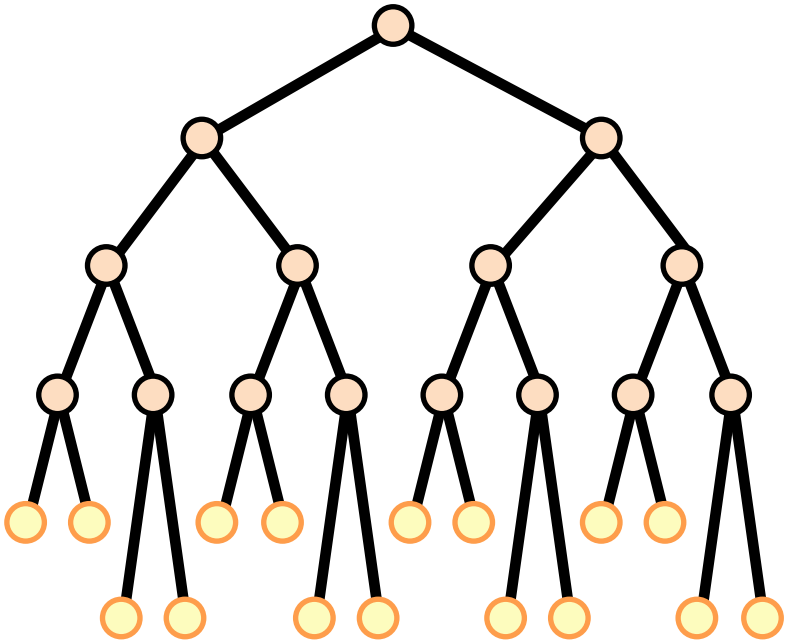
kD-Baum Suche

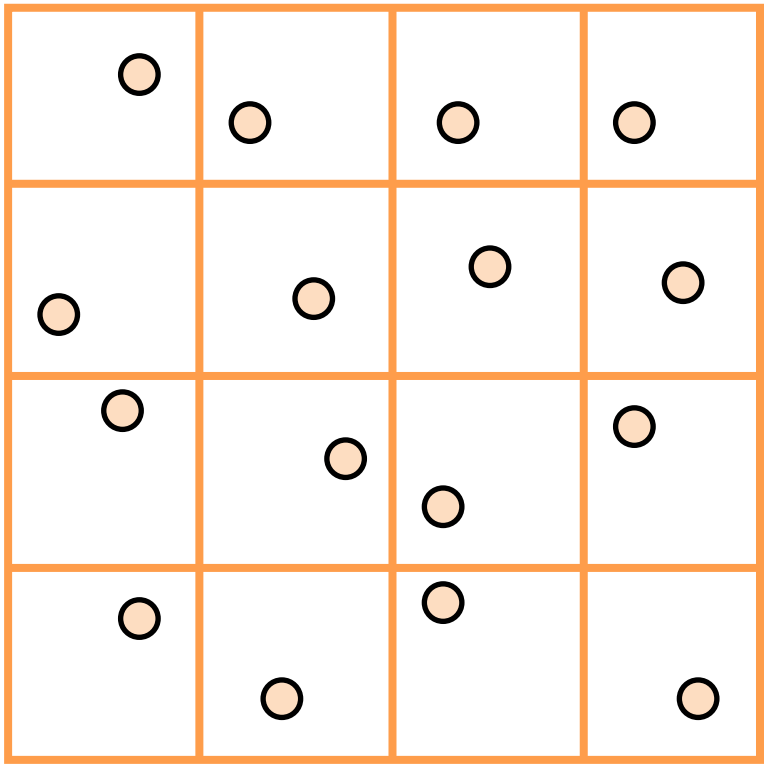
- In welcher Zelle liegt der Anfragepunkt
- In welchen Zellen muß nach dem nächsten Punkt gesucht werden
- Effizienter Auschluß weit entfernter Zellen

kD-Baum Suche

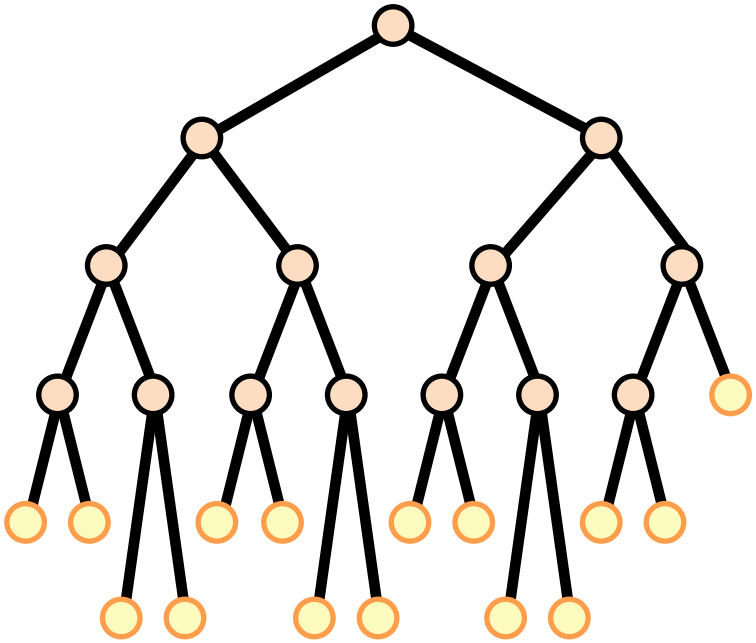
- Jedes kD-Baum Blatt enthält einen der gegebenen Punkte p_i
- Jeder innere kD-Baum Knoten enthält eine Ebene, die die Punkte im linken und rechten Teilbaum separiert.

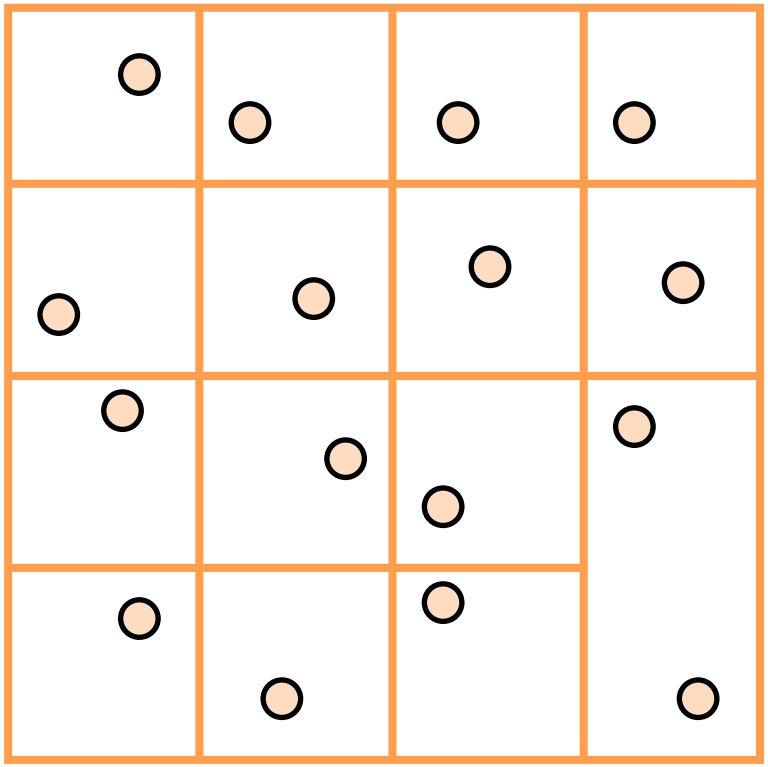
kD-Bäume



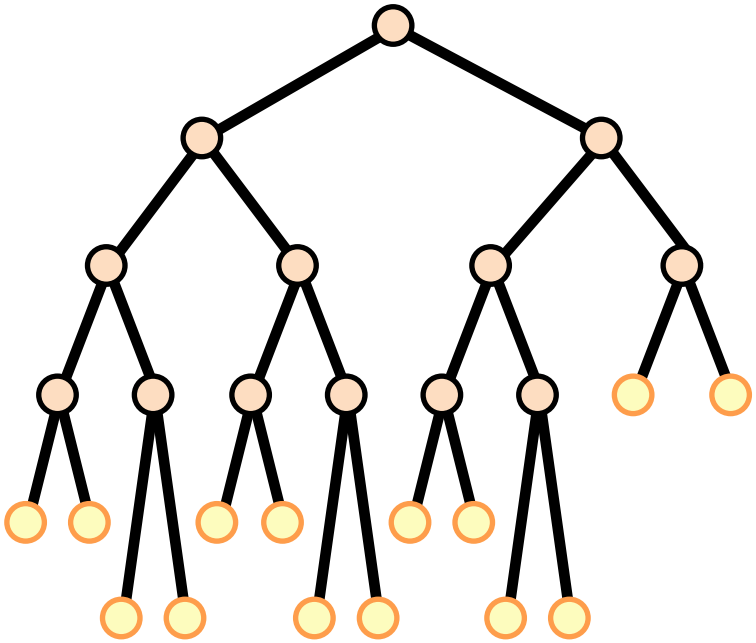


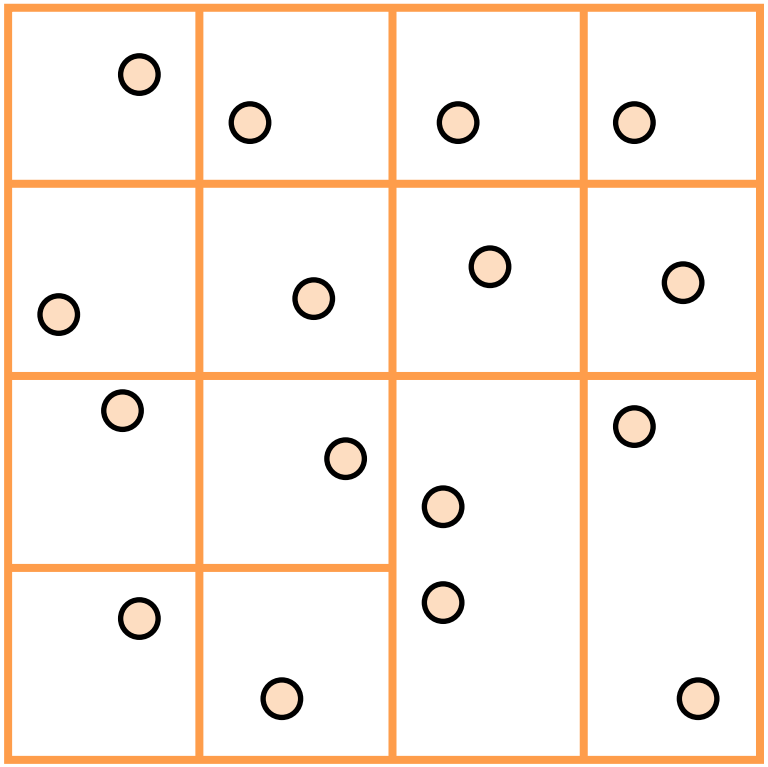
kD-Bäume



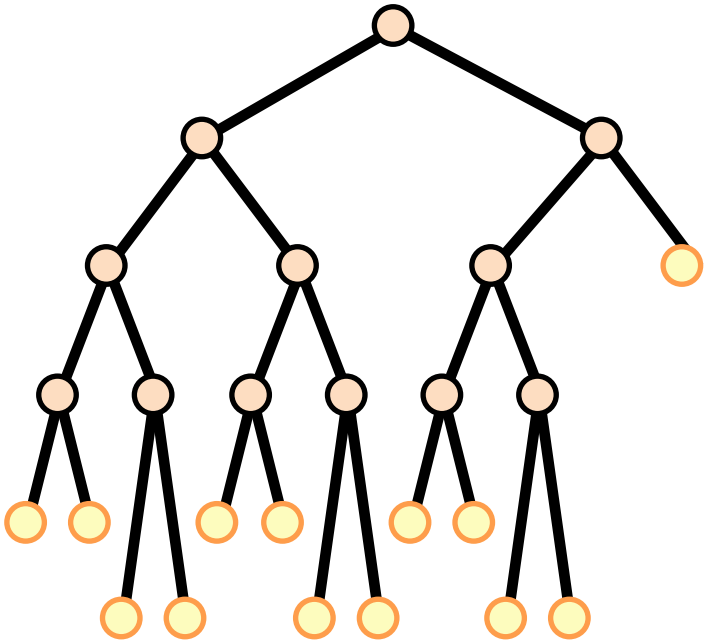


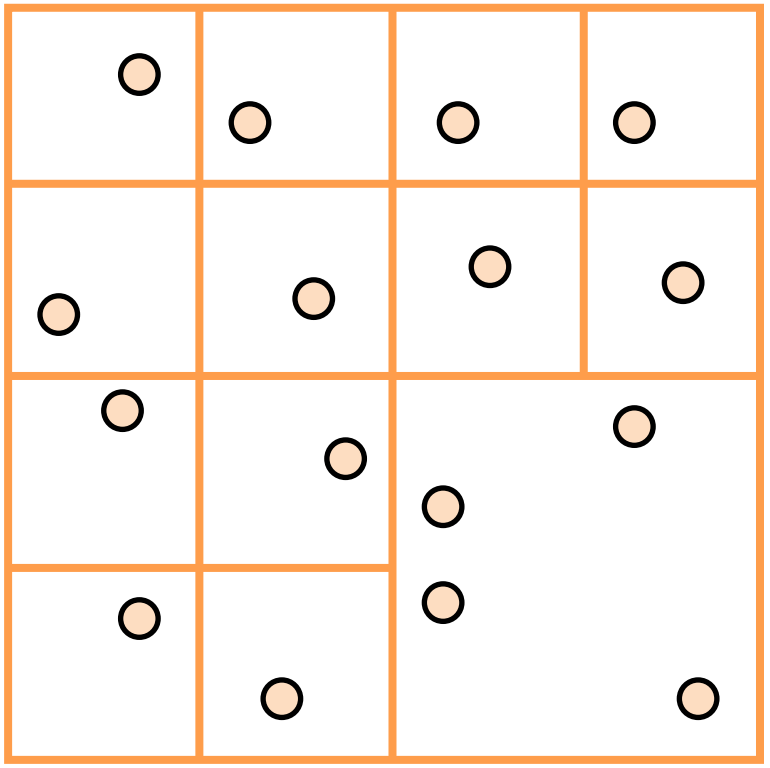
kD-Bäume





kD-Bäume





kD-Baum Suche

- Traverse(p,n)

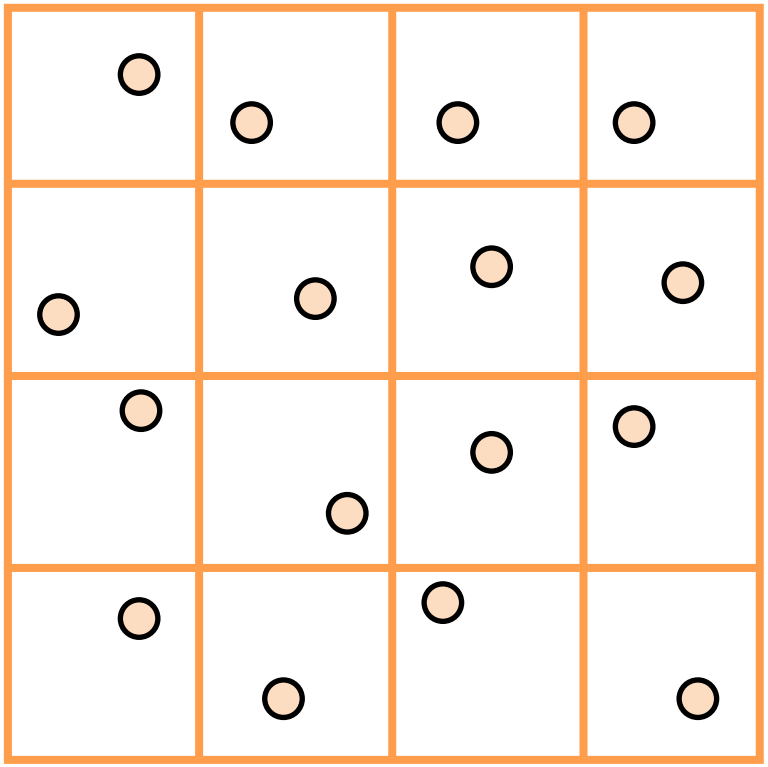
```
if n is a leaf-node then
    return ( dist(p,point[n]), point[n] )
else
    if p lies in positive half-space of n then
        (dist,near) ← Traverse(p,left[n])
    else
        (dist,near) ← Traverse(p,right[n])
    return (dist,near)
```

kD-Baum Suche

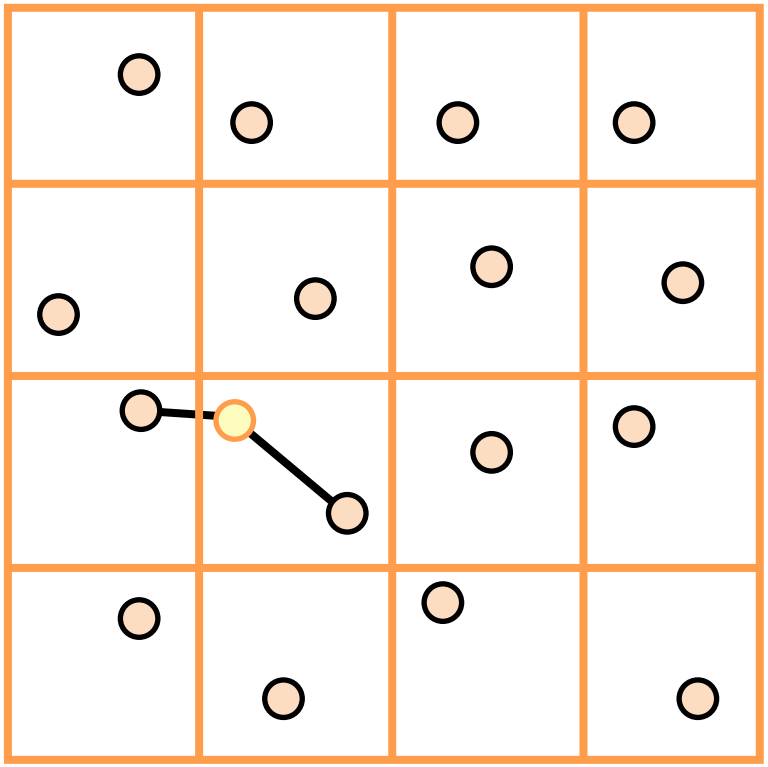
- Traverse(p,n)

```
if n is a leaf-node then
    return ( dist(p,point[n]), point[n] )
else
    if p·normal[n] − offset[n] ⩾ 0 then
        (dist,near) ← Traverse(p,left[n])
    else
        (dist,near) ← Traverse(p,right[n])
    return (dist,near)
```

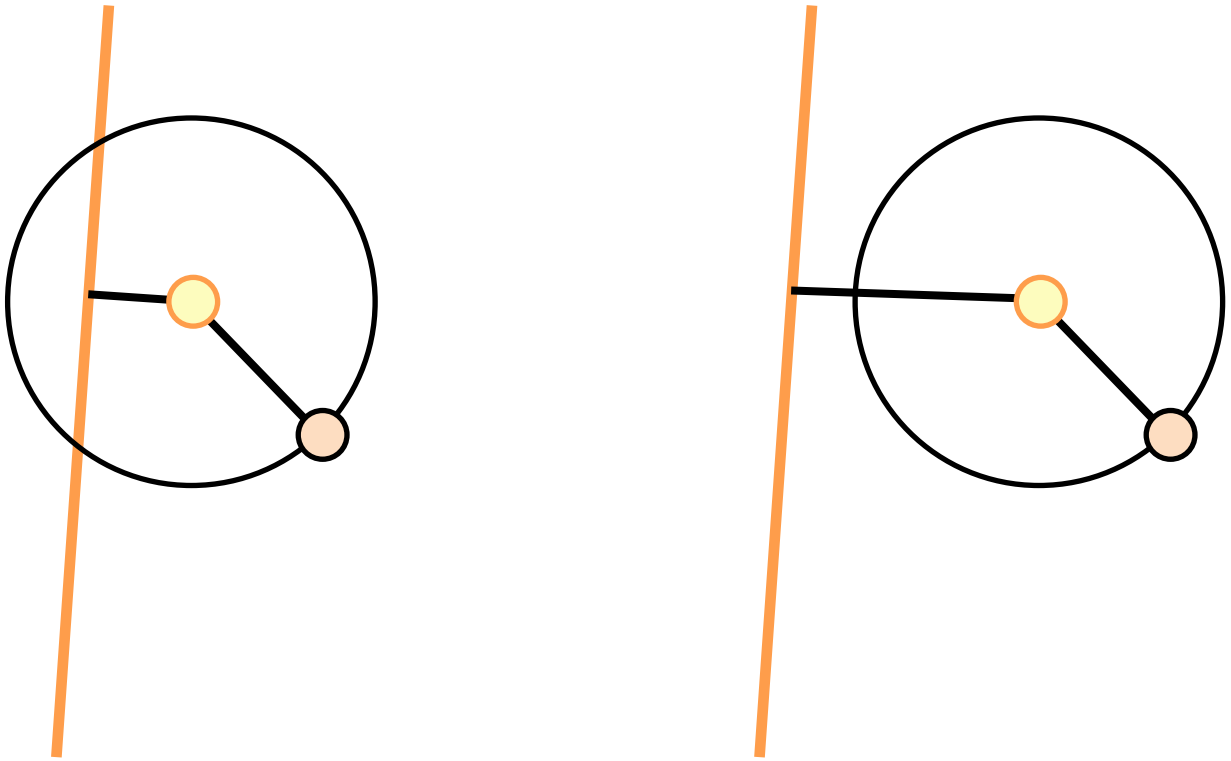
kD-Baum Suche



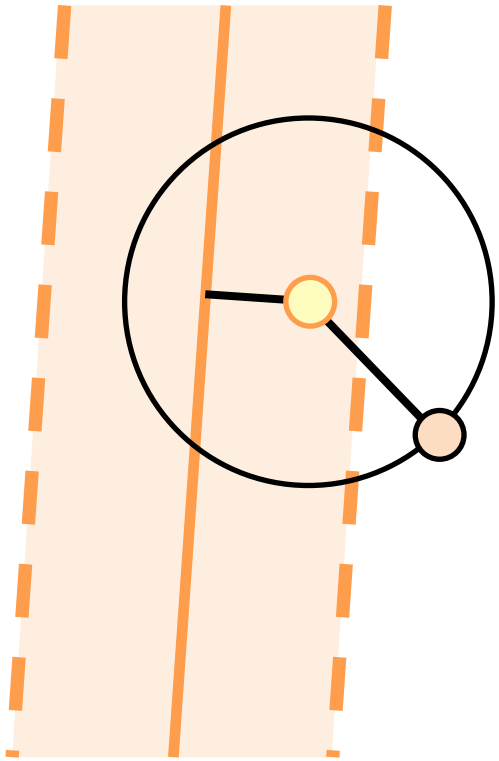
kD-Baum Suche

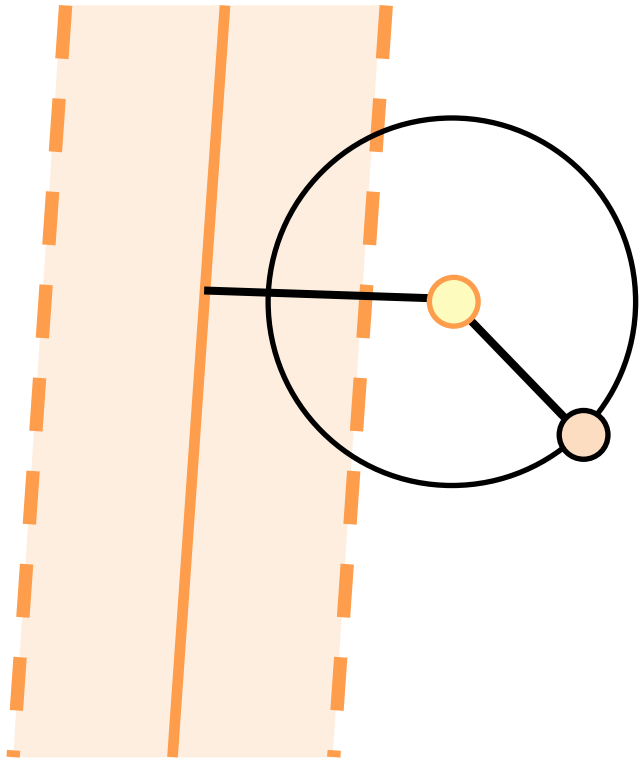


kD-Baum Suche



kD-Baum Suche





kD-Baum Suche

- Traverse(p,dist,near,n)

```
if n is a leaf-node then
    if dist(p,point[n]) < dist then
        return (dist(p,point[n]), point[n])
    else
        return (dist, near)
else
    if p·normal[n] − offset[n] ⩾ −dist then
        (dist,near) = Traverse(p,dist,near,left[n])
    if p·normal[n] − offset[n] ⩽ dist then
        (dist,near) = Traverse(p,dist,near,right[n])
    return (dist,near)
```

Aufwandsabschätzung

- Bei regelmäßig verteilten Datenpunkten ist der erwartete Aufwand O(log n)
- Der Algorithmus funktioniert auch für allgemeinere BSP-Bäume, bei denen in jedem Knoten eine beliebige separierende Ebene definiert wird.

k nächste Punkte

- Geg. Punktmenge $p_1, \ldots, p_n$
- Anfrage: (x,y) [hier: R^2, allg.: R^k]
- Suche die k besten Kandidaten
- Verwalte die aktuell k Besten in einem Heap
- Neue Kandidaten verdrängen die alten

k nächste Punkte

- Nearest(p,neighbors,n)
  if n is a leaf-node then
    if dist(p,point[n]) < neighbors.top() then
      remove_top(neighbors)
      add(neigbors, dist(p,point[n]), point[n])
  else
    if p·normal[n] − offset[n] ⩾ −neighbors.top() then
      neighbors ← Traverse(p,neighbors,left[n])
    if p·normal[n] − offset[n] ⩽ neighbors.top() then
      neighbors ← Traverse(p,neighbors,right[n])
  return neighbors

Aufwandsabschätzung

- Fast der gleiche Aufwand wie bei der einfachen Suche: $O(\log n \times \log k)$

Range Queries

- Bereichsabfragen
  - Suche alle Punkte in einem Kreis (x,y,r)
  - Suche alle Punkte in einem Intervall $[a,b] \times [c,d]$
  - ...
- Einfache Modifikation des Suchalgorithmus

Range Queries

- Traverse(p,maxdist,n)

```
if n is a leaf-node then
    if dist(p,point[n]) < maxdist then
        output point[n]
else
    if p·normal[n]−offset[n] ⩾ −maxdist then
        Traverse(p,maxdist,left[n])
    if p·normal[n]−offset[n] ⩽  maxdist then
        Traverse(p,maxdist,right[n])
```

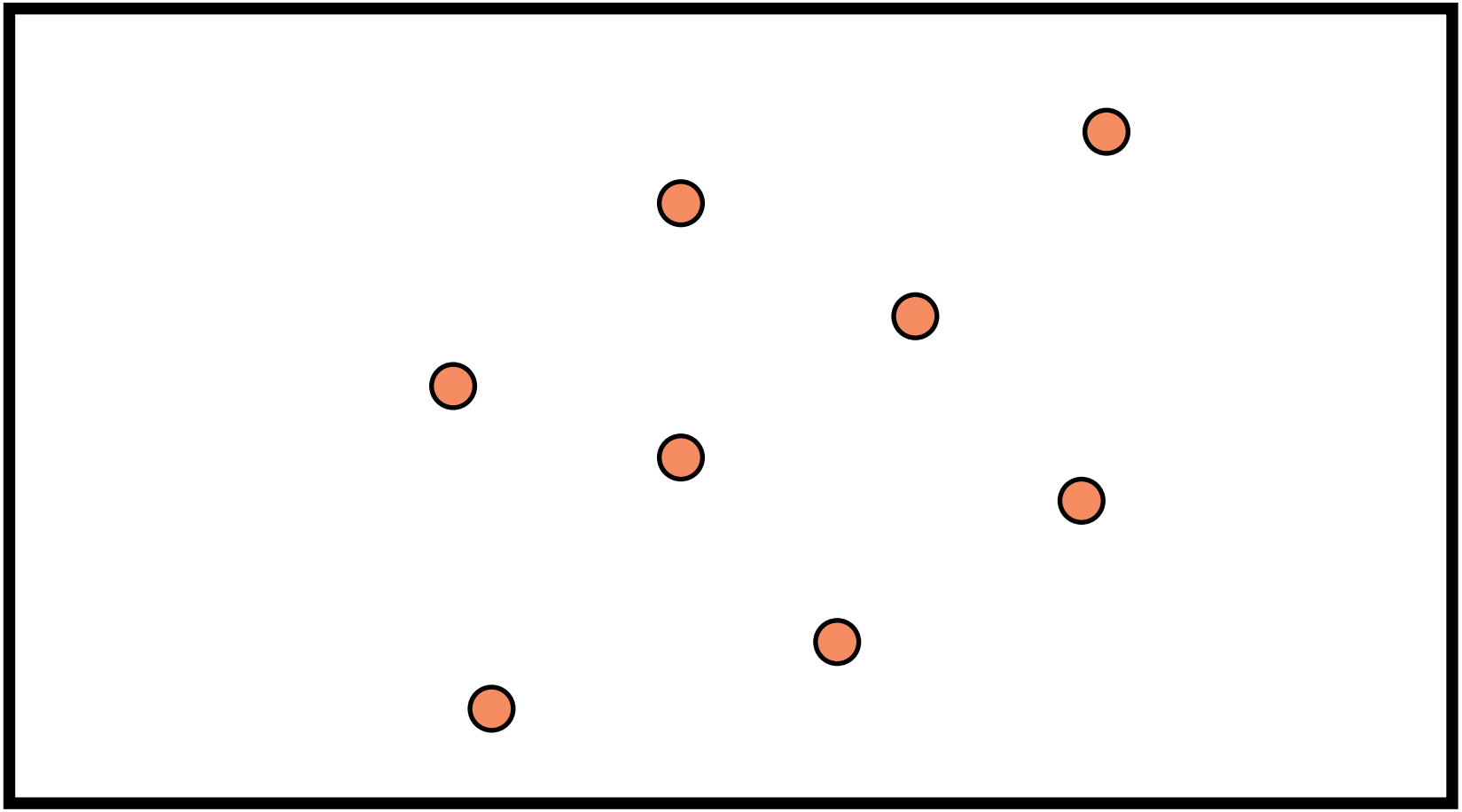
Abstandsdiagramme

- Bei häufigen Nachbarschafts-Anfragen kann der Zugriff durch Vorberechnung von Abstandsdiagrammen beschleunigt werden
- Diese zerlegen den R^n (hier den R^2) in kleine Bereiche, für die die Nachbarschaften jeweils vorberechnet werden können

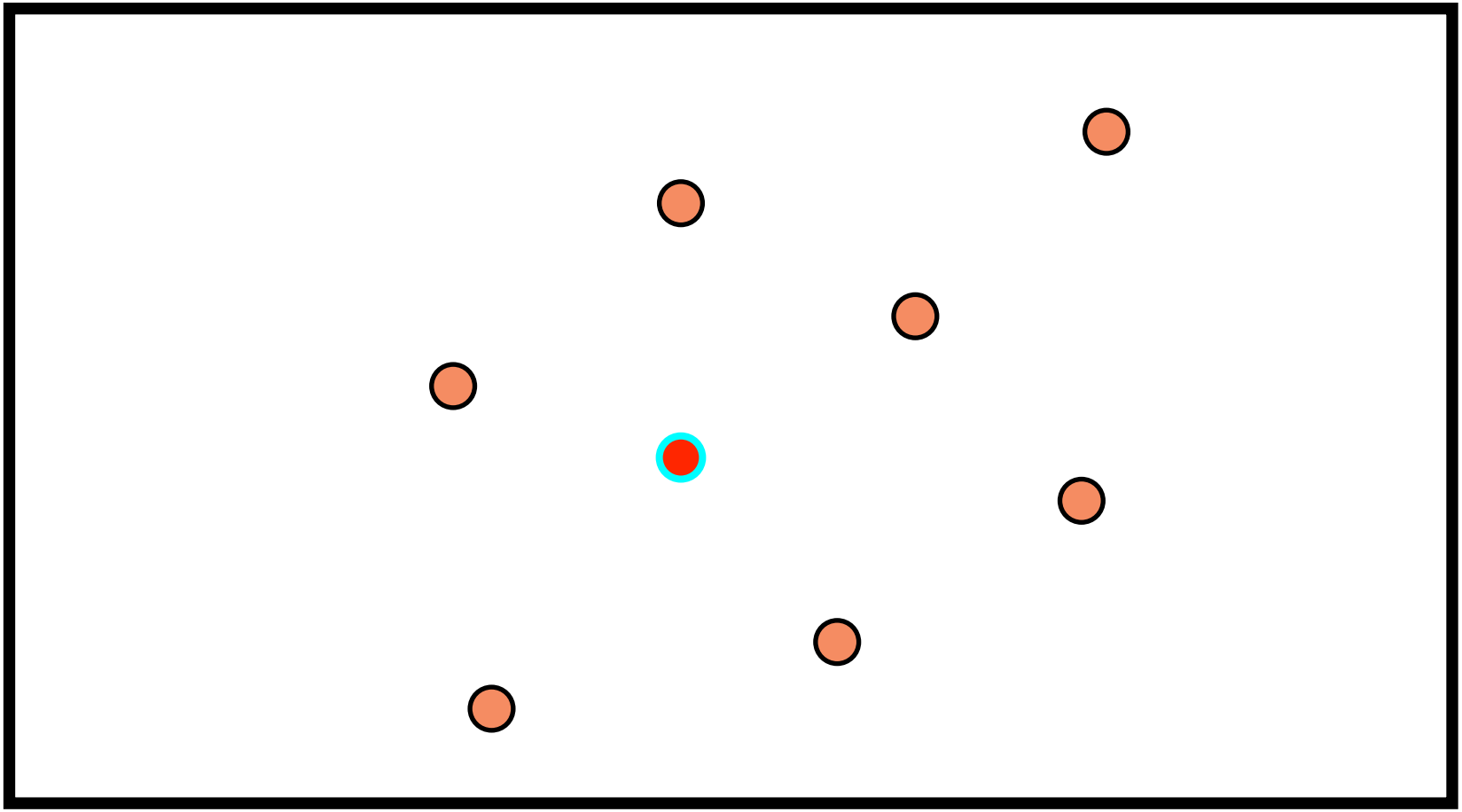
Voronoi-Gebiete

- Geg. Punktmenge $p_1, \ldots, p_n$
- Das Voronoi-Gebiet zu einem Punkt p_i enthält den Bereich des R^2, der näher an p_i als an jedem anderen p_j liegt
- $V(p_i) = \{ q \mid \forall\ j \neq i : dist(q,p_i) < dist(q,p_j) \}$

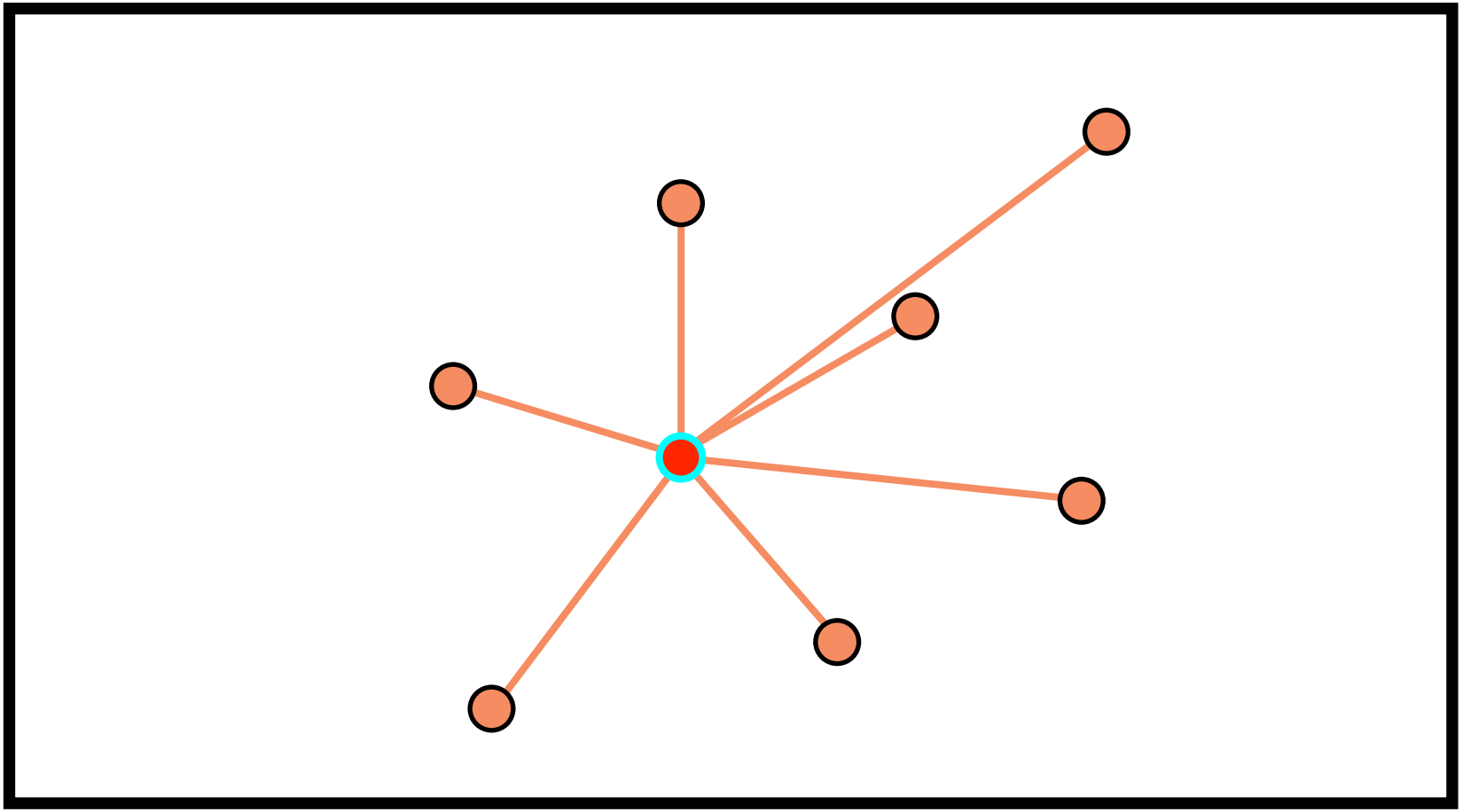
Voronoi-Gebiete



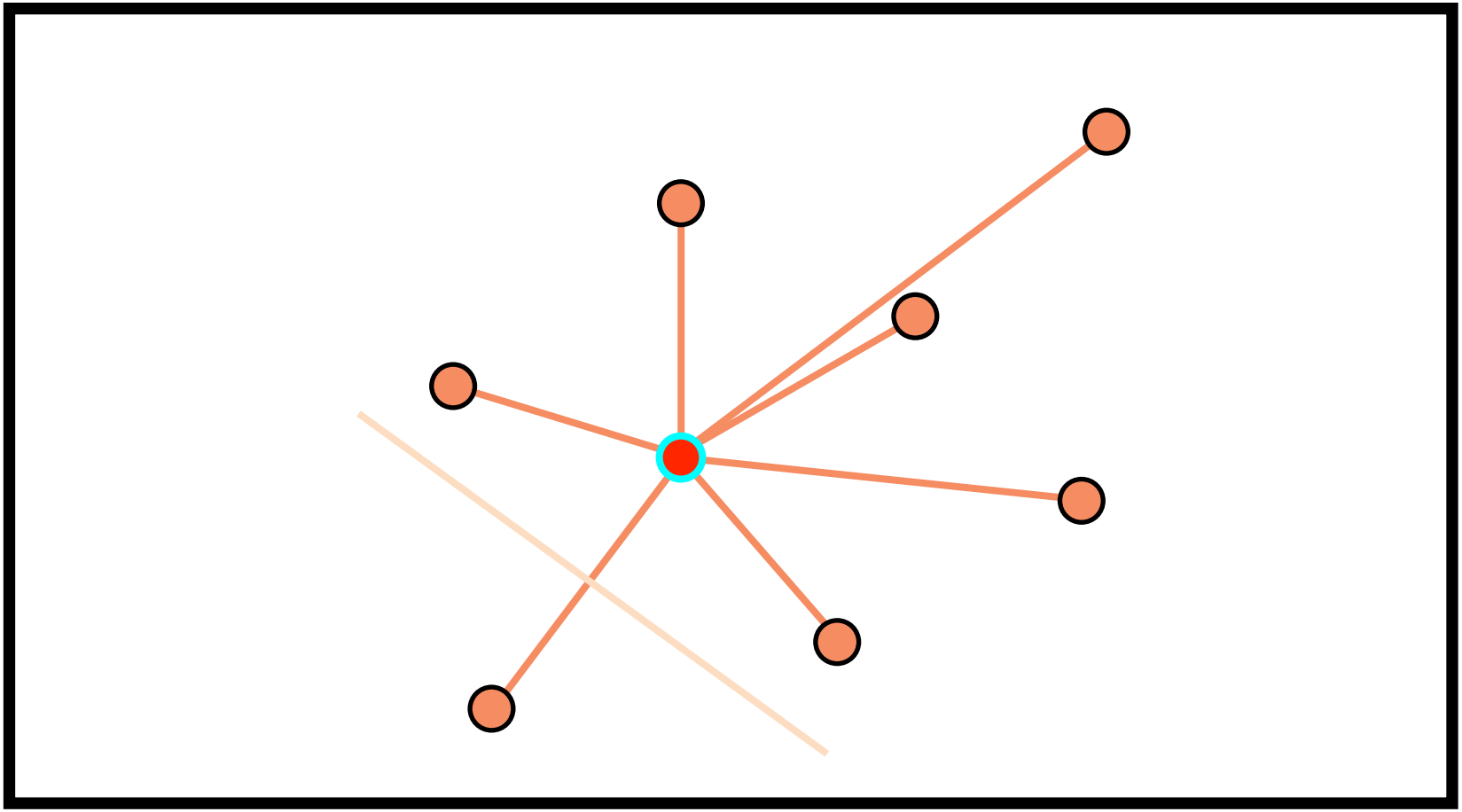
Voronoi-Gebiete



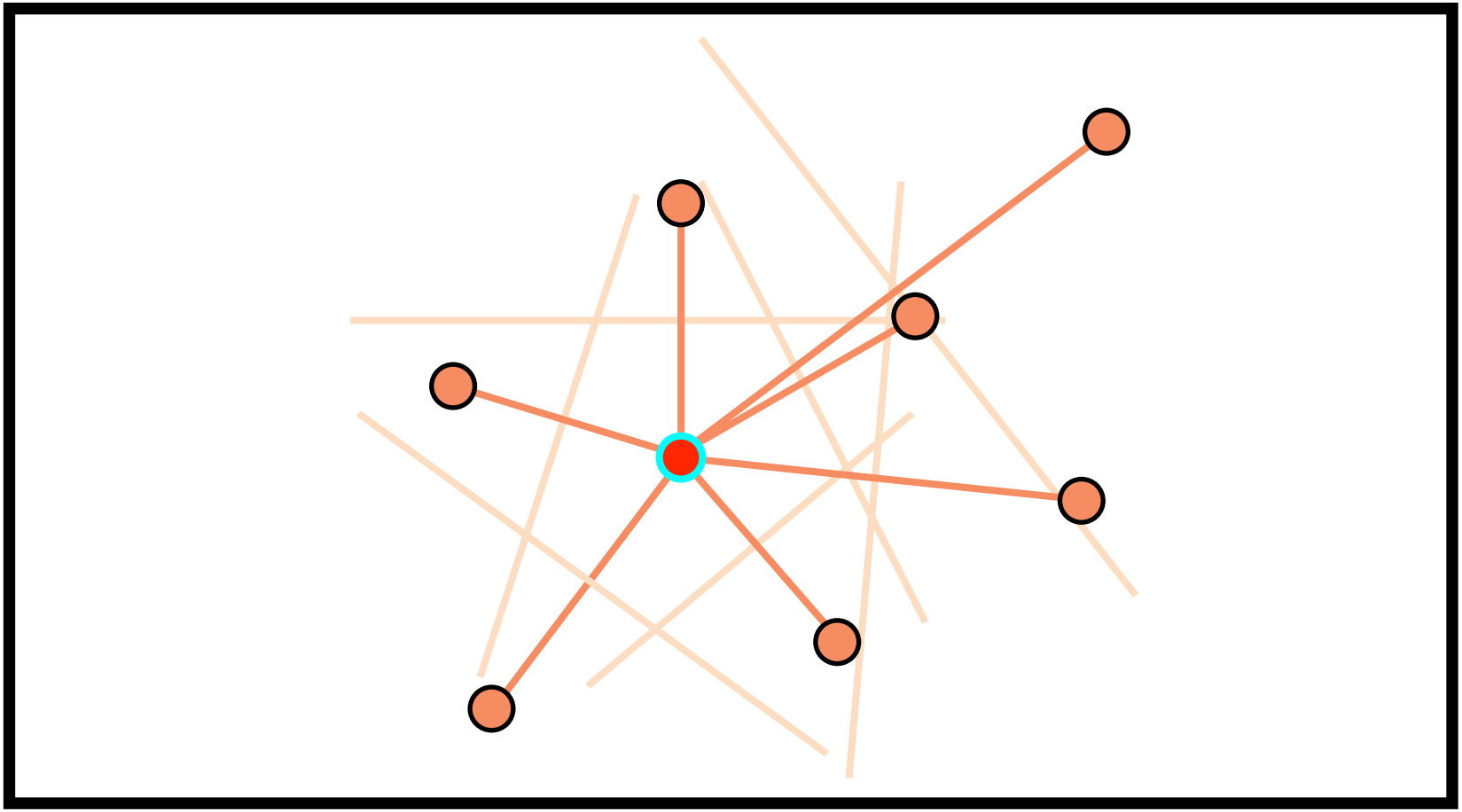
Voronoi-Gebiete



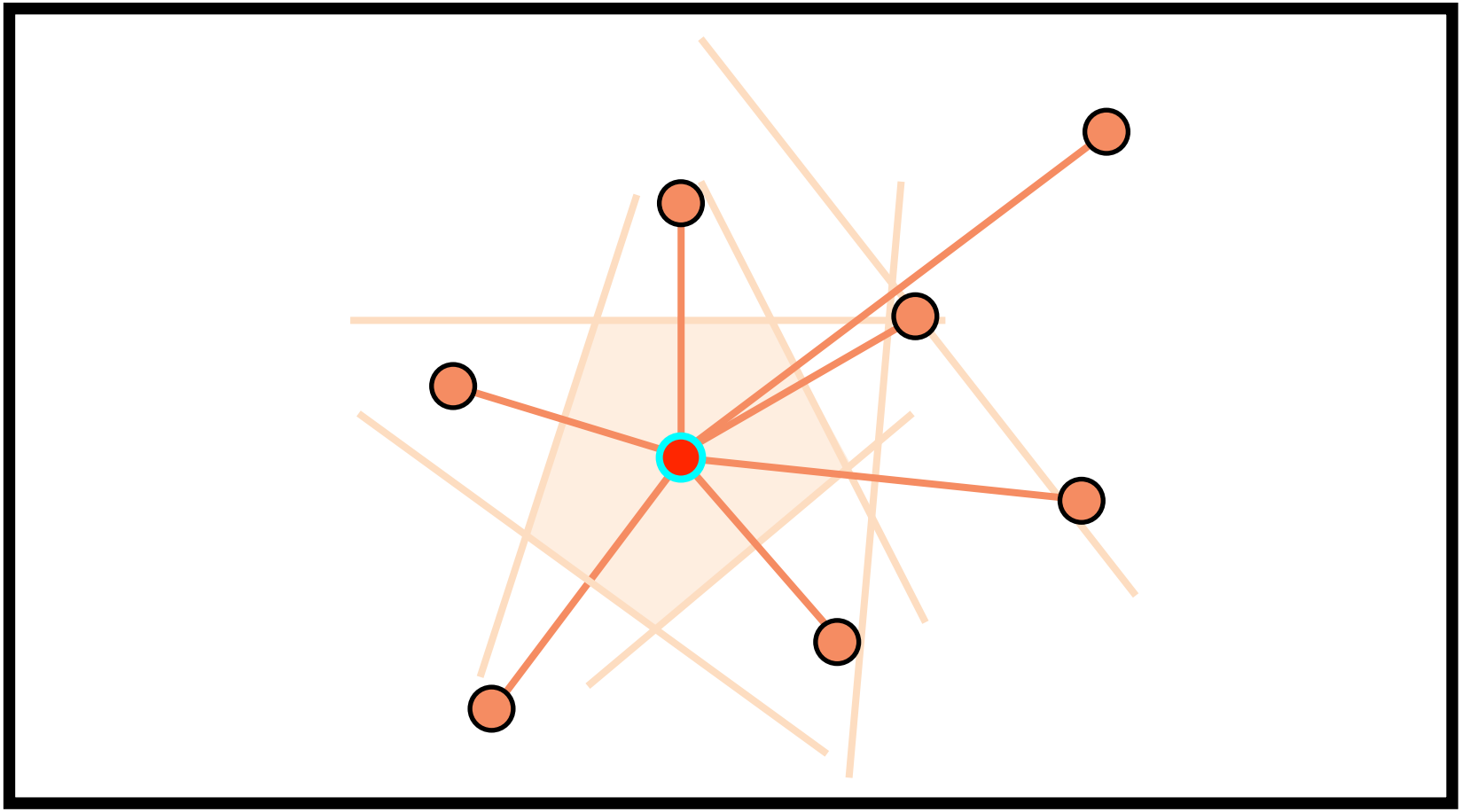
Voronoi-Gebiete



Voronoi-Gebiete



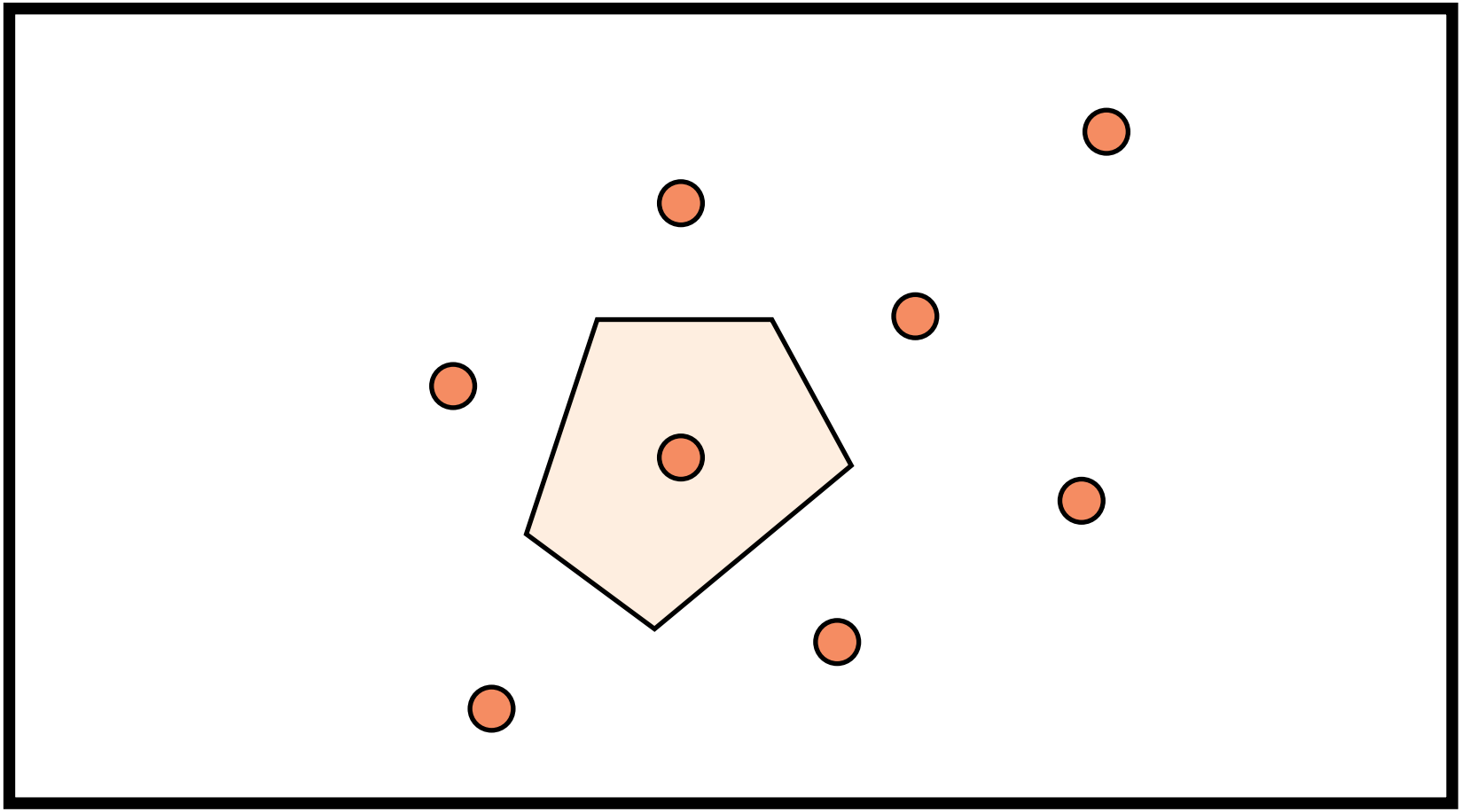
Voronoi-Gebiete



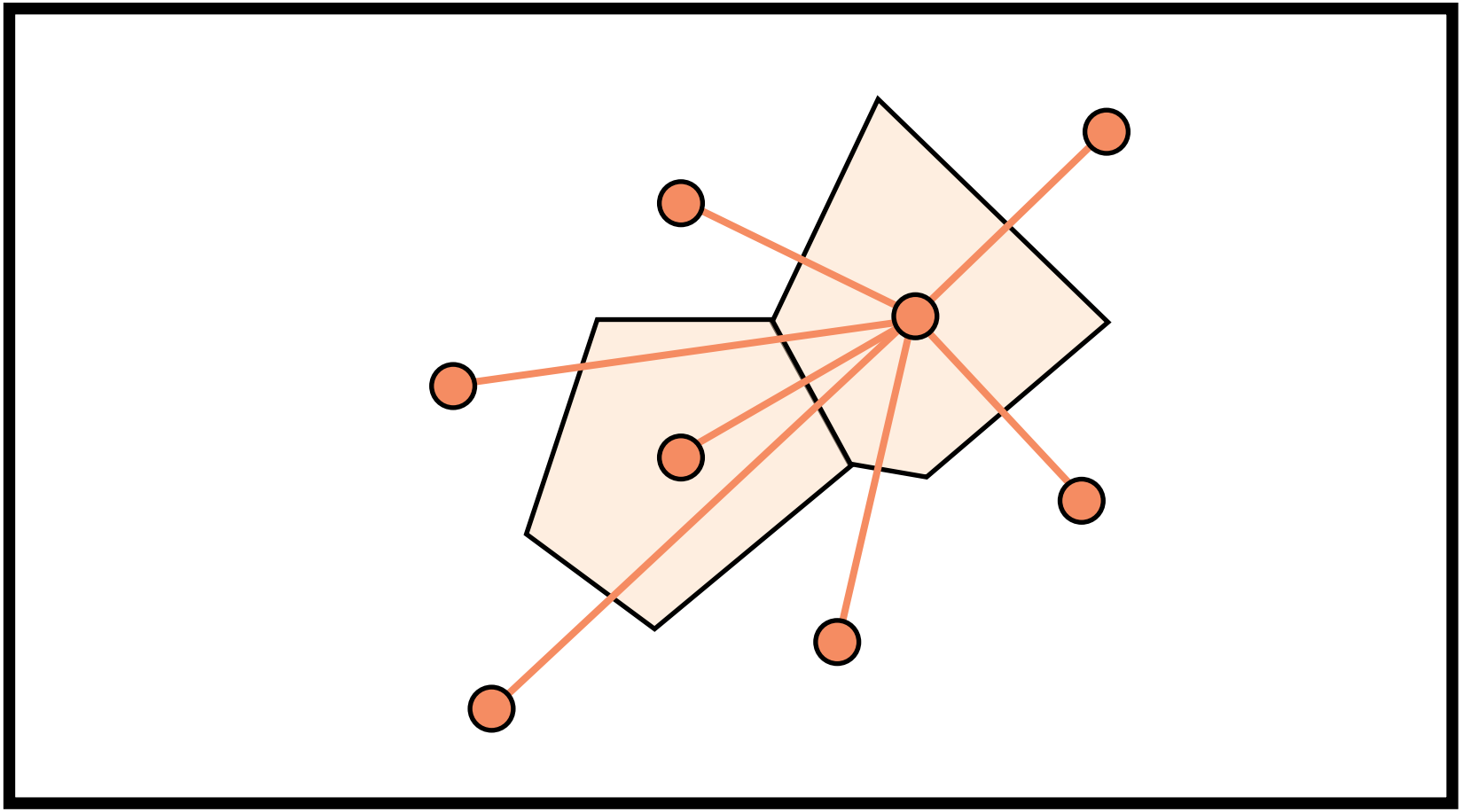
Voronoi-Gebiete

- Polygonale Gebiete
- Schnitt von Halbräumen
- Konvexe Gebiete

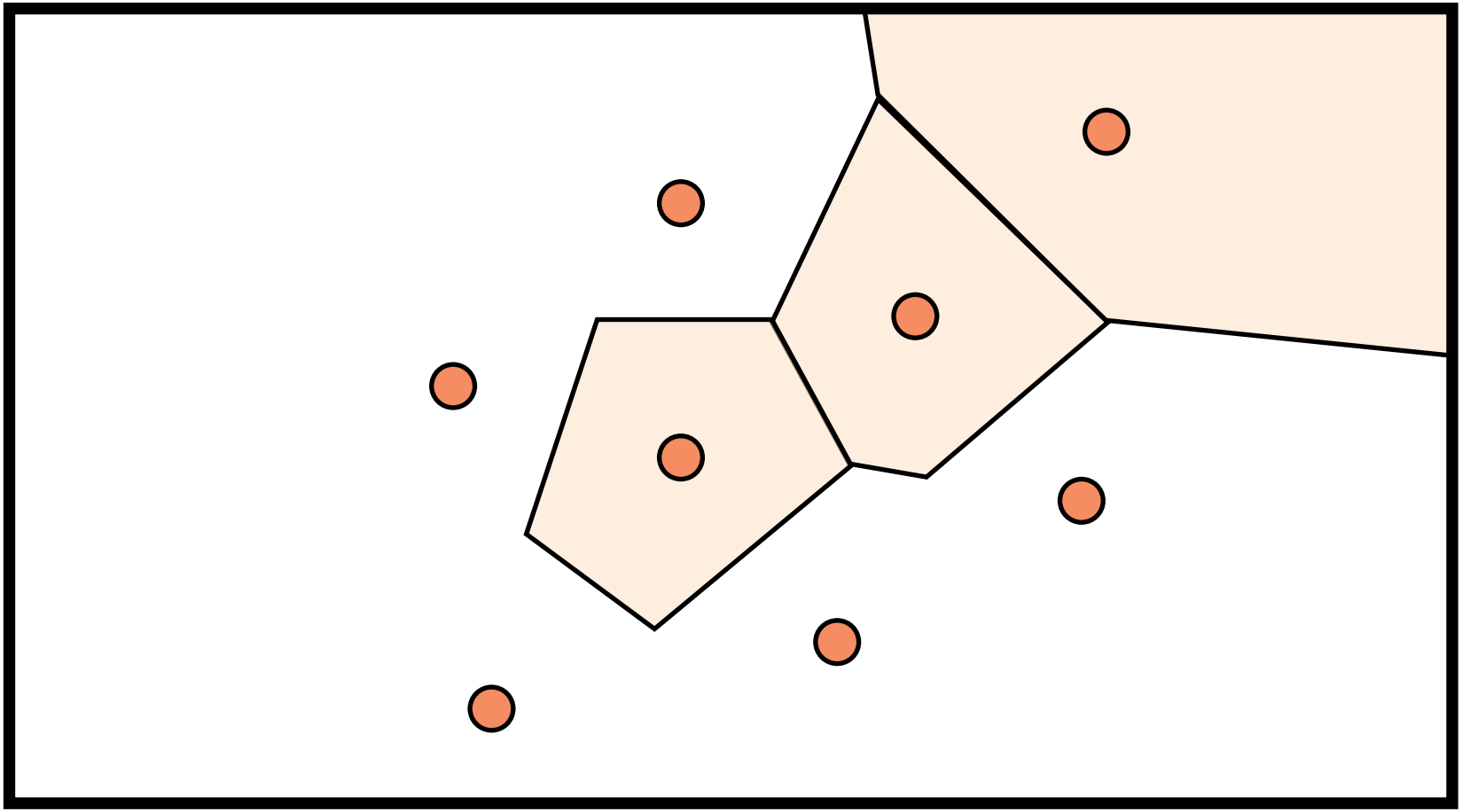
Voronoi-Diagramm



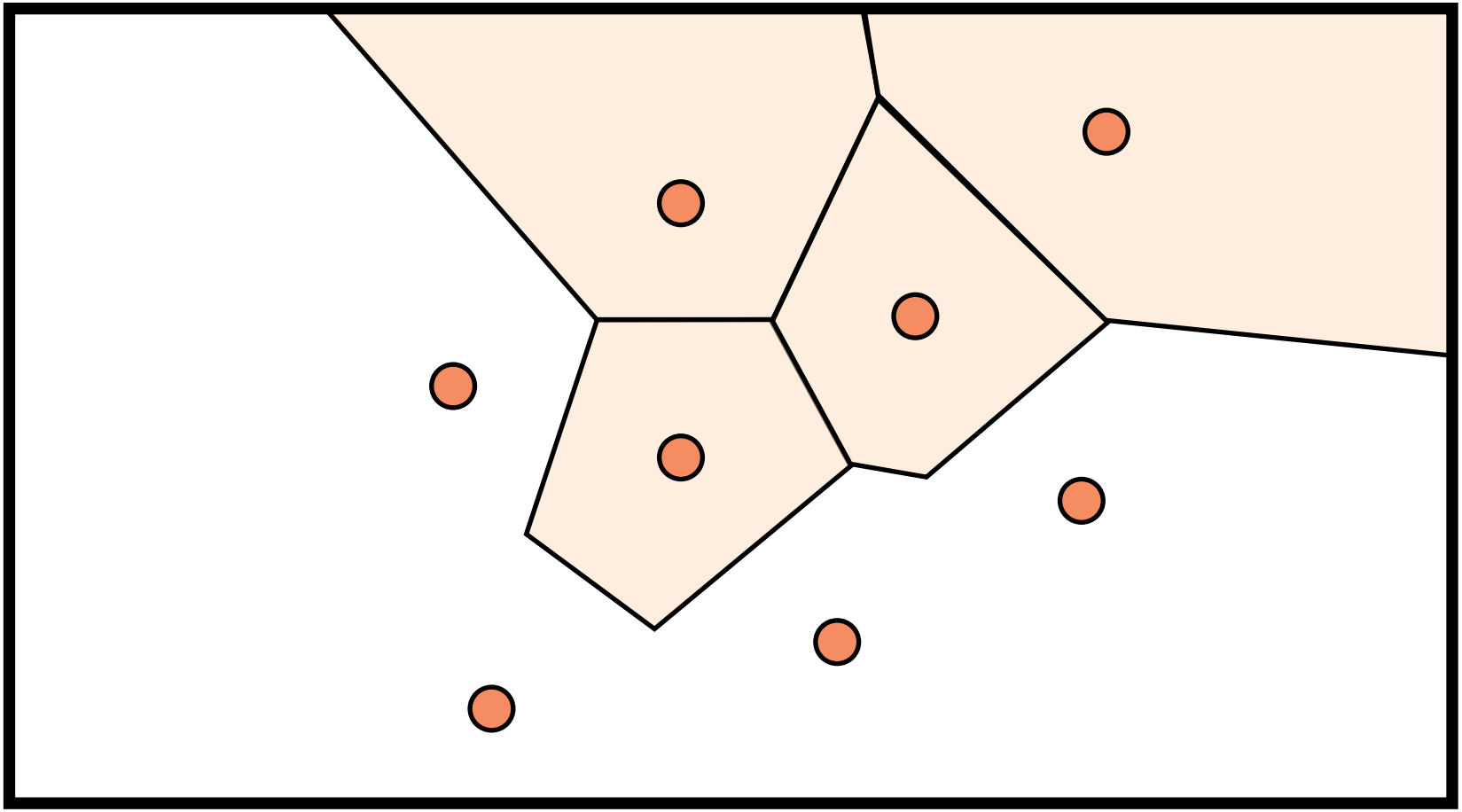
Voronoi-Diagramm



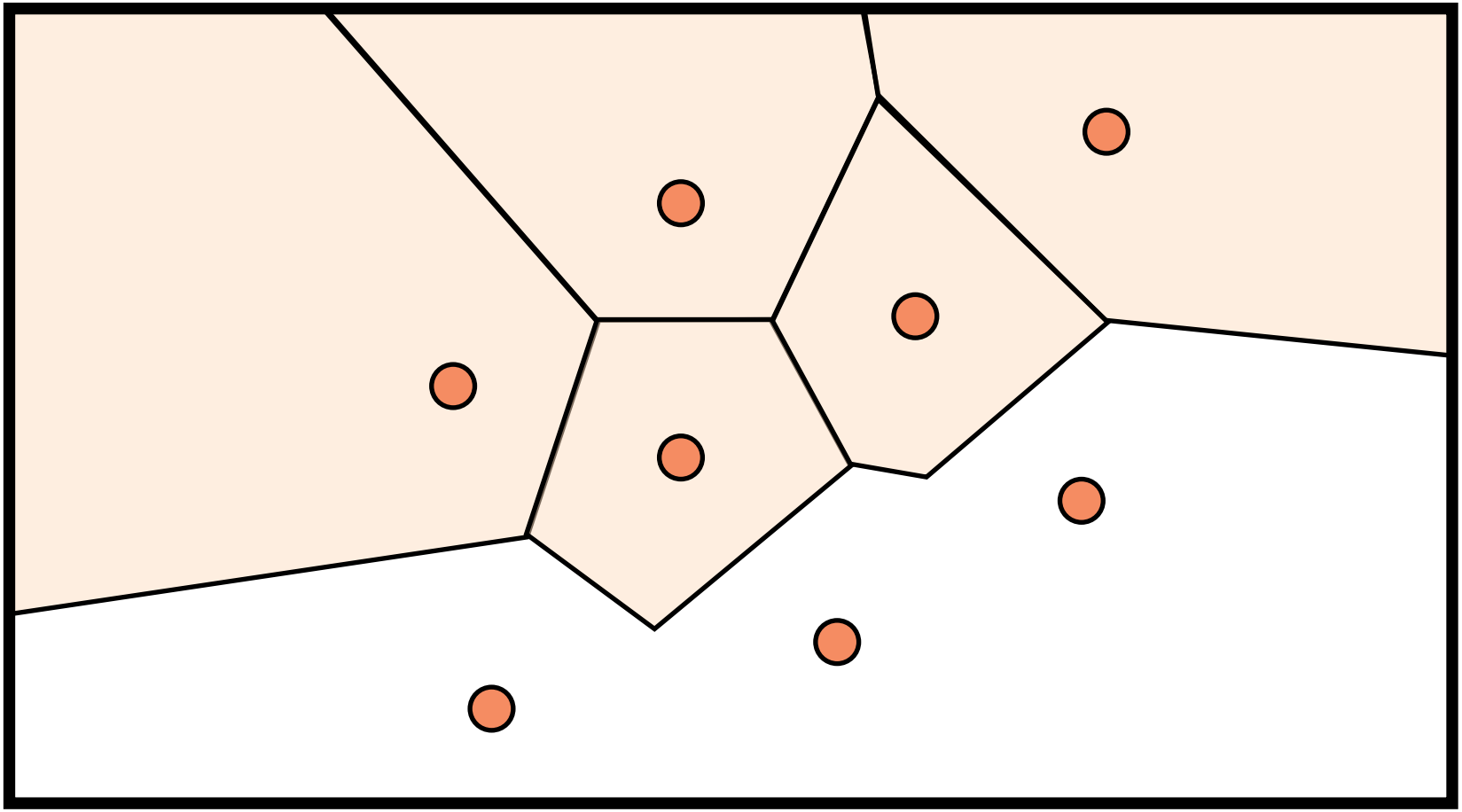
Voronoi-Diagramm



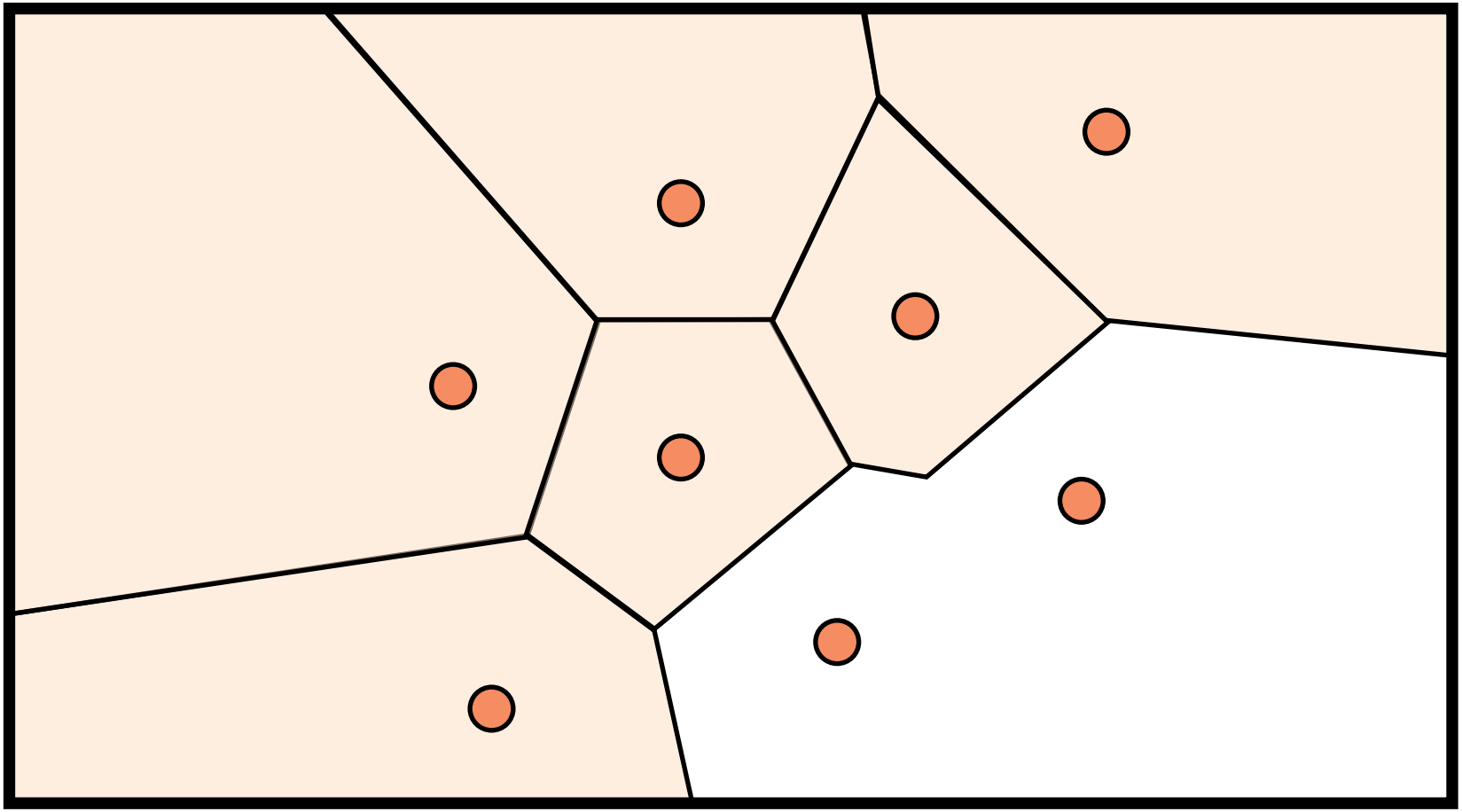
Voronoi-Diagramm



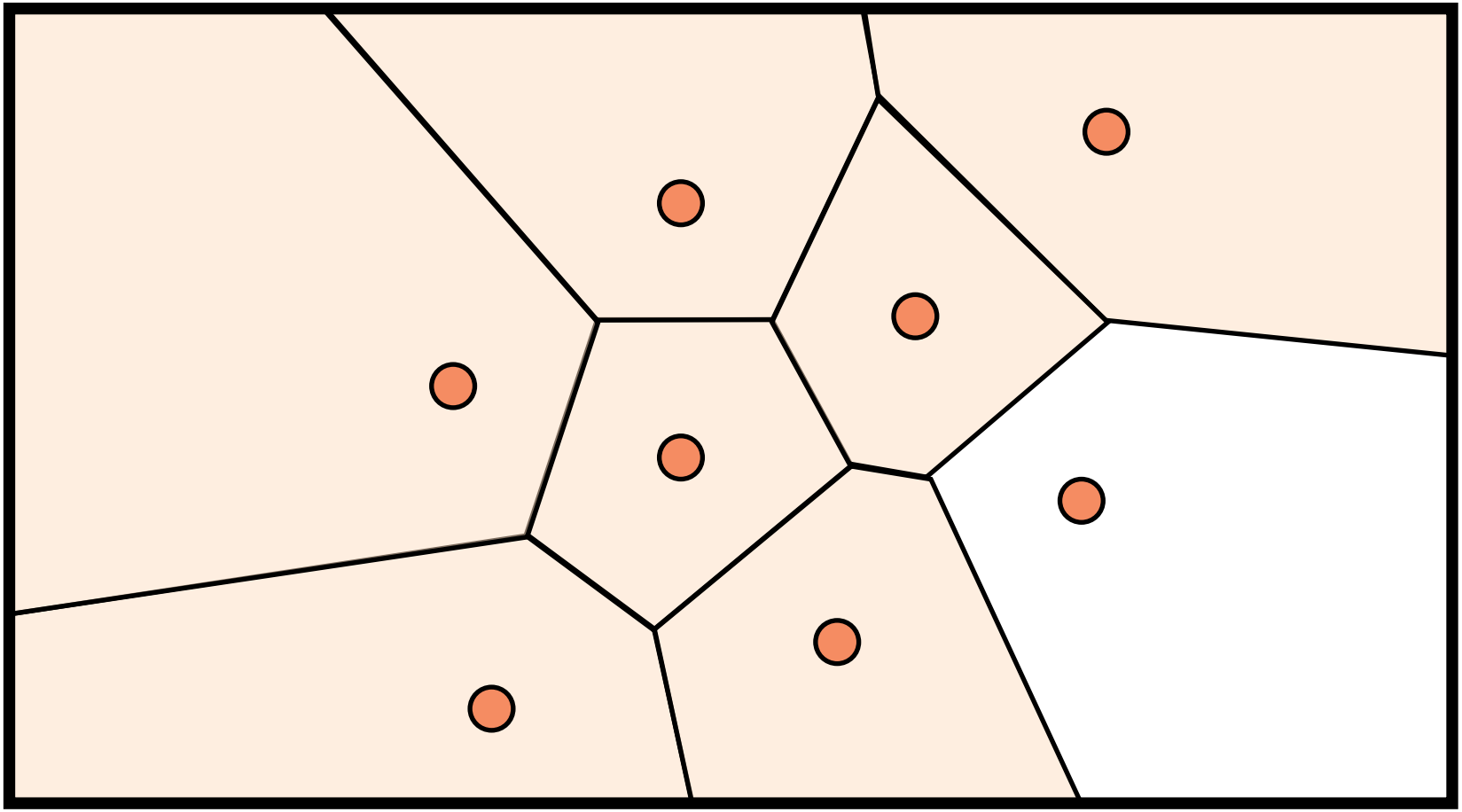
Voronoi-Diagramm



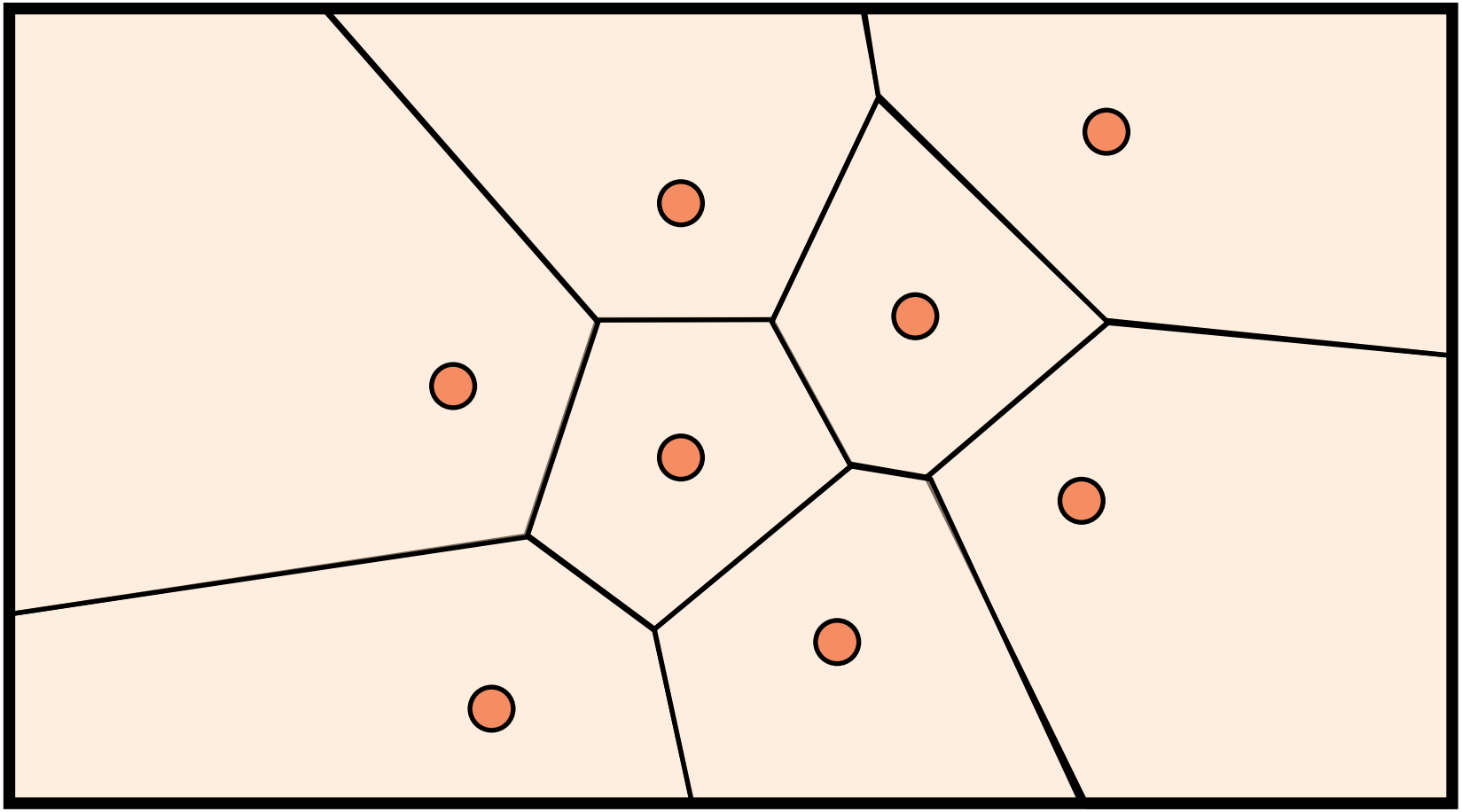
Voronoi-Diagramm



Voronoi-Diagramm



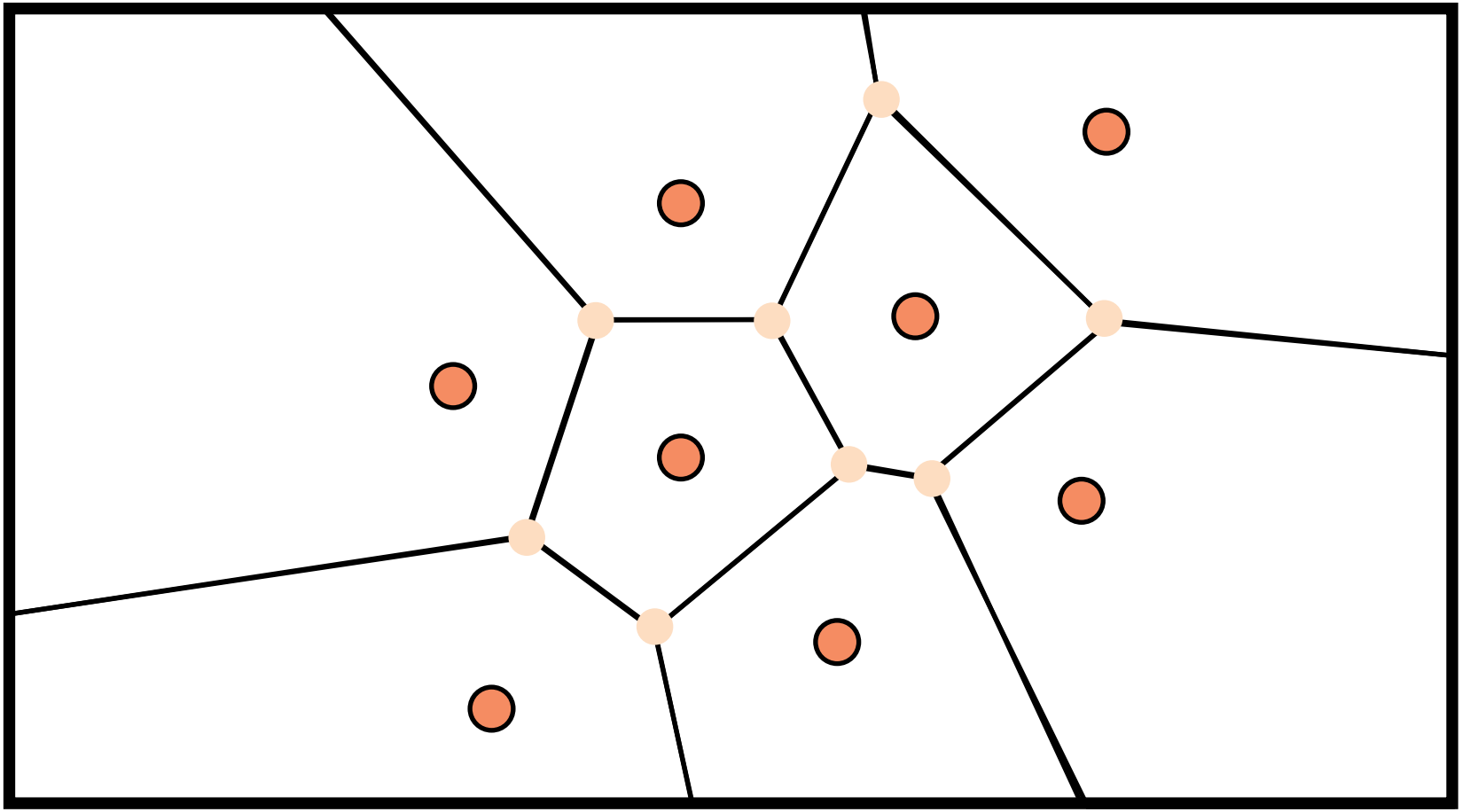
Voronoi-Diagramm



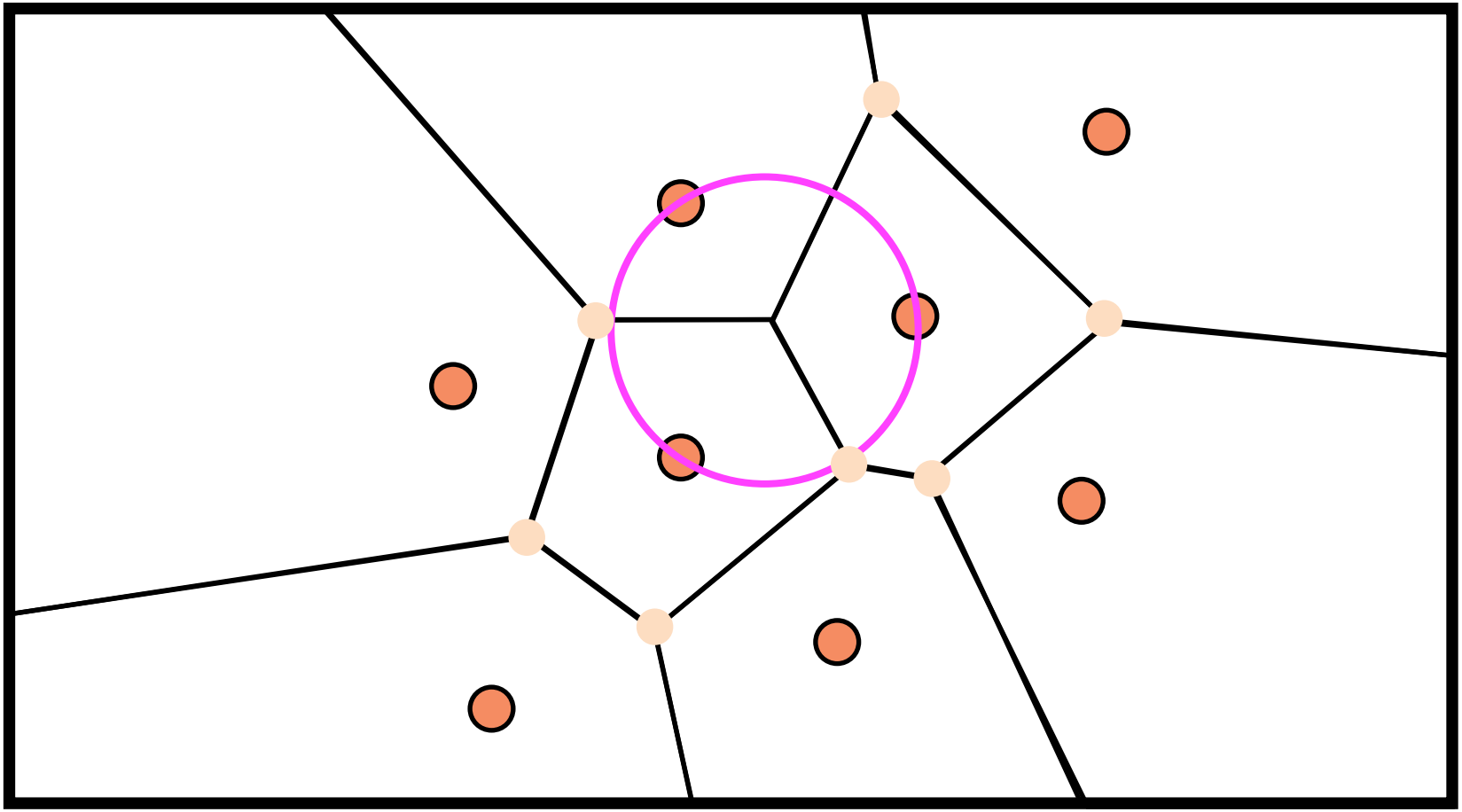
Voronoi-Diagramm

- Ein VD ist eine (planare) Graph-Struktur, die den R^2 in konvexe Regionen unterteilt
- Die Voronoi-Kanten liegen auf den Mittelsenkrechten der Verbindungsstrecken
- Im allg. Treffen an einem Voronoi-Vertex genau 3 Kanten zusammen
- Voronoi-Vertices sind Umkreismittelpunkte

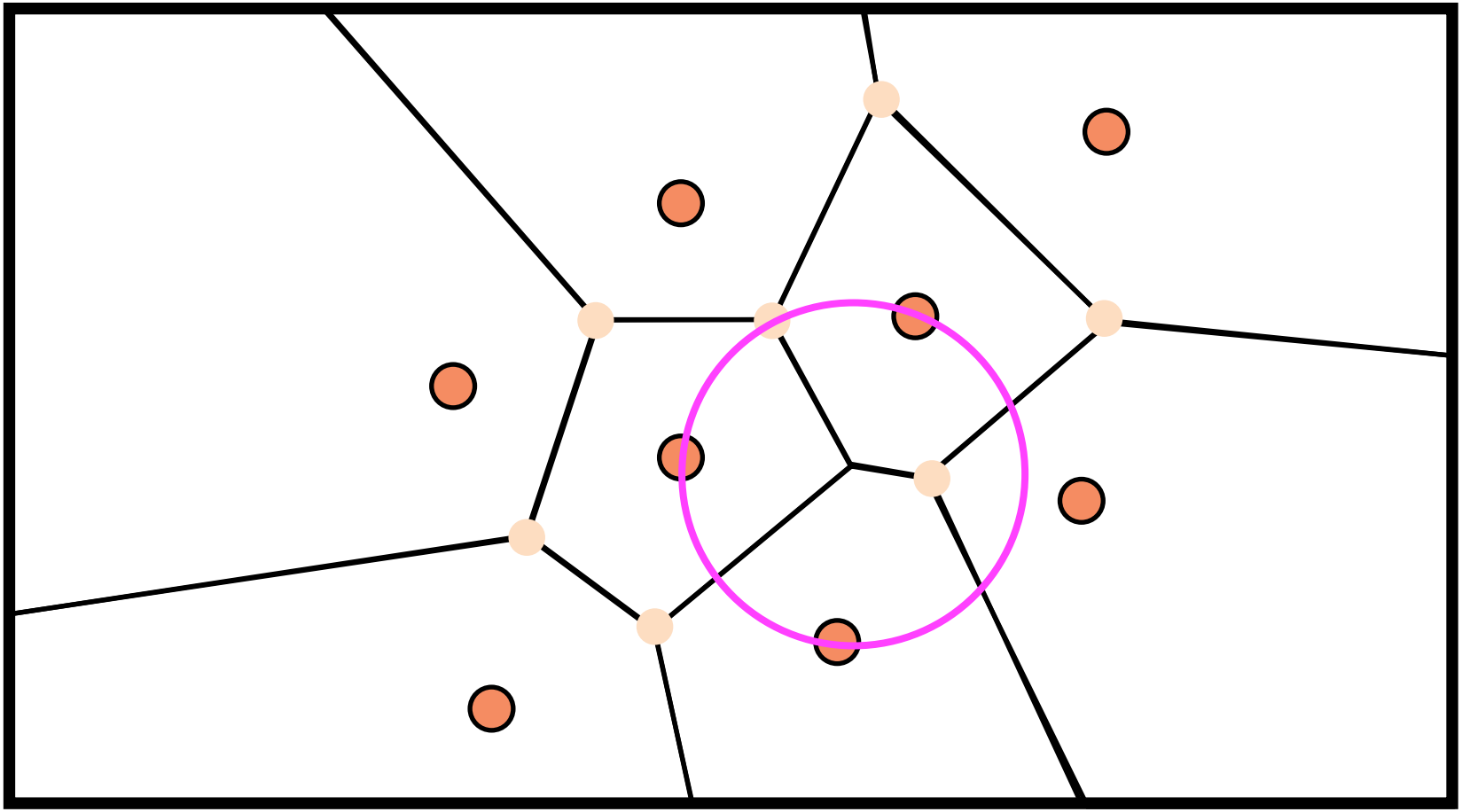
Voronoi-Diagramm



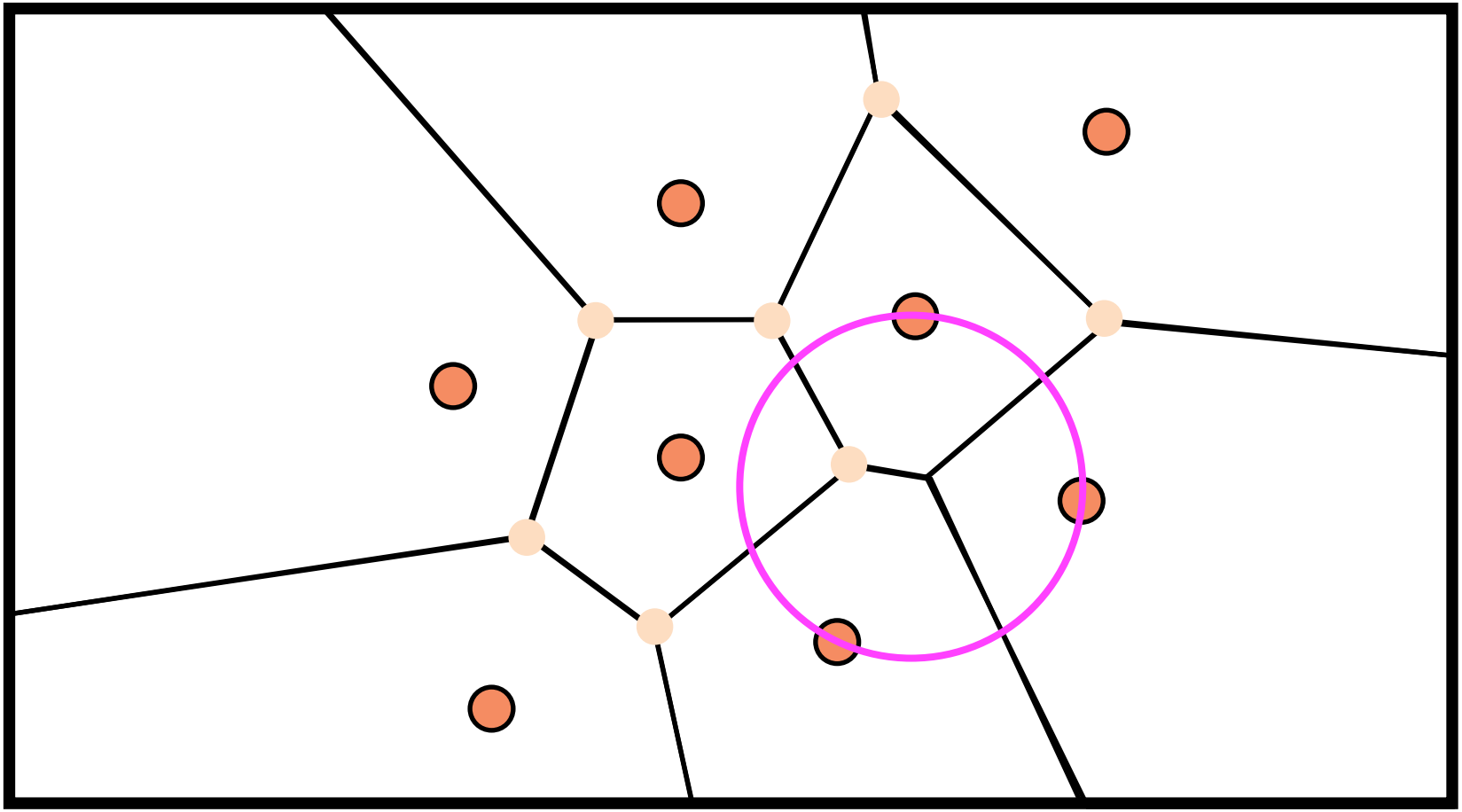
Voronoi-Diagramm



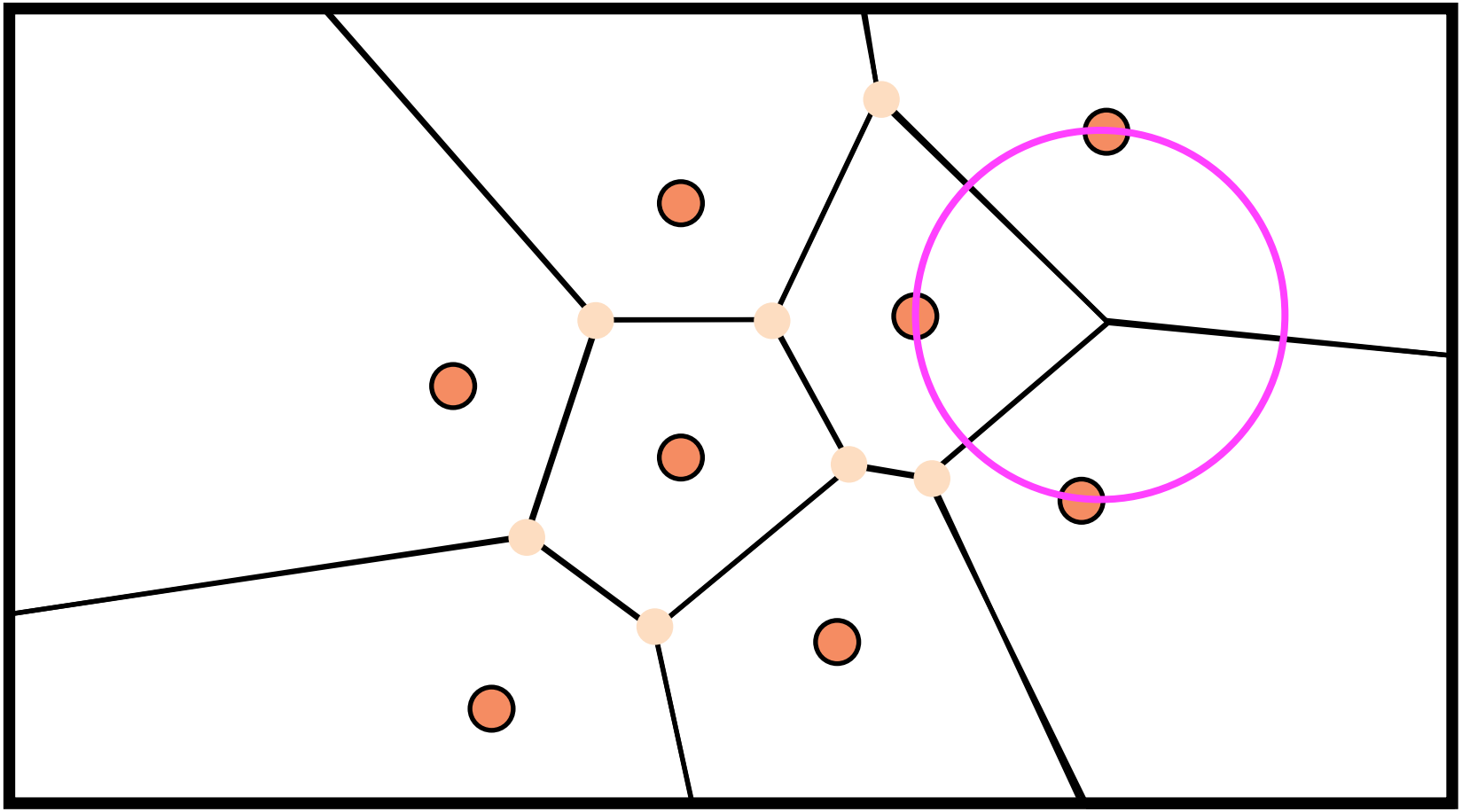
Voronoi-Diagramm



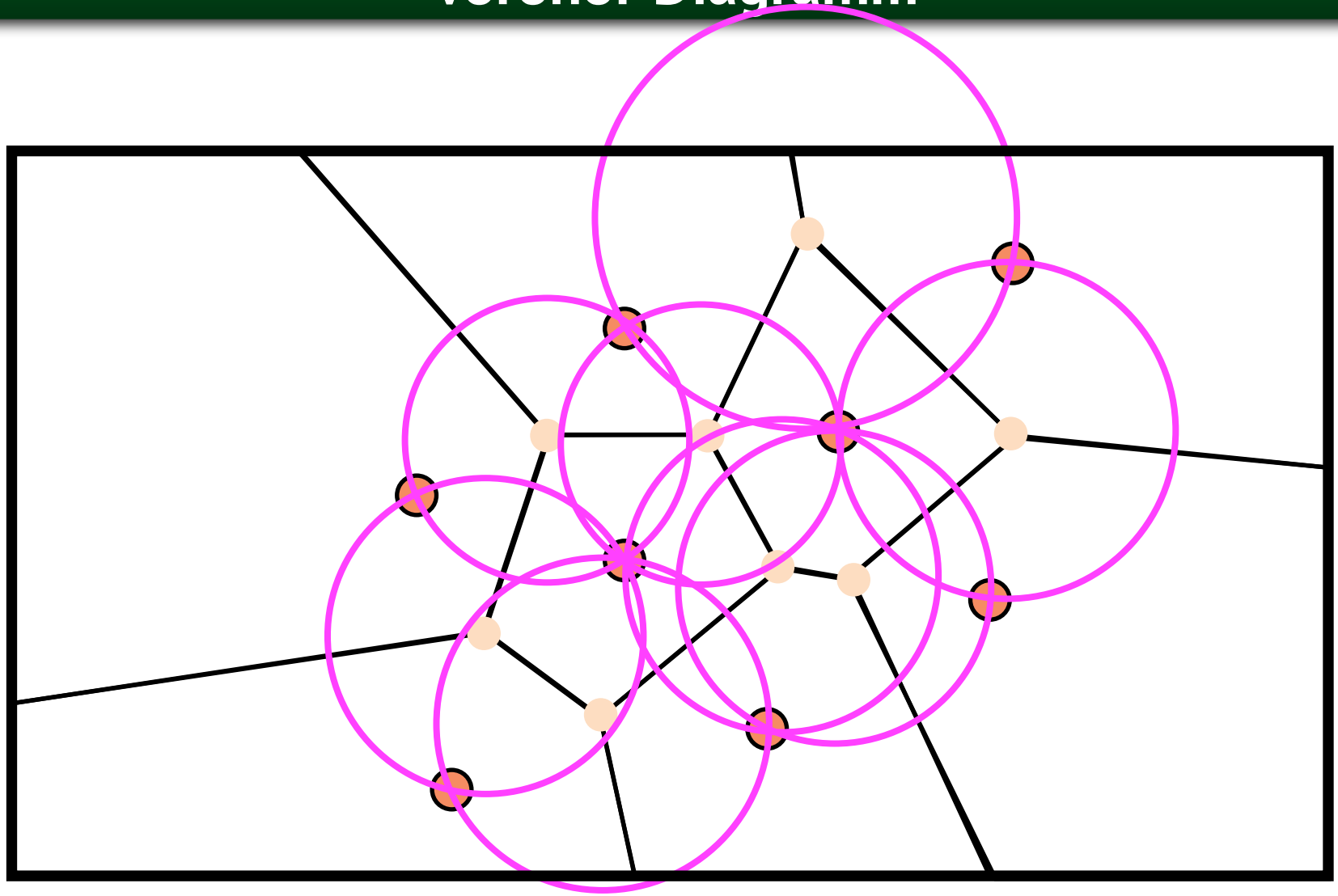
Voronoi-Diagramm



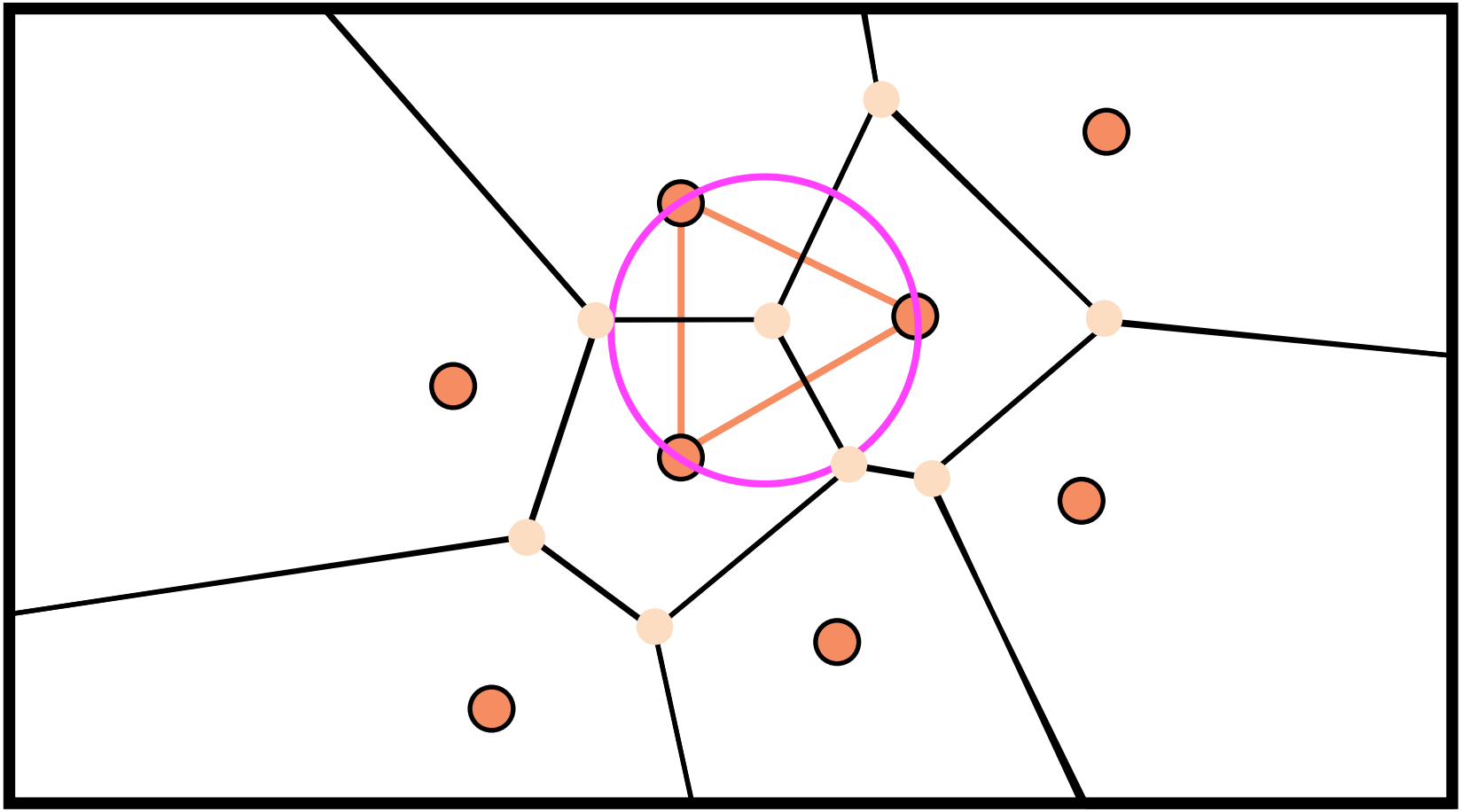
Voronoi-Diagramm



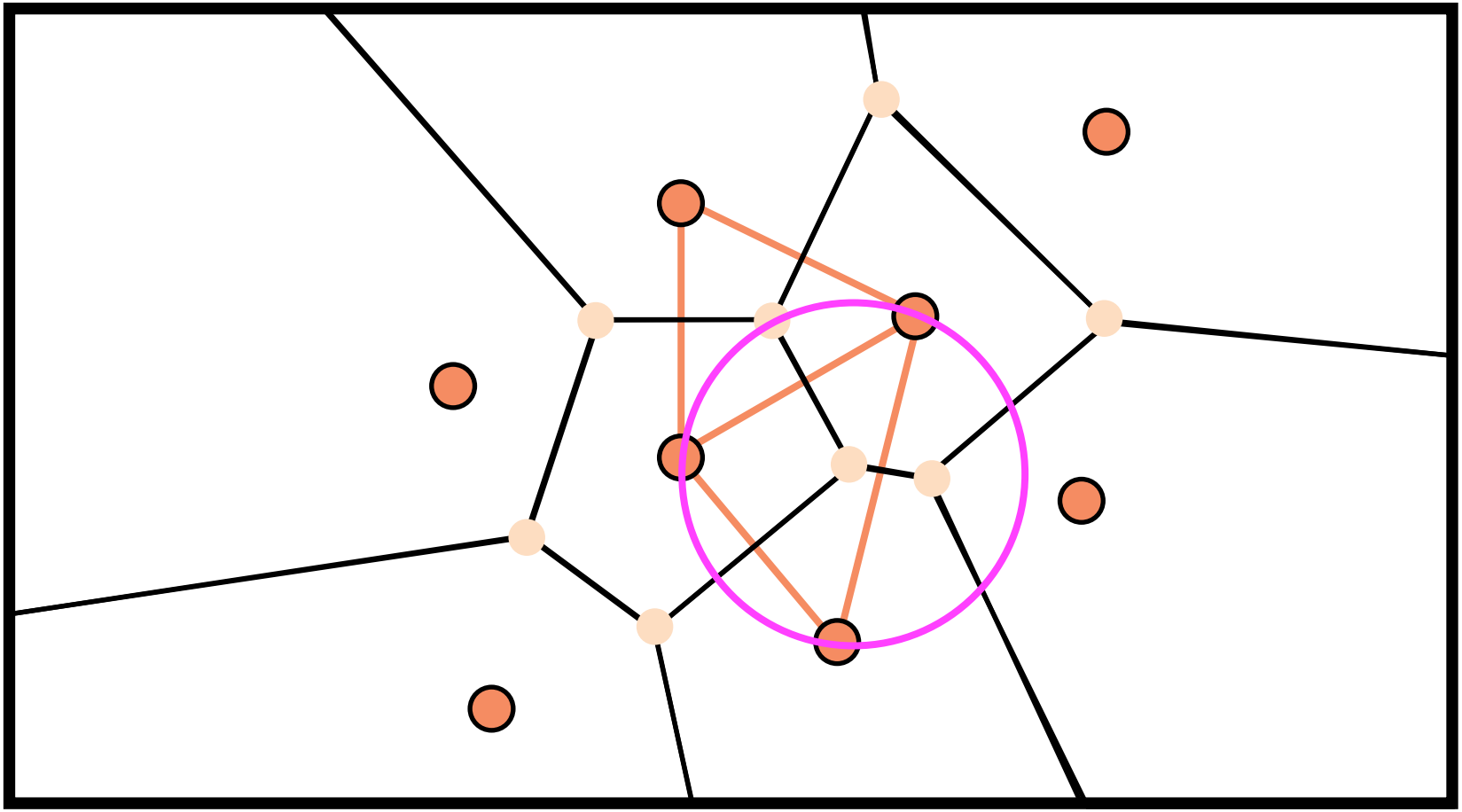
Voronoi-Diagramm



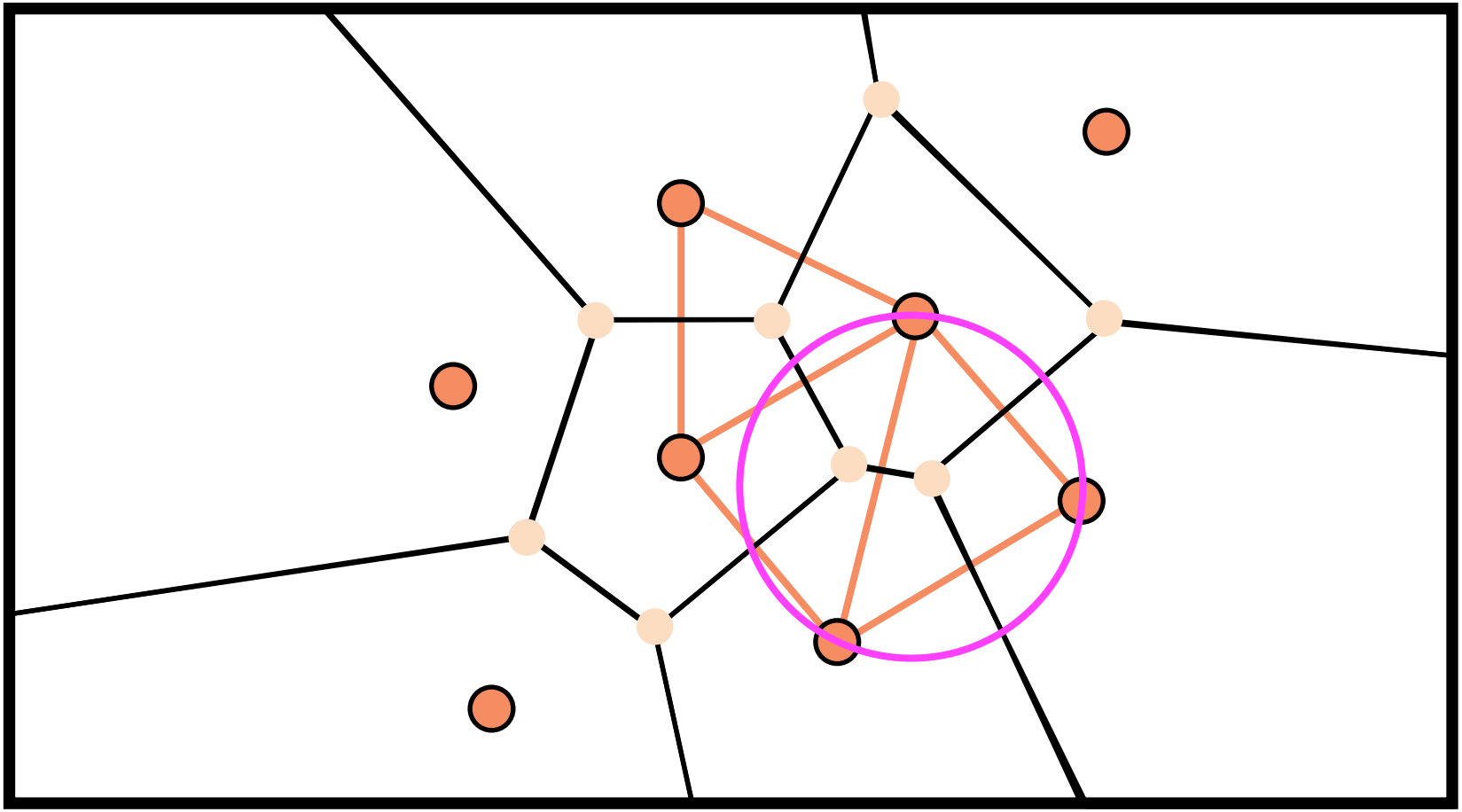
Voronoi-Diagramm



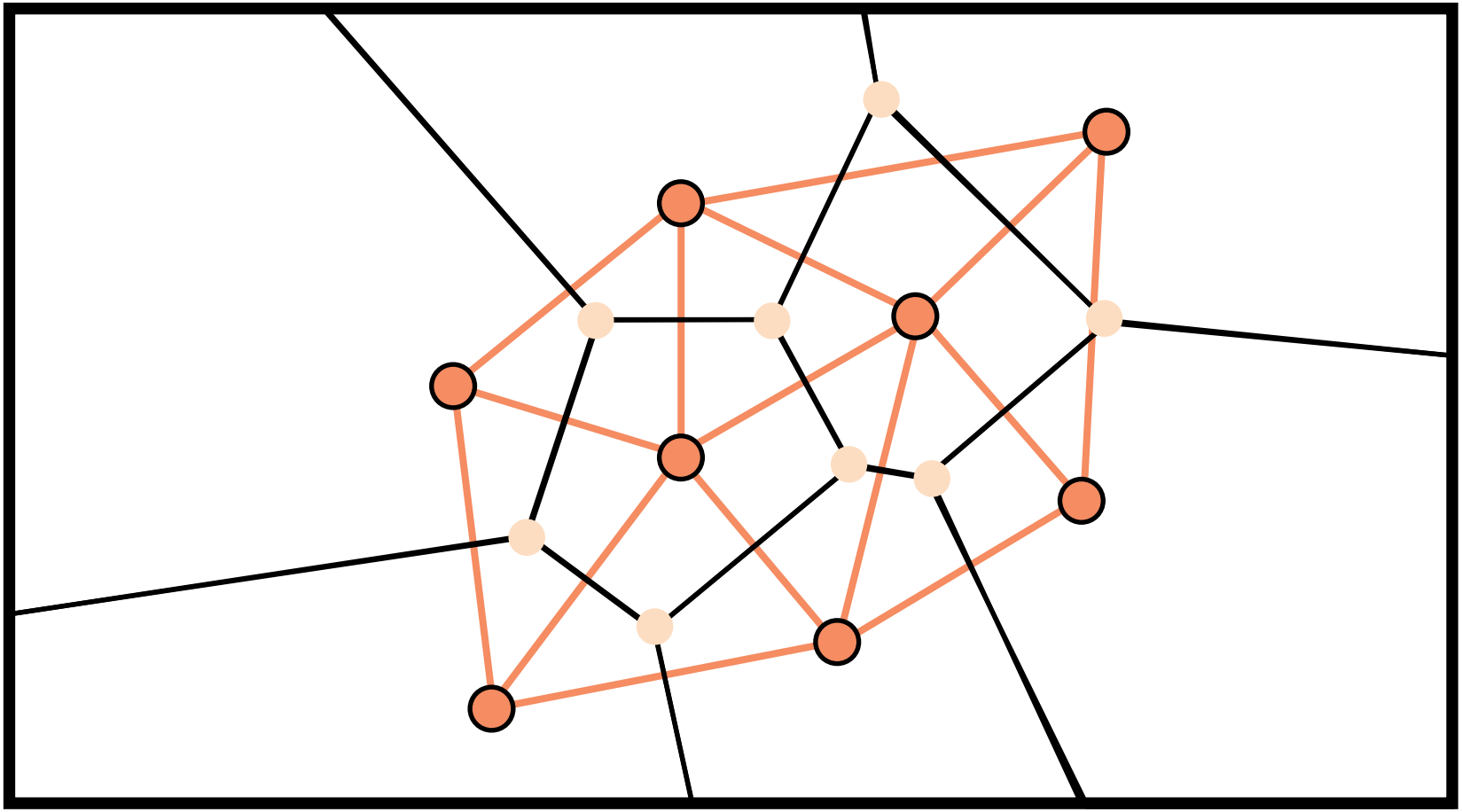
Voronoi-Diagramm



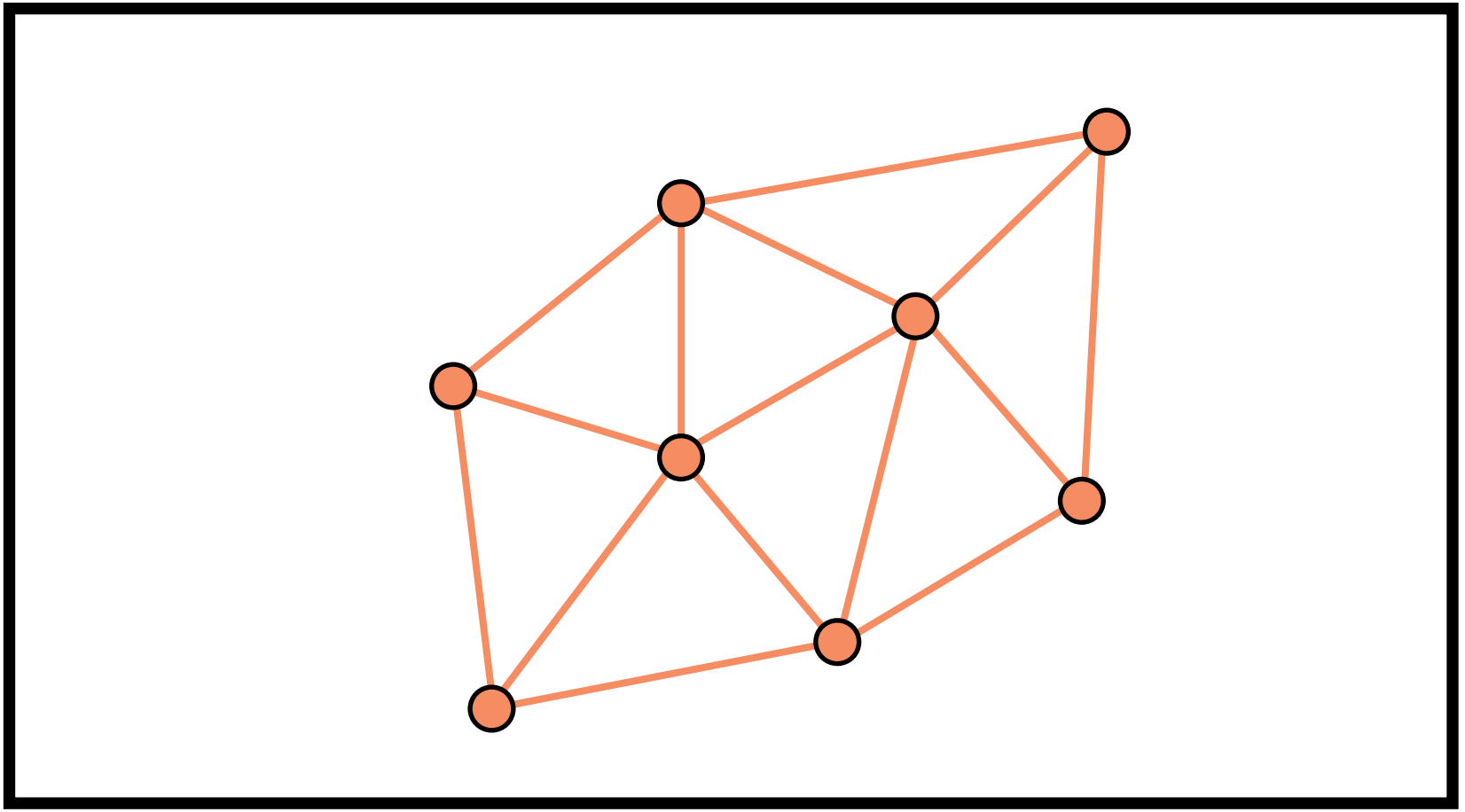
Voronoi-Diagramm



Voronoi-Diagramm



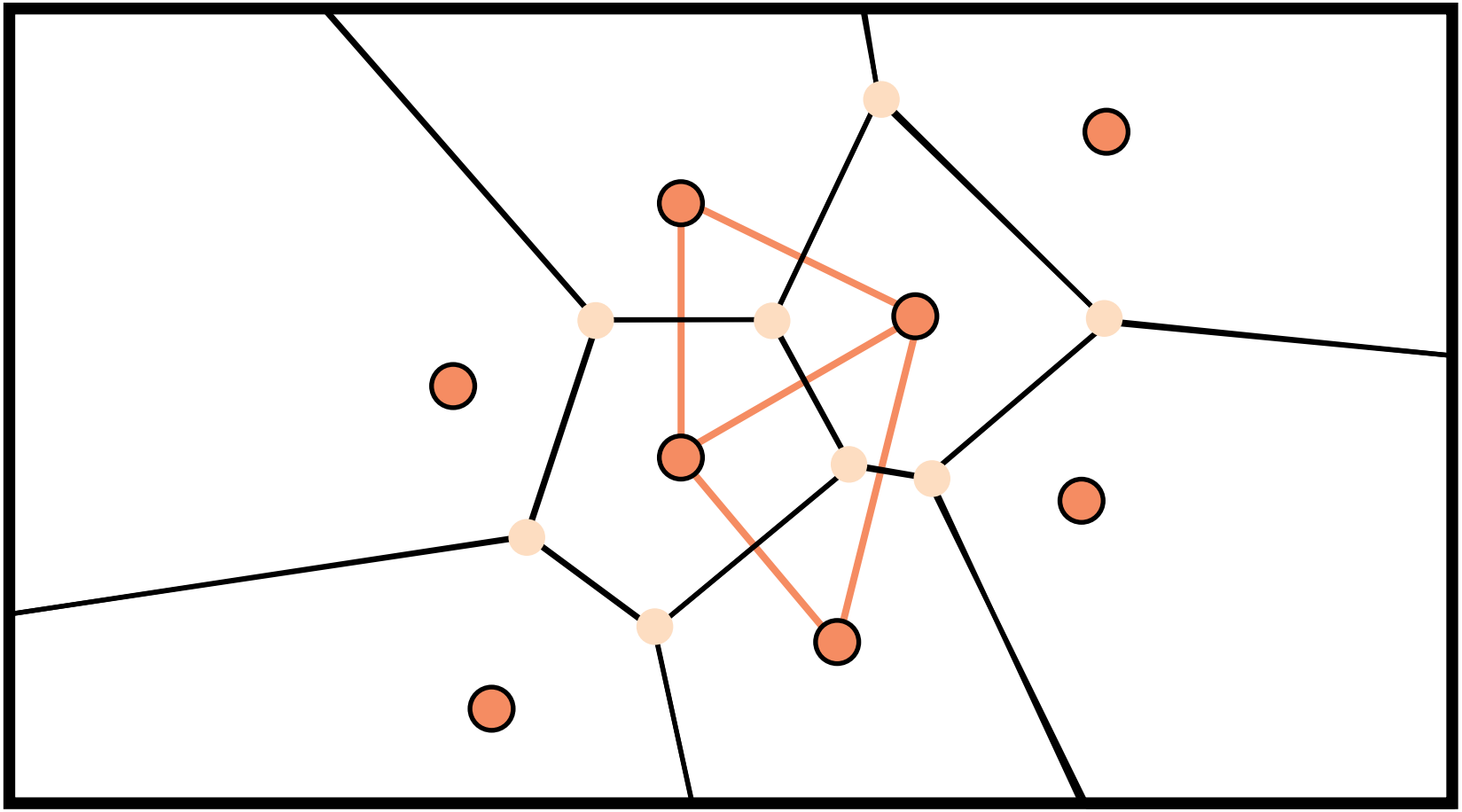
Delaunay-Triangulierung



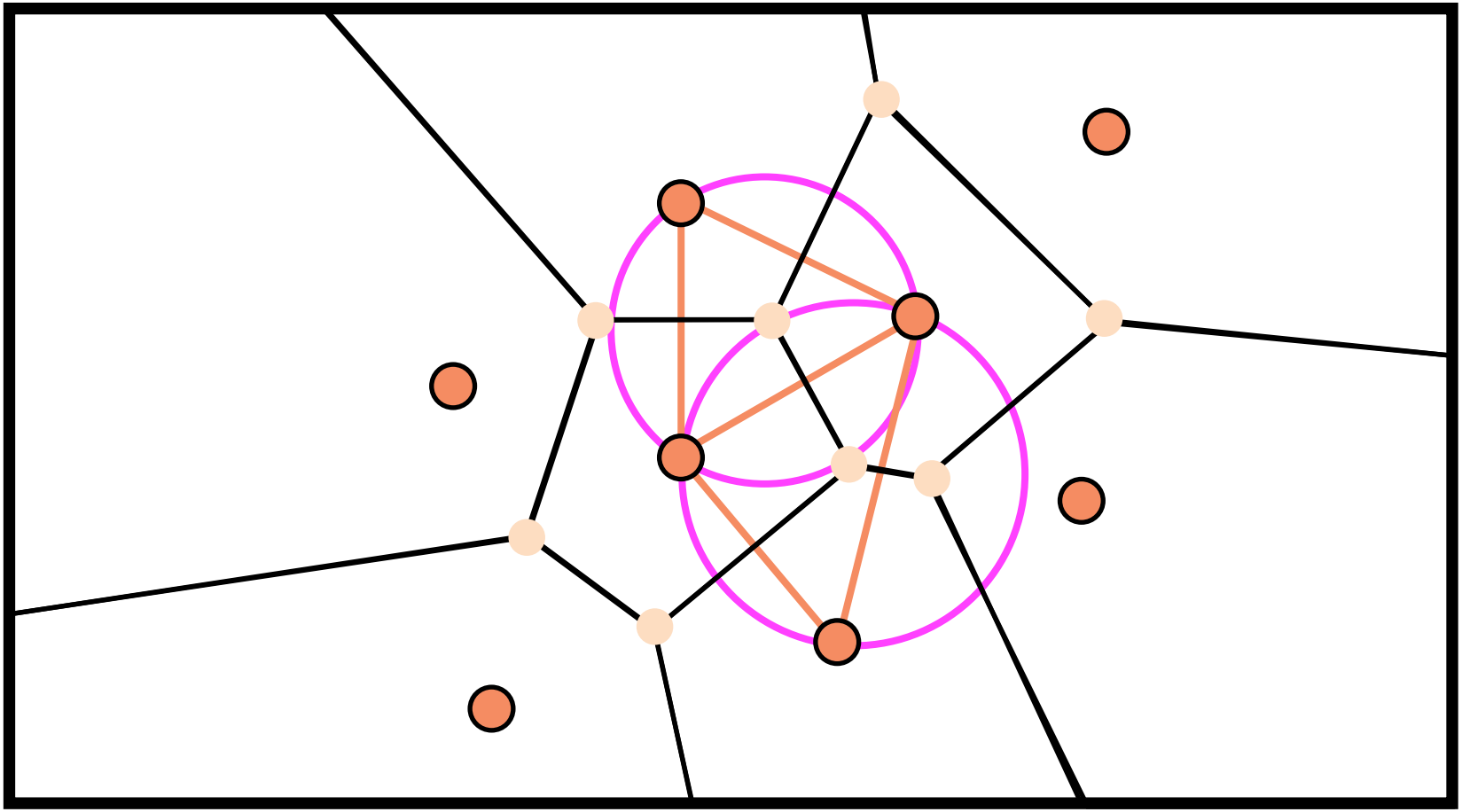
Delaunay-Triangulierung

- Eigenschaften (1):
  - Dualer Graph zum Voronoi-Diagramm
  - Zerlegt die konvexe Hülle der Punkte in disjunkte Dreiecke (mit den p_i als Ecken)
  - Die Region innerhalb eines Dreiecks ist einem der drei Eckpunkte am nächsten
  - Der Umkreis jedes Dreiecks enthält keinen weiteren Punkt p_j

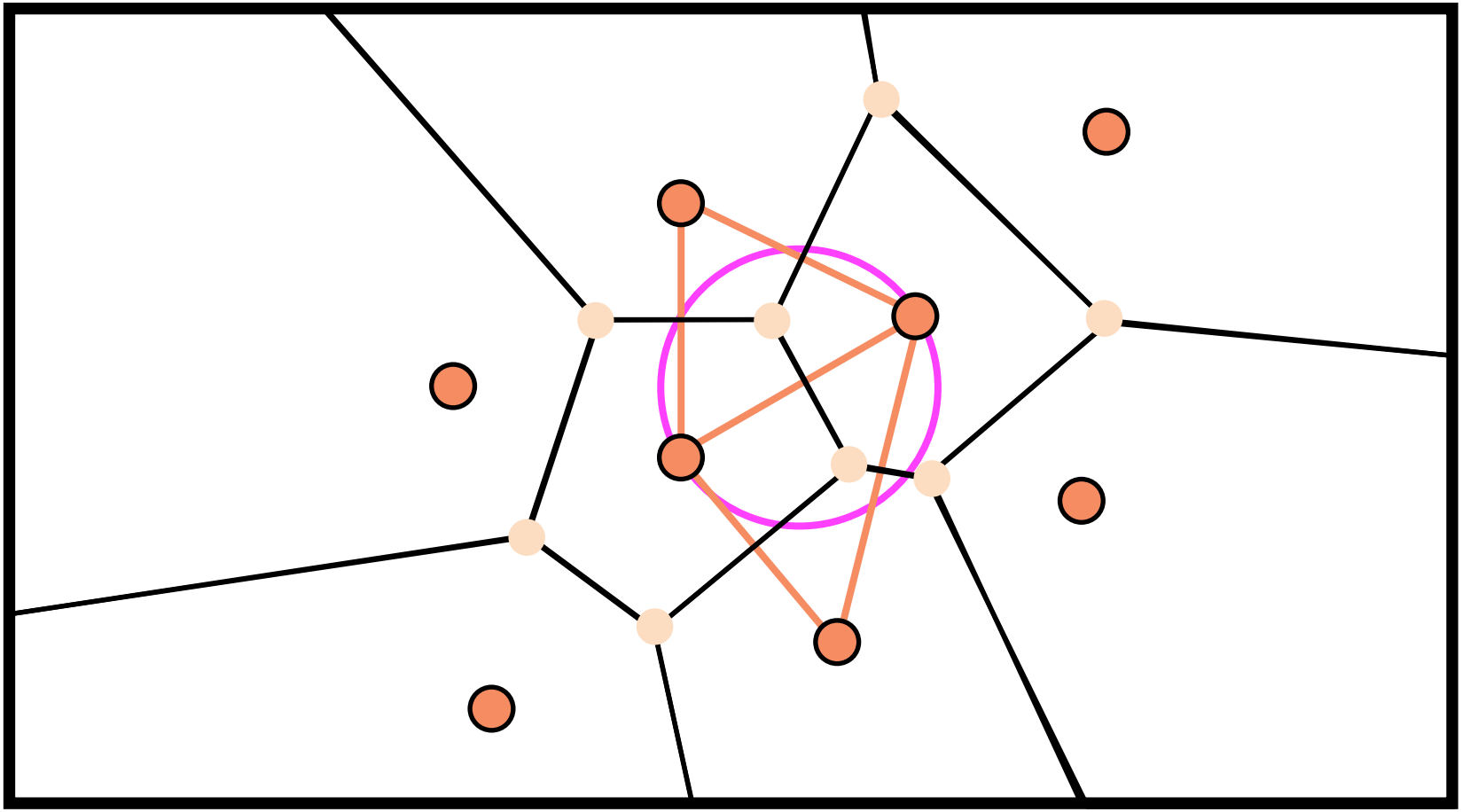
Voronoi-Diagramm



Voronoi-Diagramm



Voronoi-Diagramm



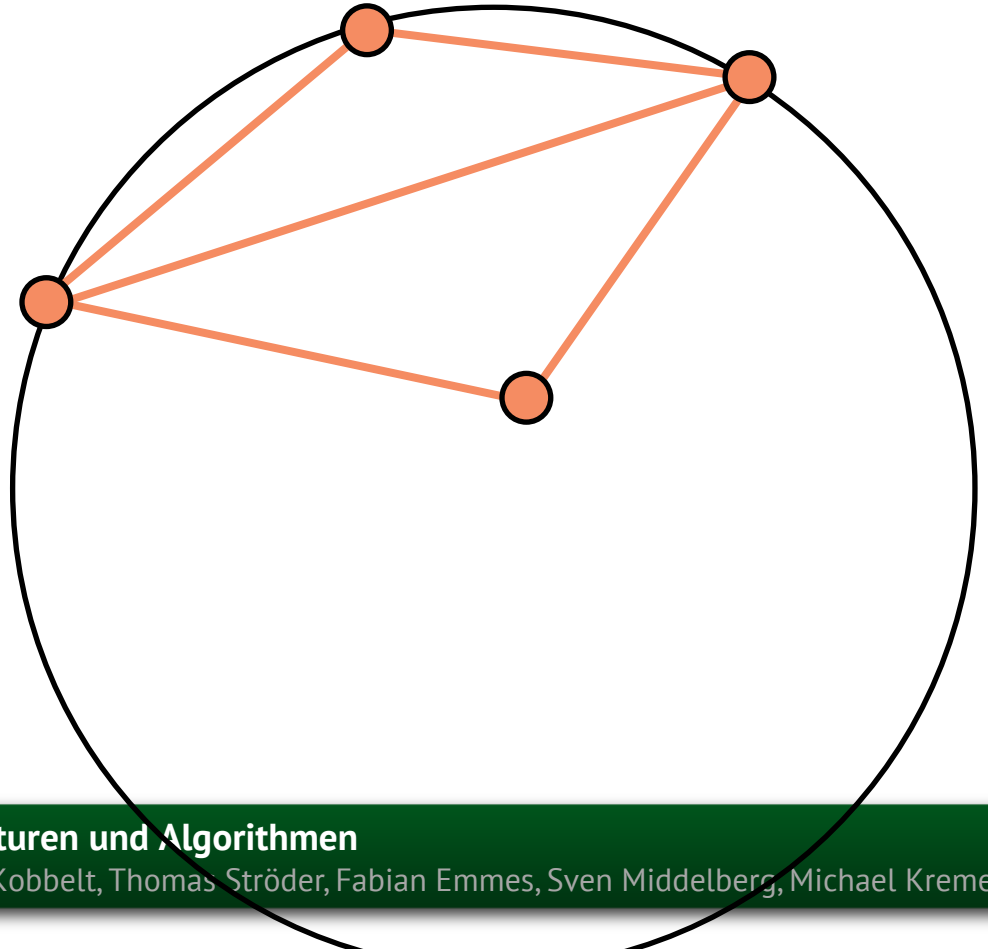
Delaunay-Triangulierung

- Eigenschaften(2):
  - Der Umkreis jeder Kante enthält keinen weiteren Punkt p_j
  - Die DT ist eindeutig definiert
  - Maximaler minimaler Innenwinkel
  - Einfacher zu handhaben als VD, da keine allg. Graph-Struktur verarbeitet werden muß

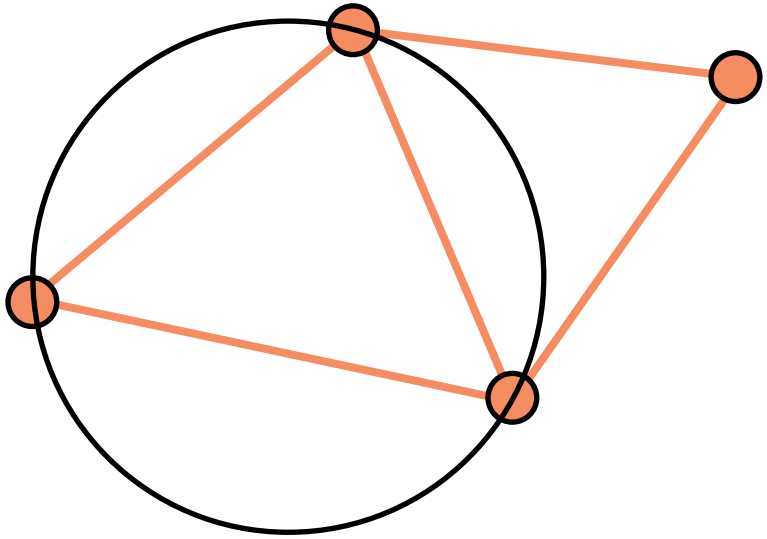
Delaunay-Triangulierung

- Inkrementeller Algorithmus zur Erzeugung …
  - Für nur drei Punkte p_1, p_2, p_3 trivial
  - Füge weitere Punkte $p_4, \ldots$ ein und stelle die Delaunay-Eigenschaften wieder her

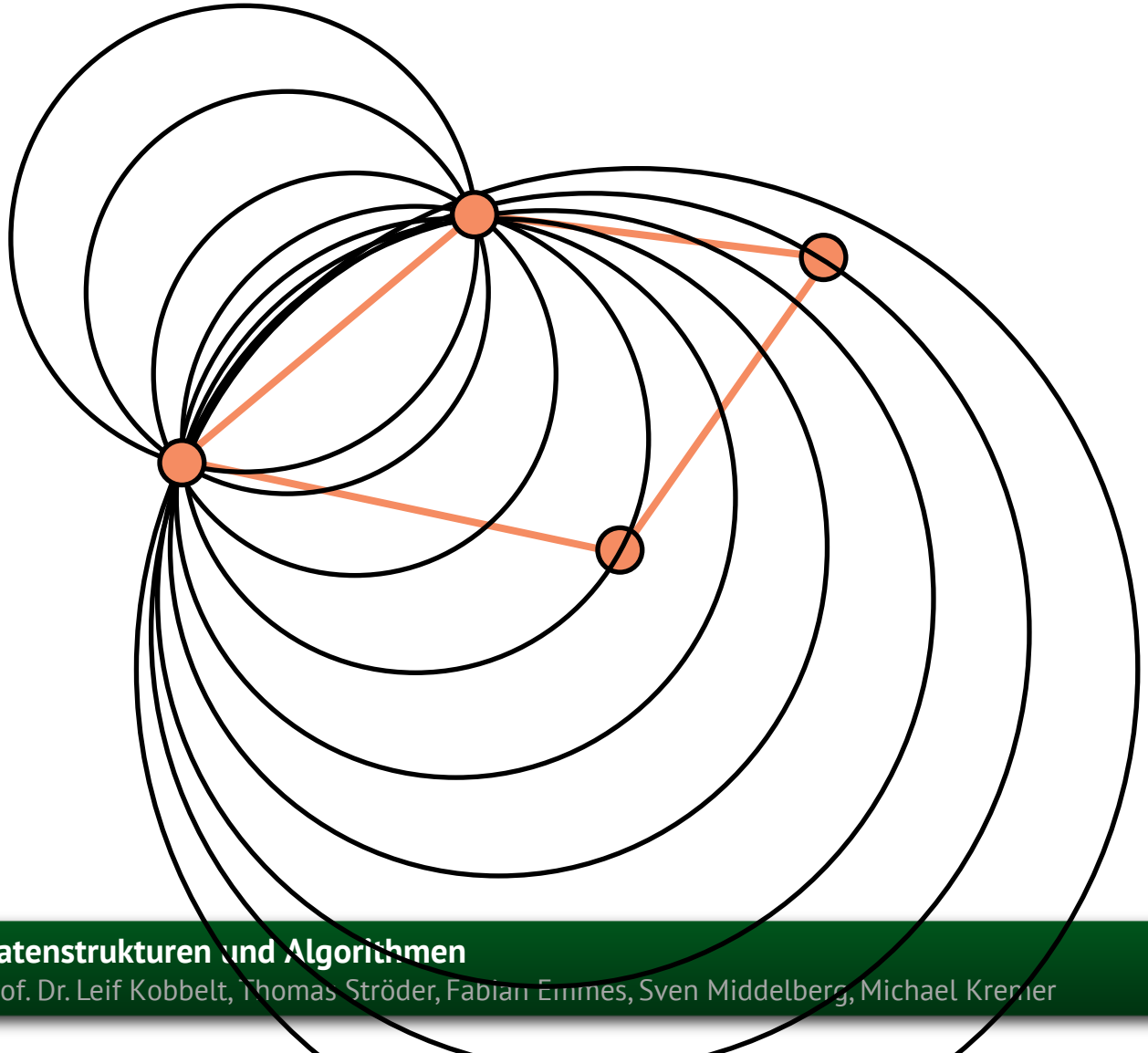
Edge-Flipping



Edge-Flipping



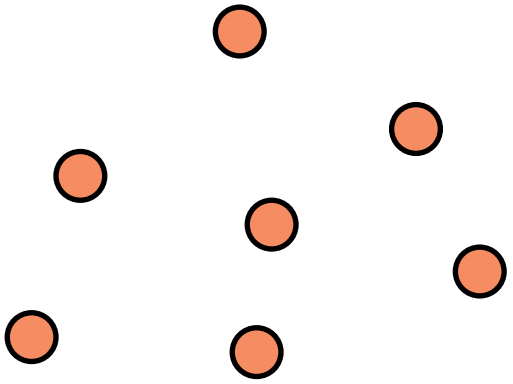
Edge-Flipping



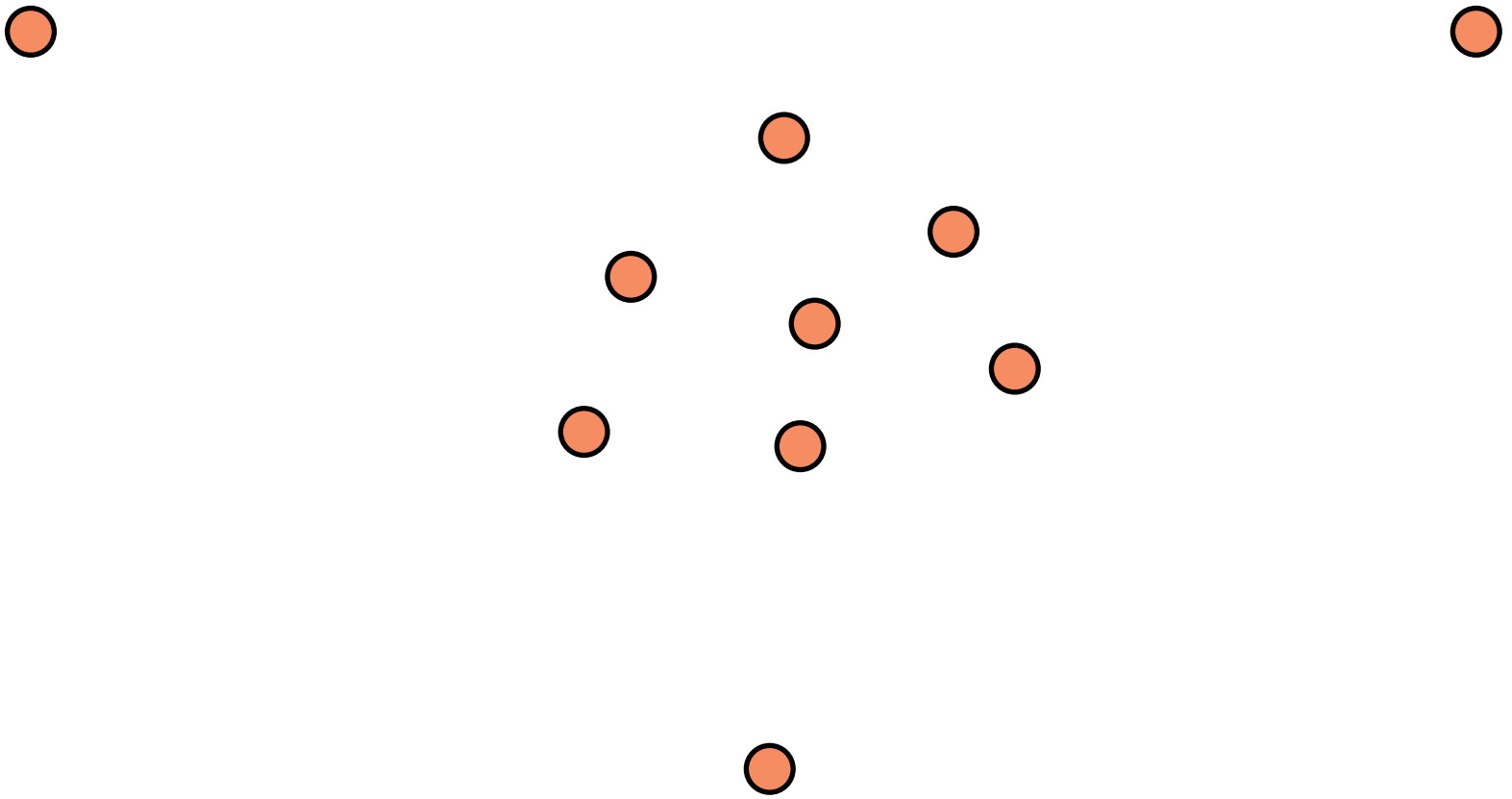
Inkrementeller Algorithmus

- Geg: Punktmenge $p_1, p_2, \ldots, p_n$
- Zur Vermeidung von Spezialfällen:
  Ergänze drei zusätzliche Punkte
  q_1, q_2, q_3,
  so dass alle p_i innerhalb des Dreiecks
  $[q_1, q_2, q_3]$ liegen ⇒ Einfügen nur im Inneren
- q_1, q_2, q_3 können leicht entfernt werden

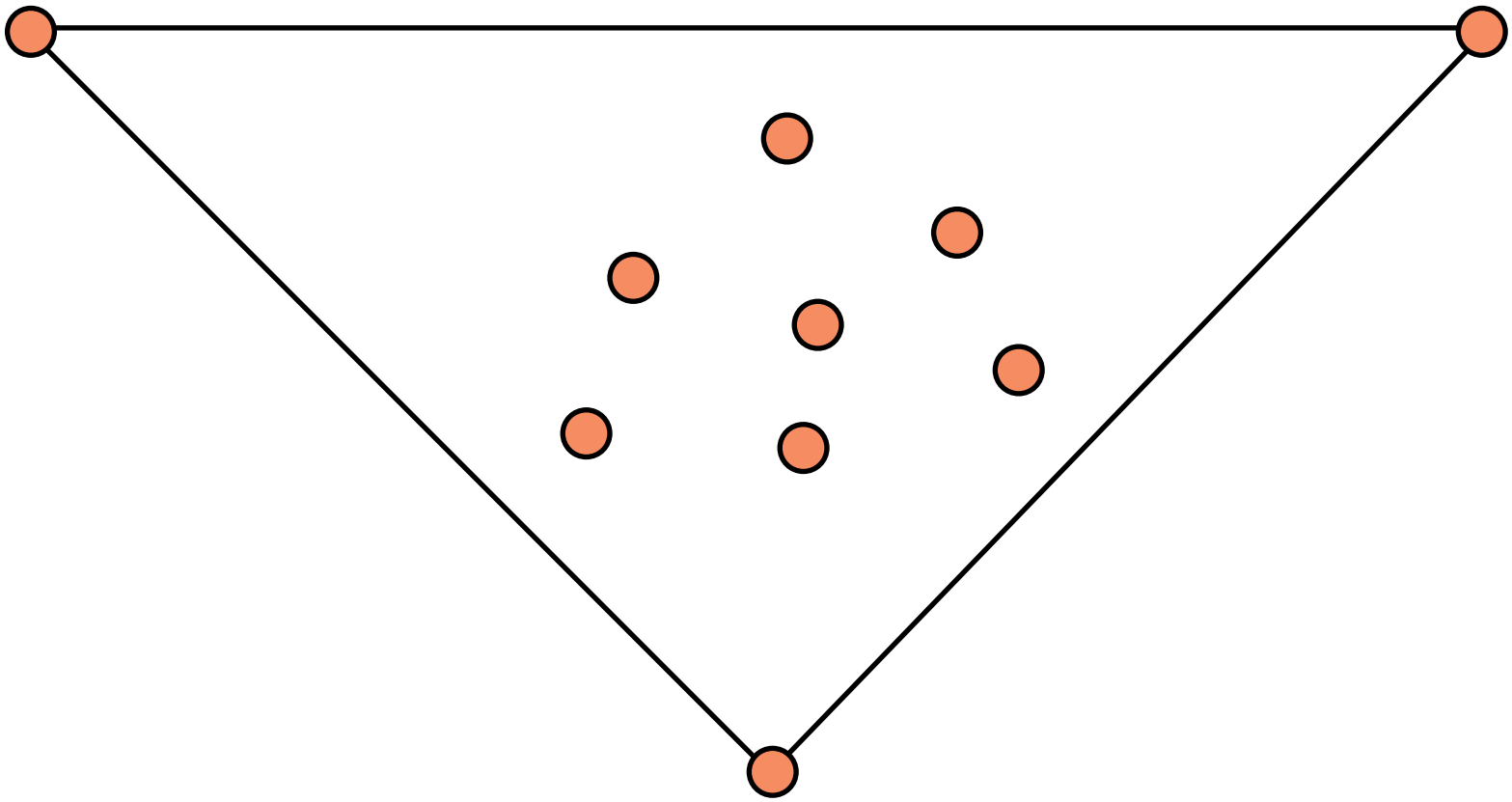
Inkrementeller Algorithmus



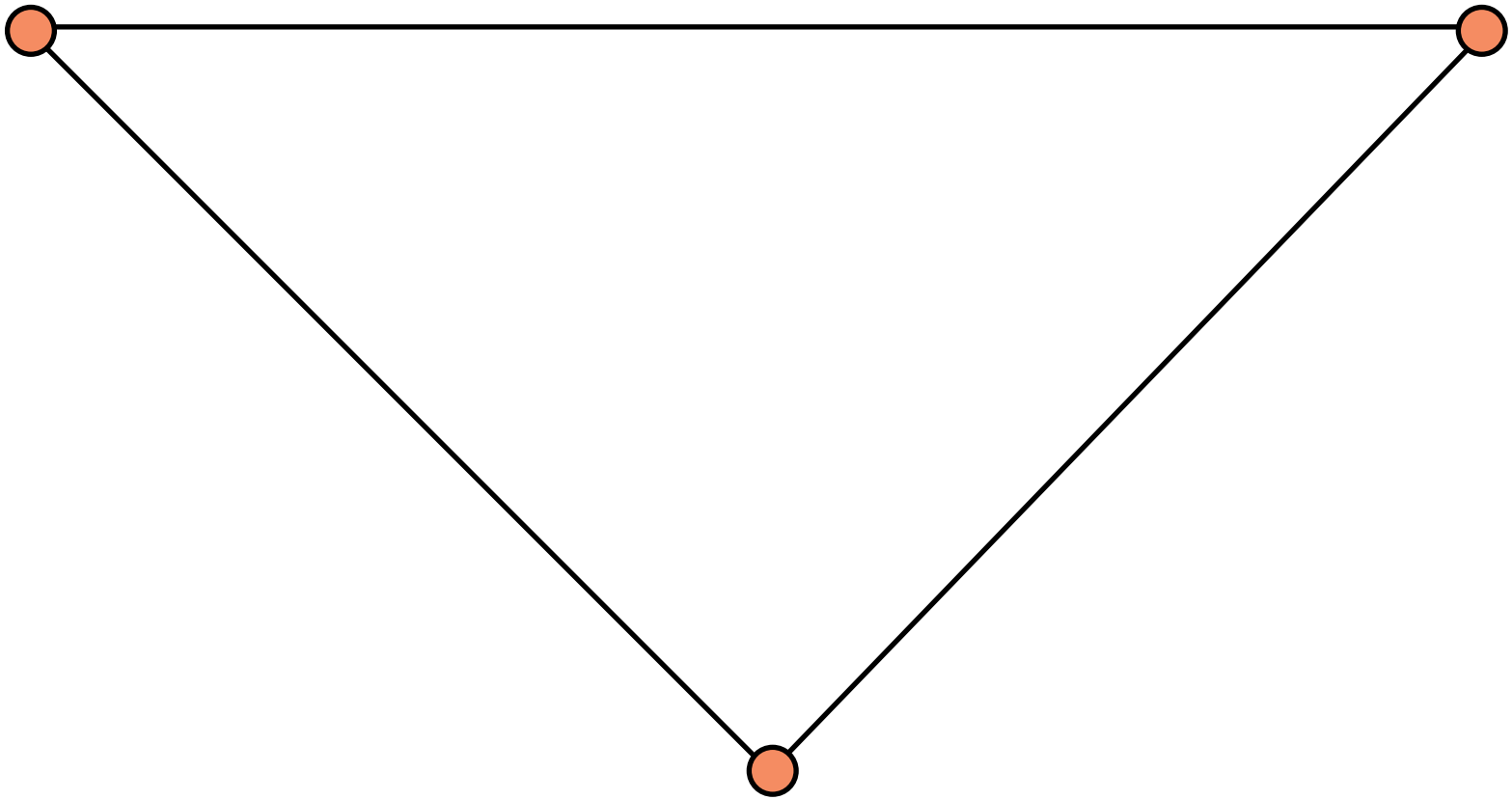
Inkrementeller Algorithmus



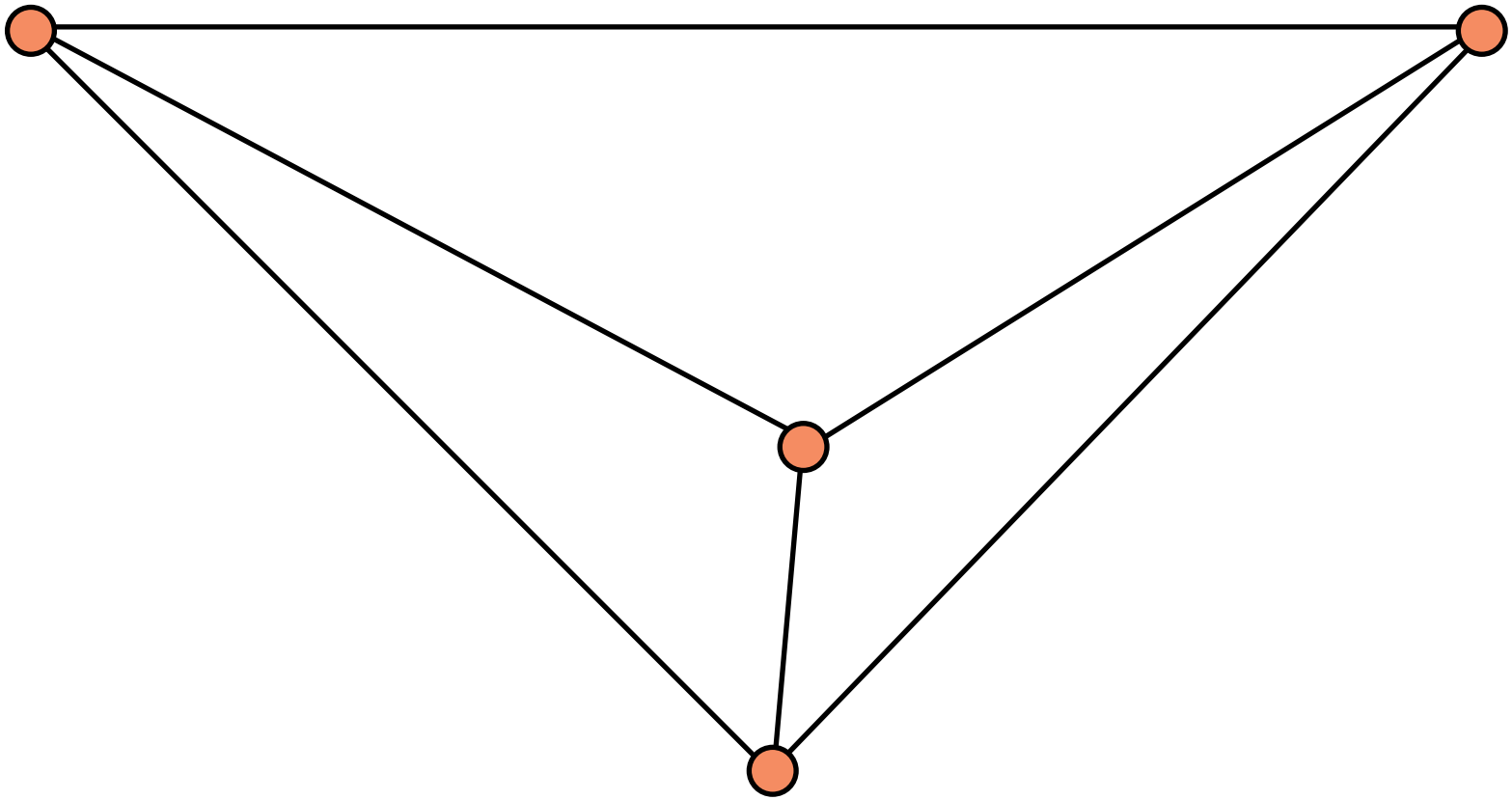
Inkrementeller Algorithmus



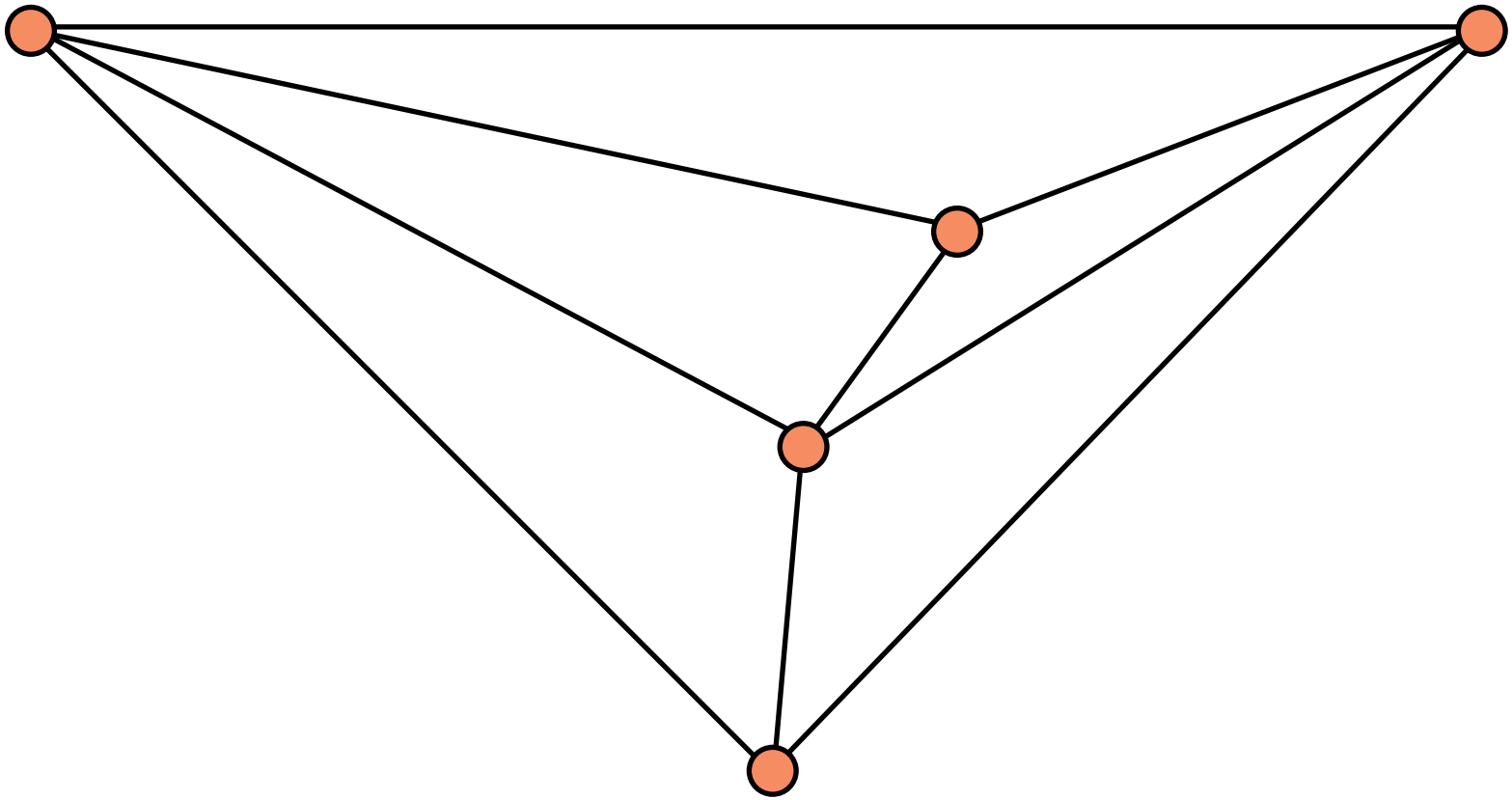
Inkrementeller Algorithmus



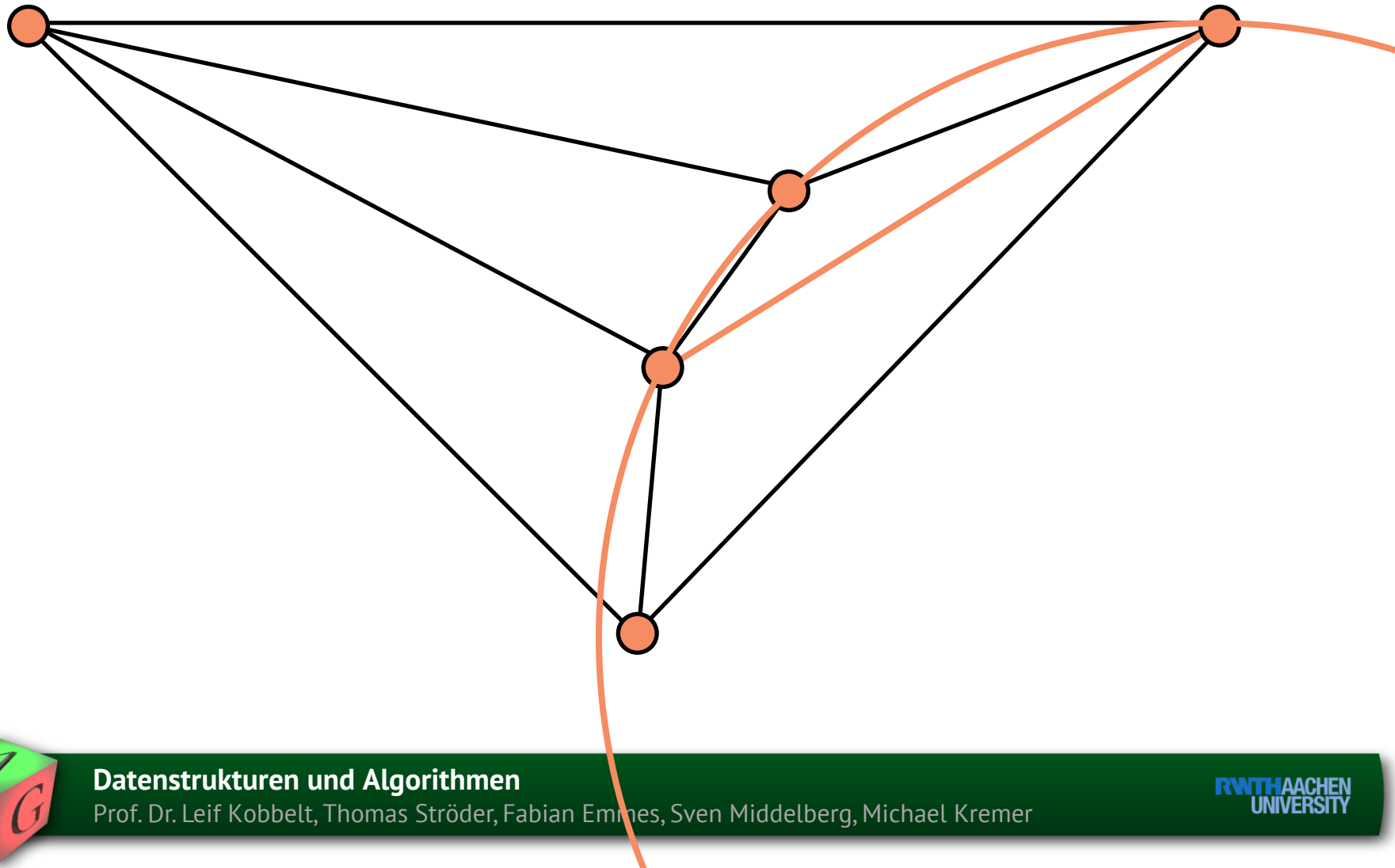
Inkrementeller Algorithmus



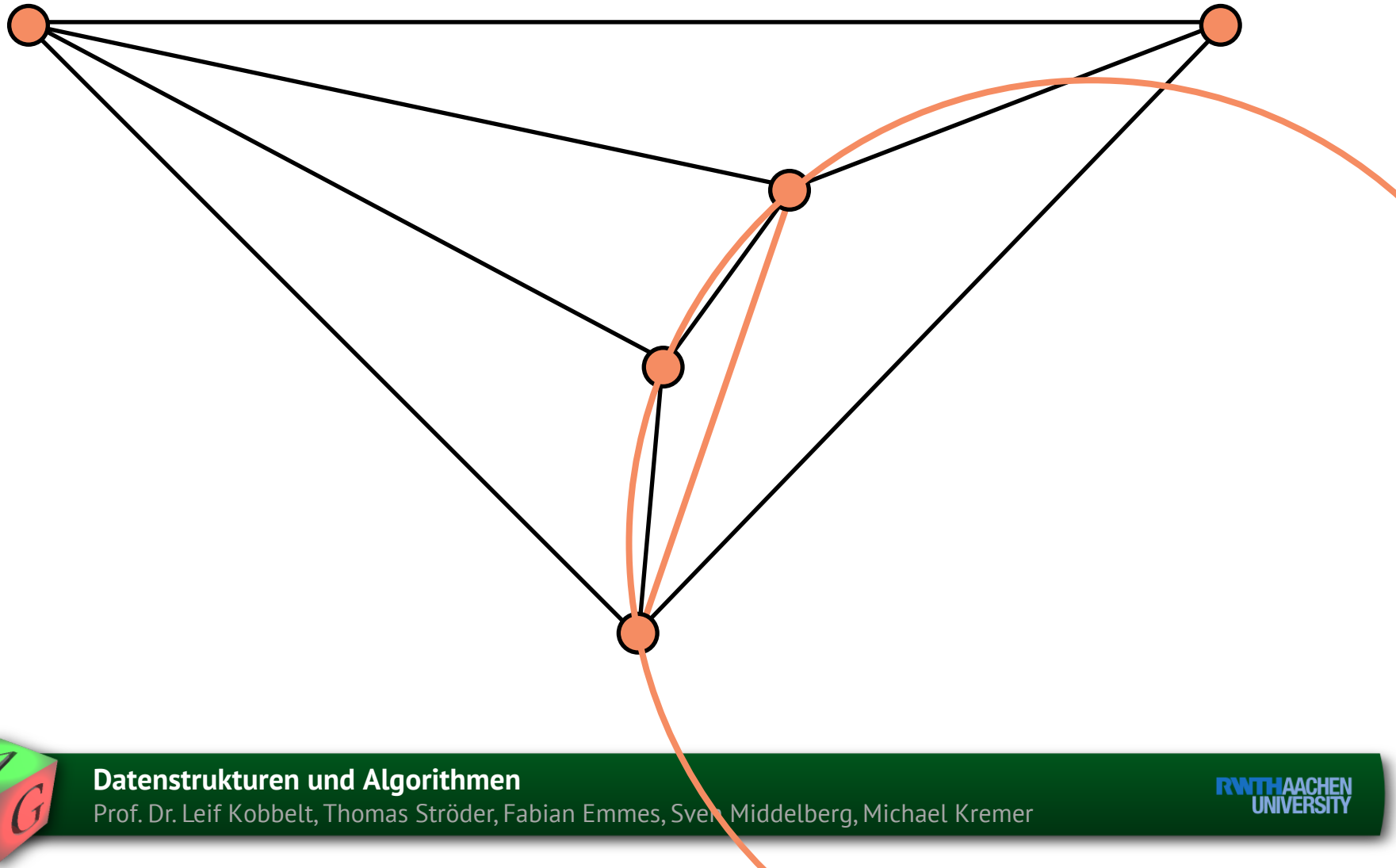
Inkrementeller Algorithmus



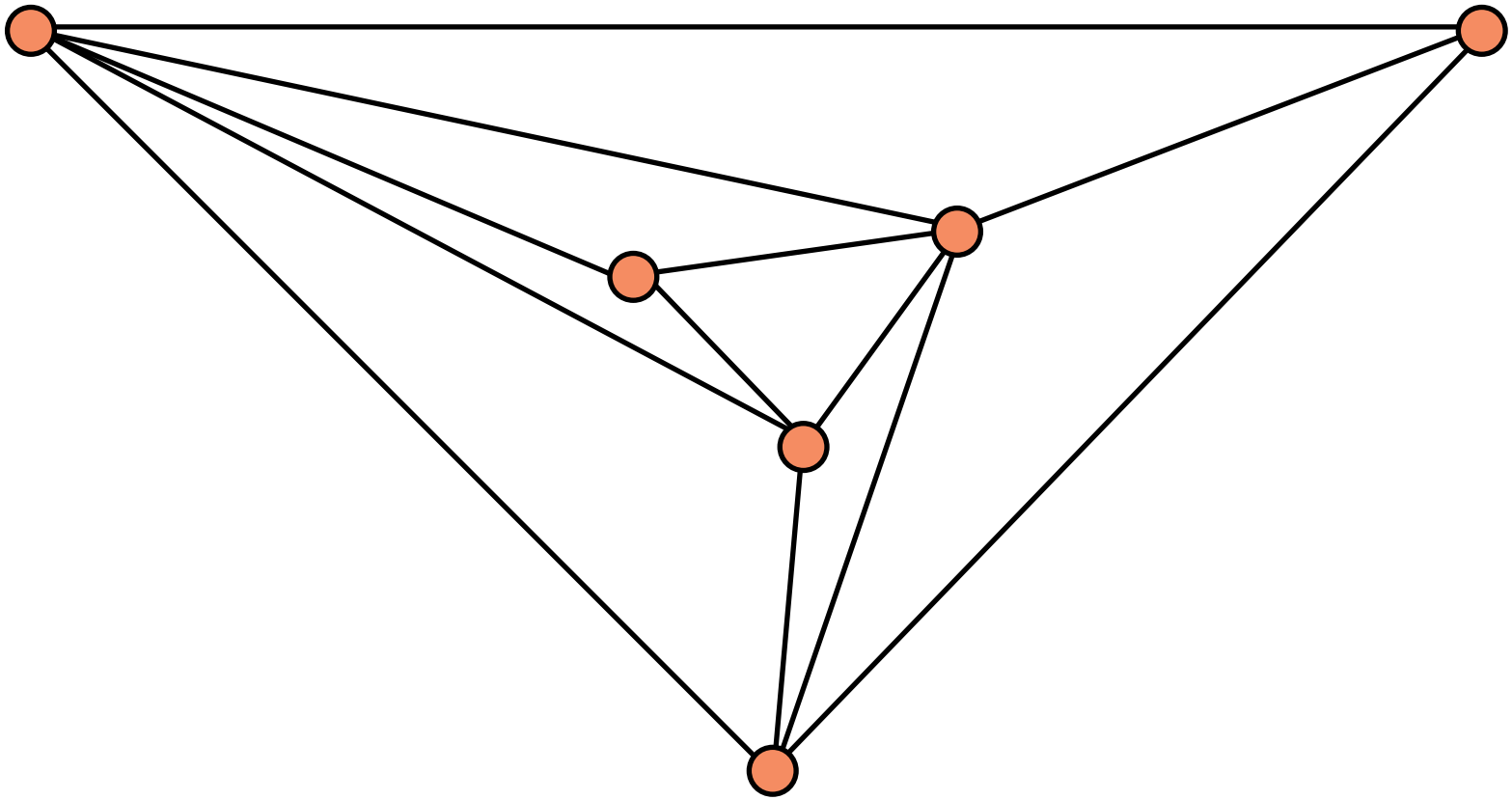
Inkrementeller Algorithmus



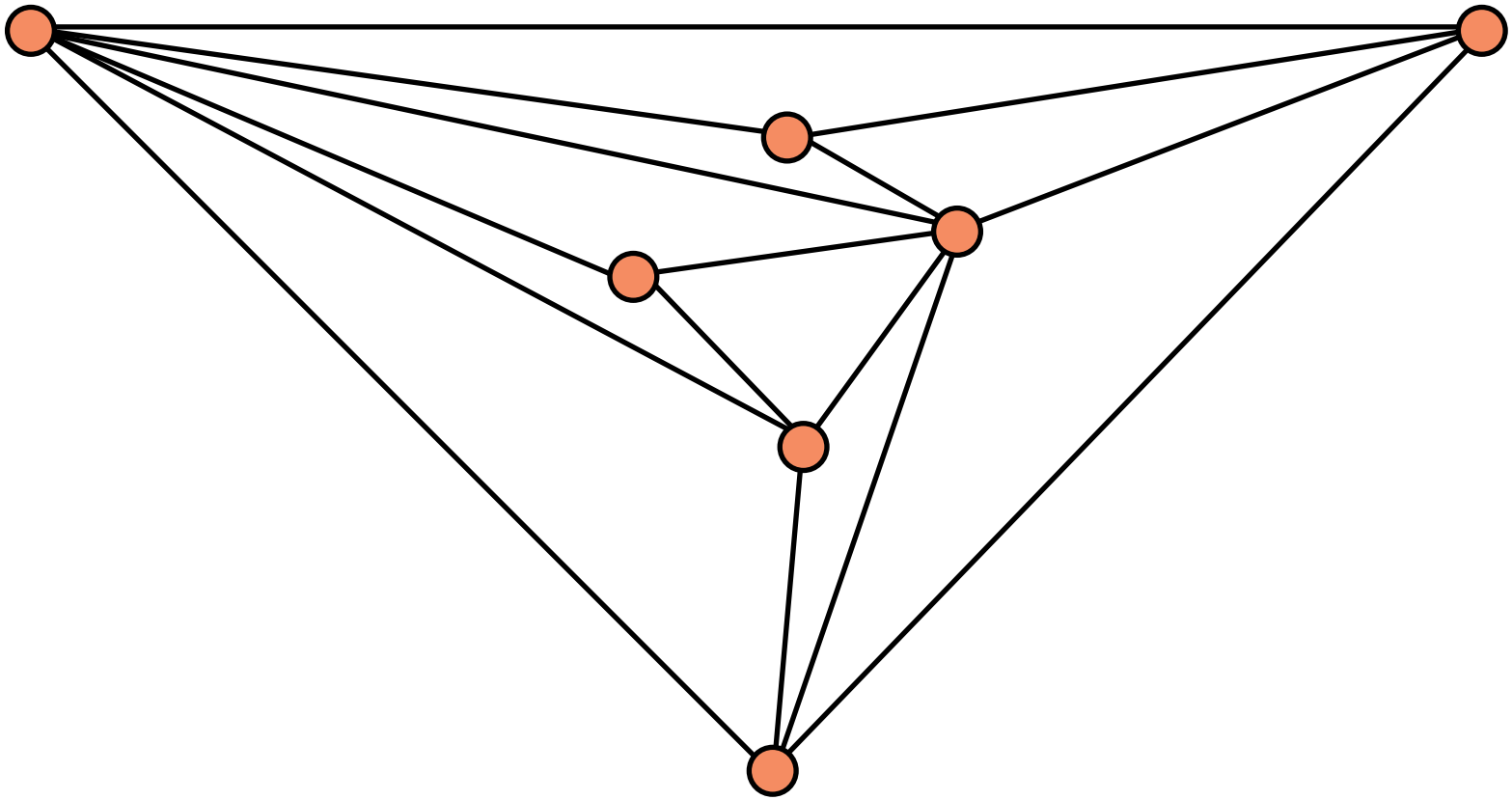
Inkrementeller Algorithmus



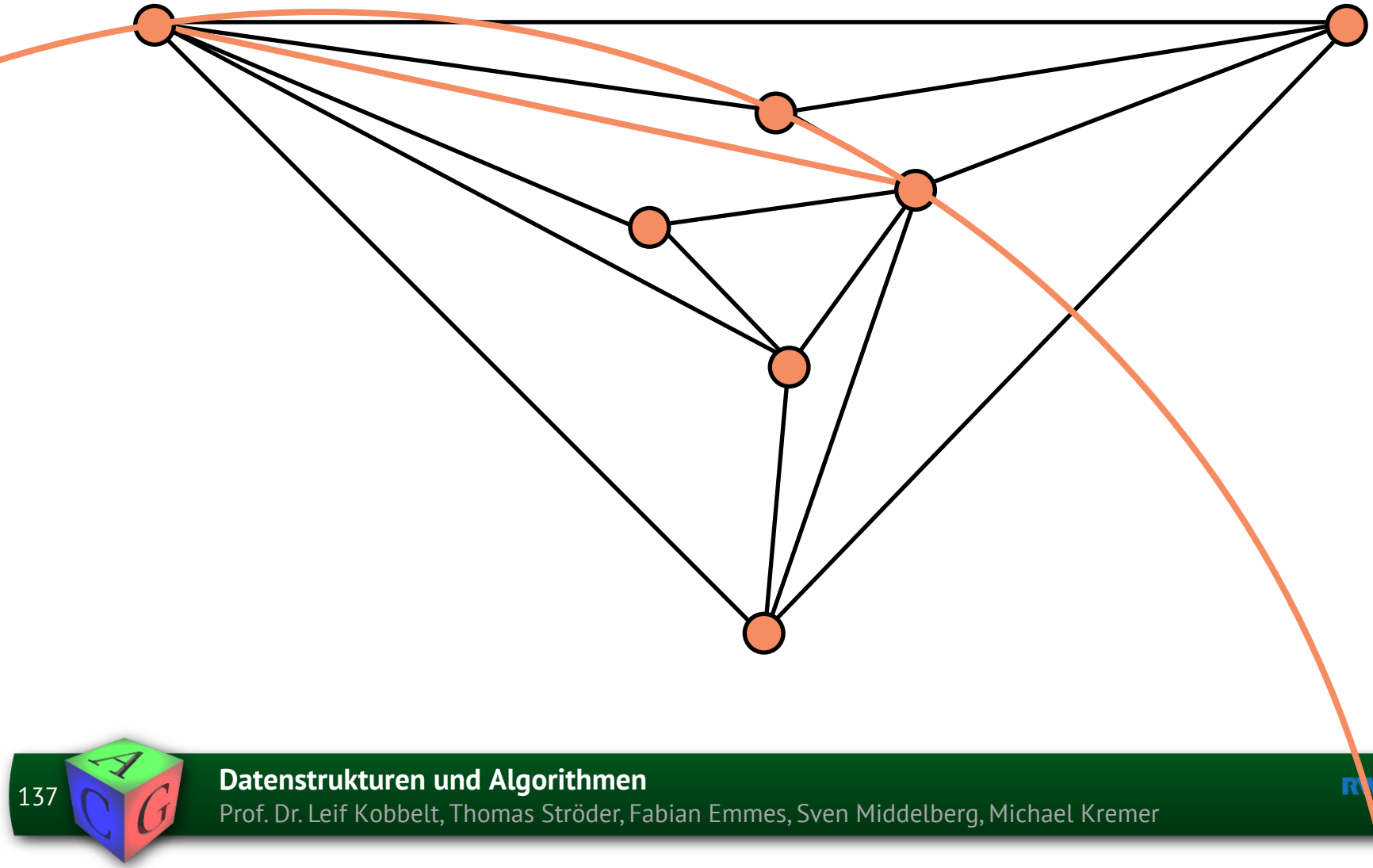
Inkrementeller Algorithmus



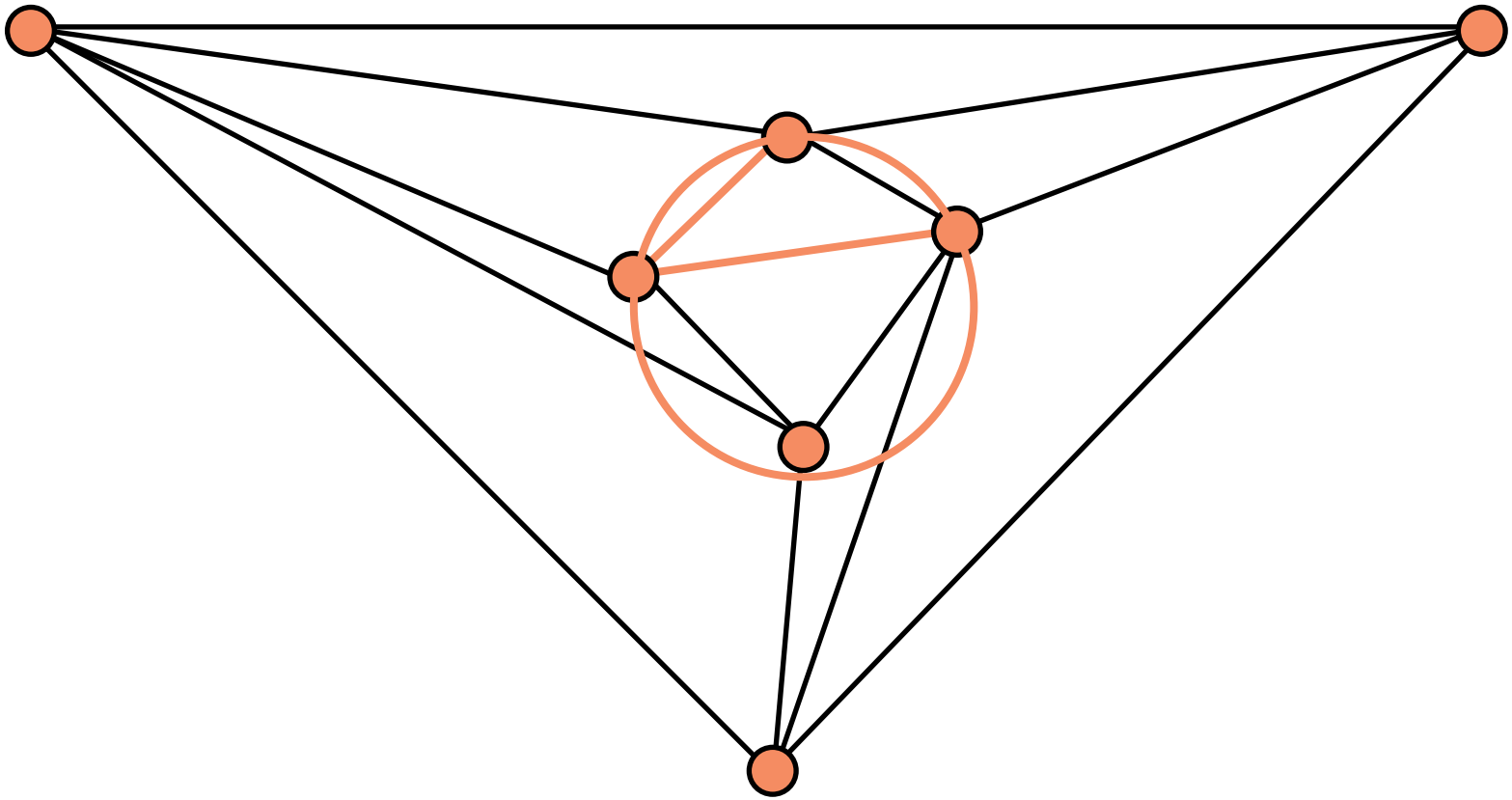
Inkrementeller Algorithmus



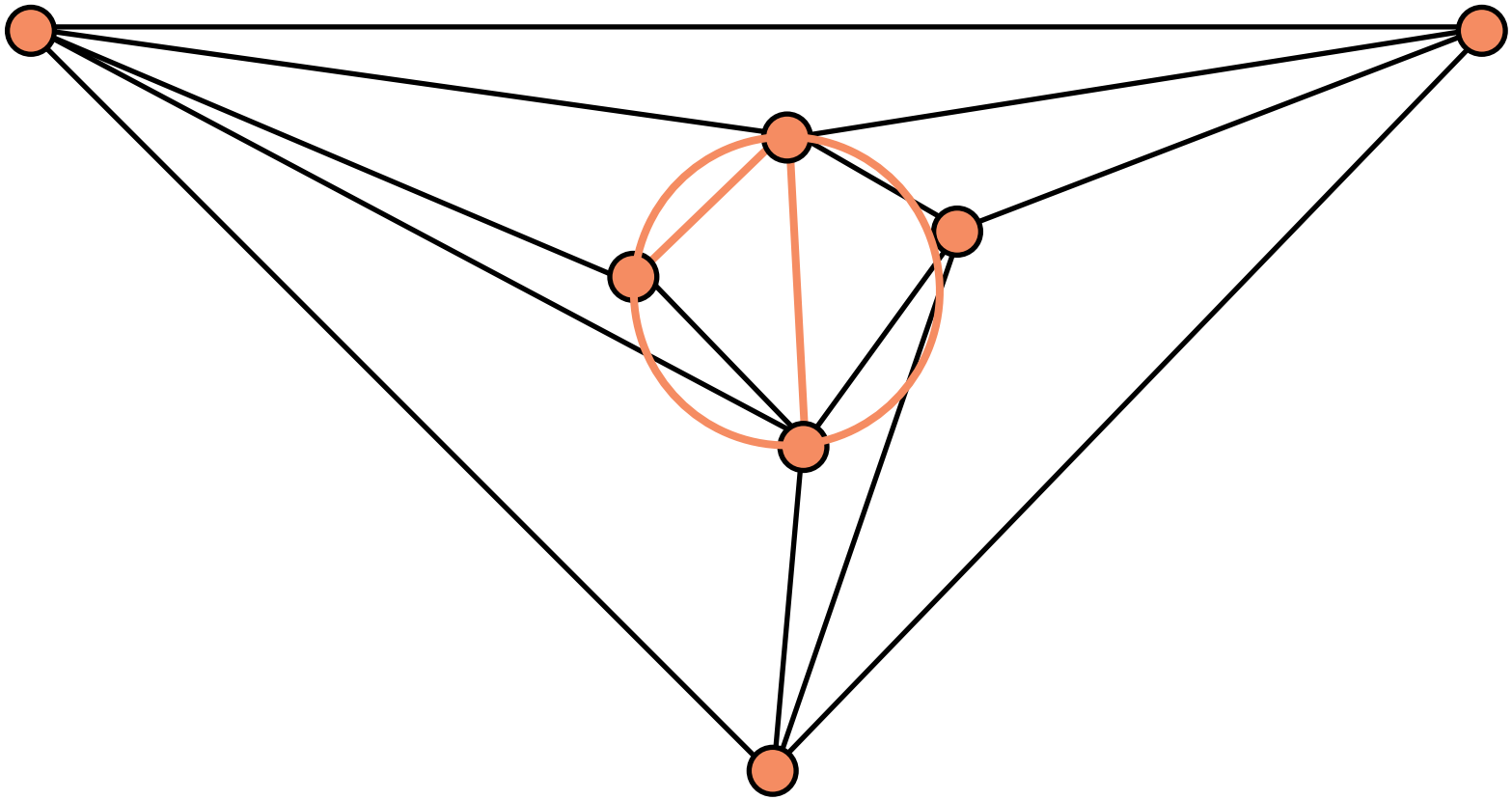
Inkrementeller Algorithmus



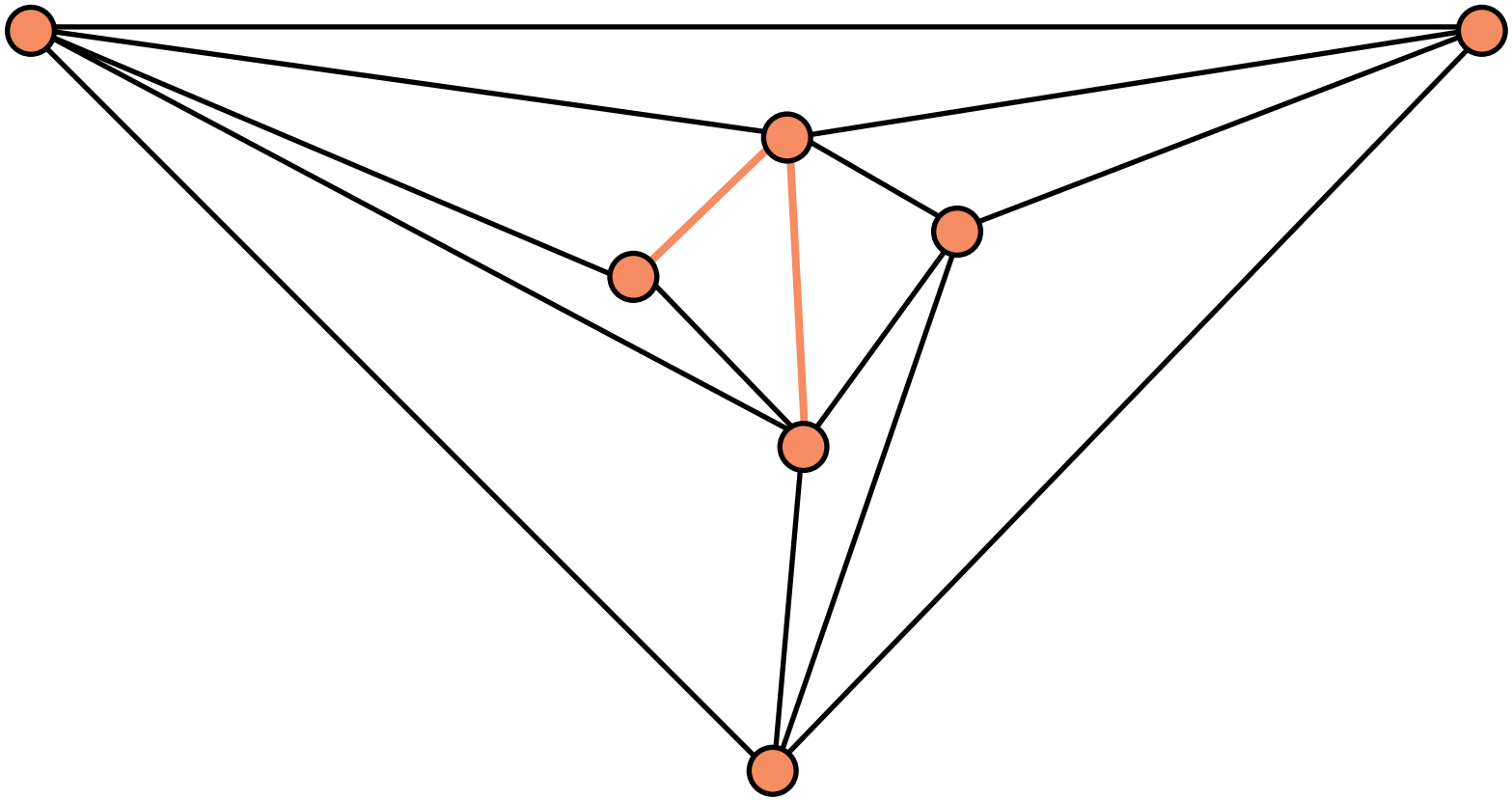
Inkrementeller Algorithmus



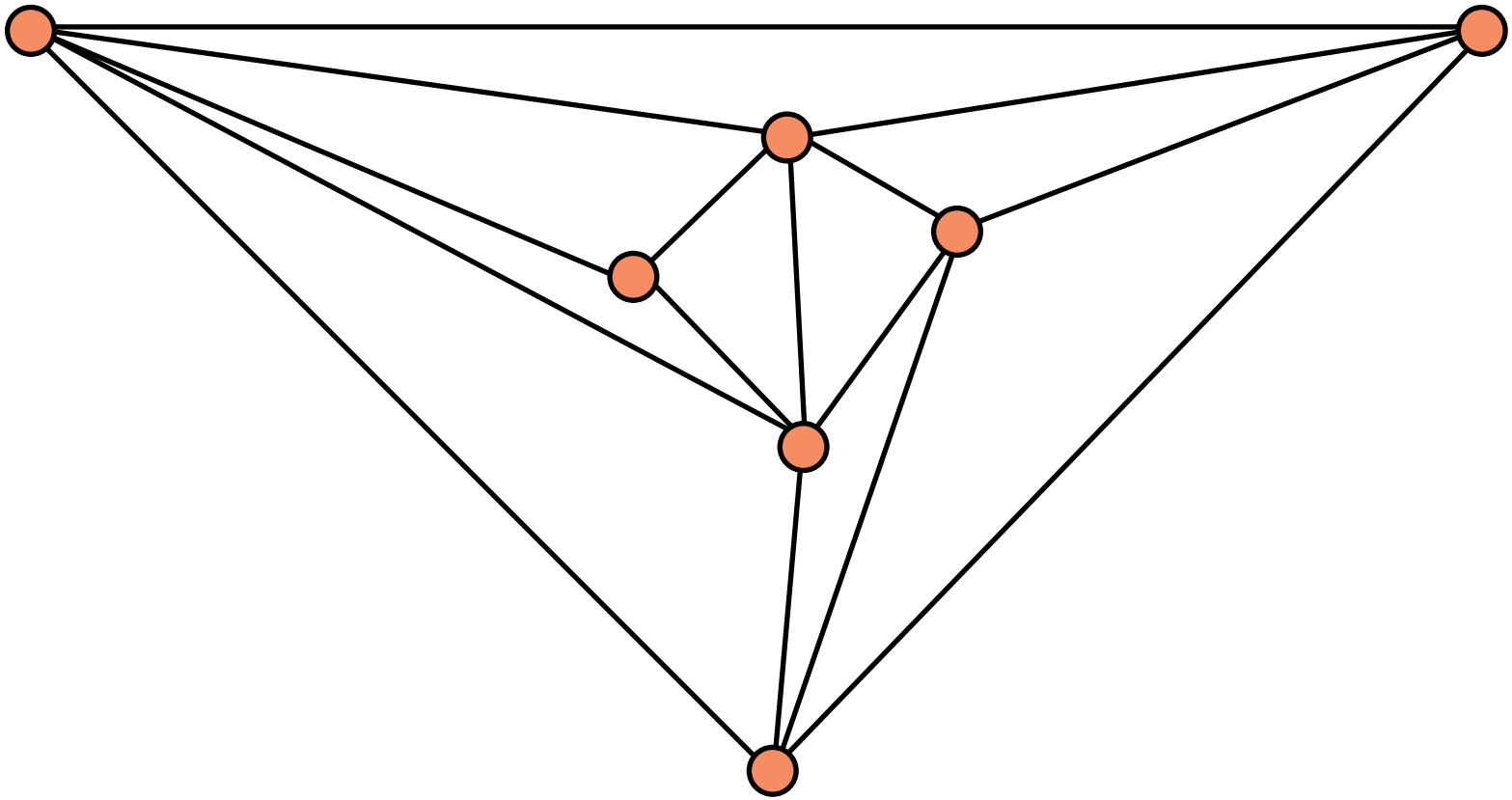
Inkrementeller Algorithmus



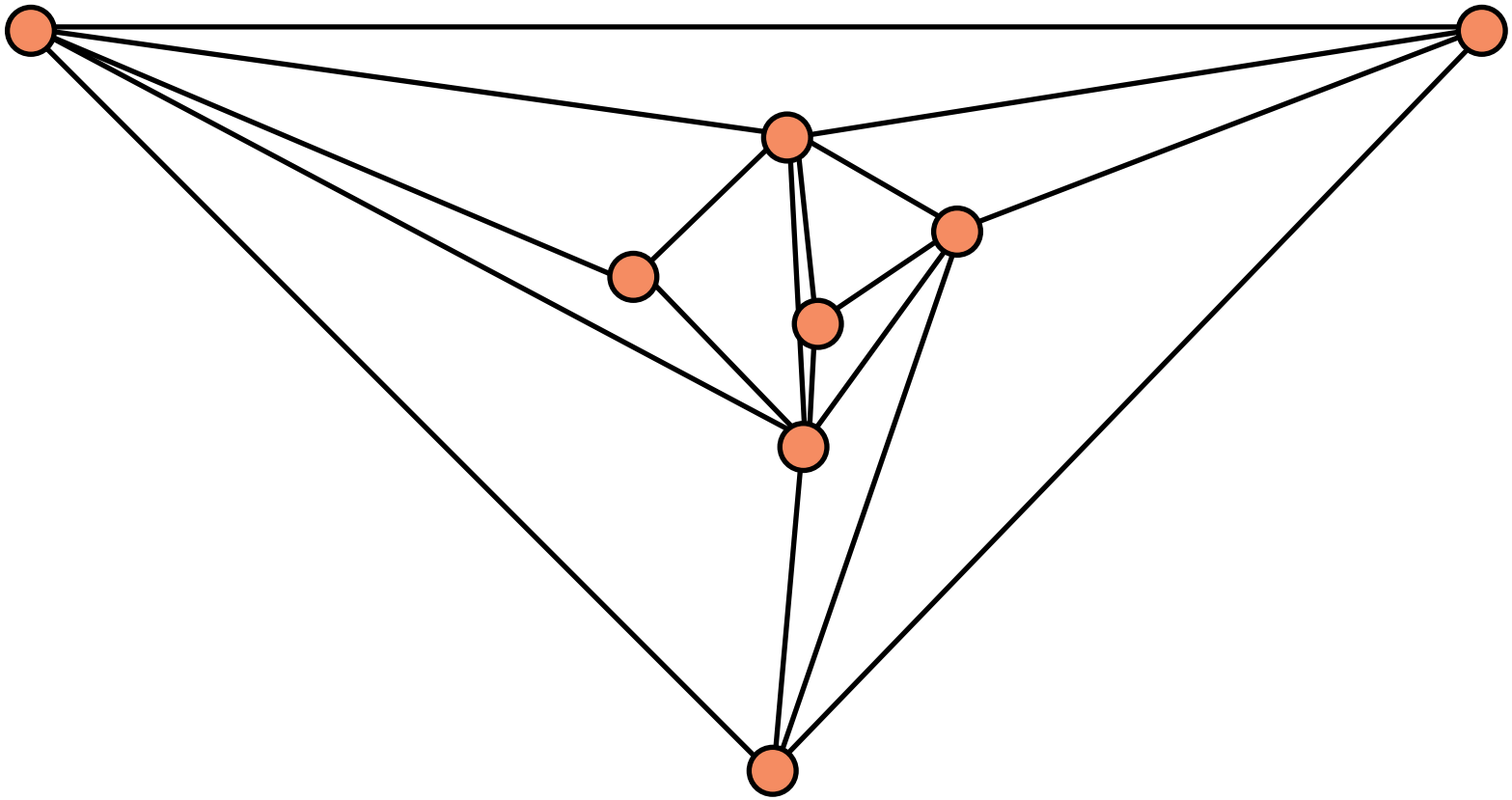
Inkrementeller Algorithmus



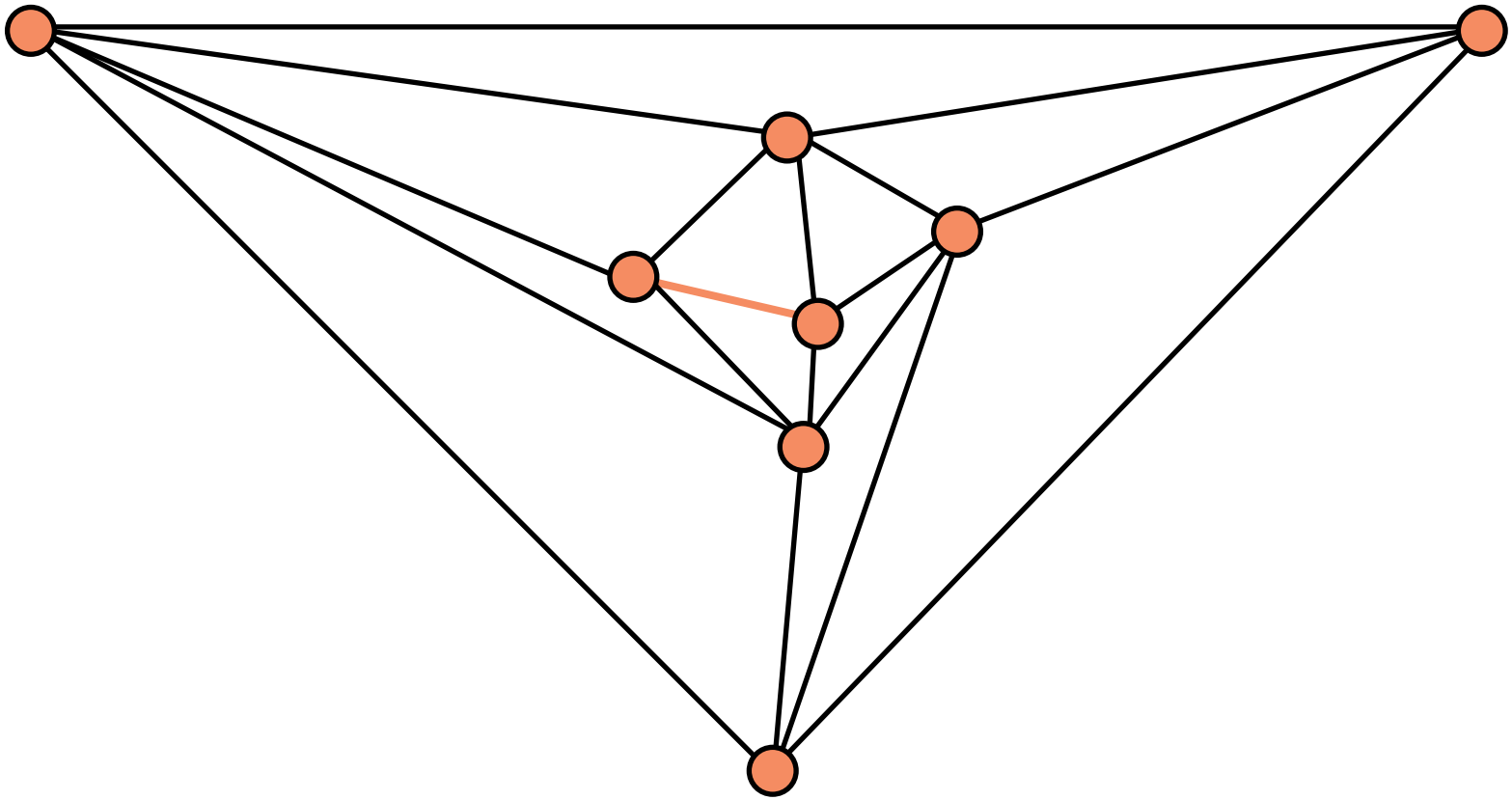
Inkrementeller Algorithmus



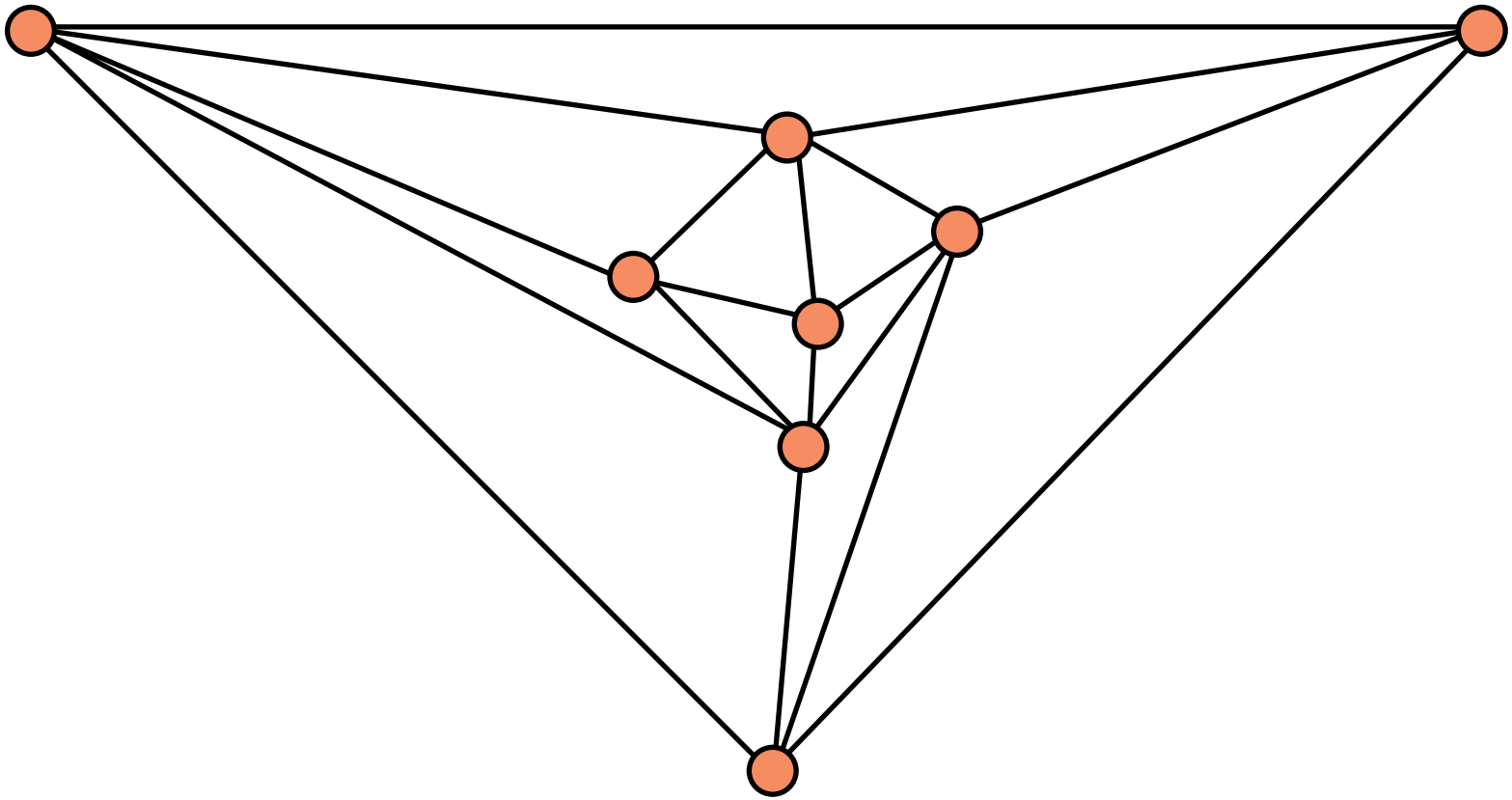
Inkrementeller Algorithmus



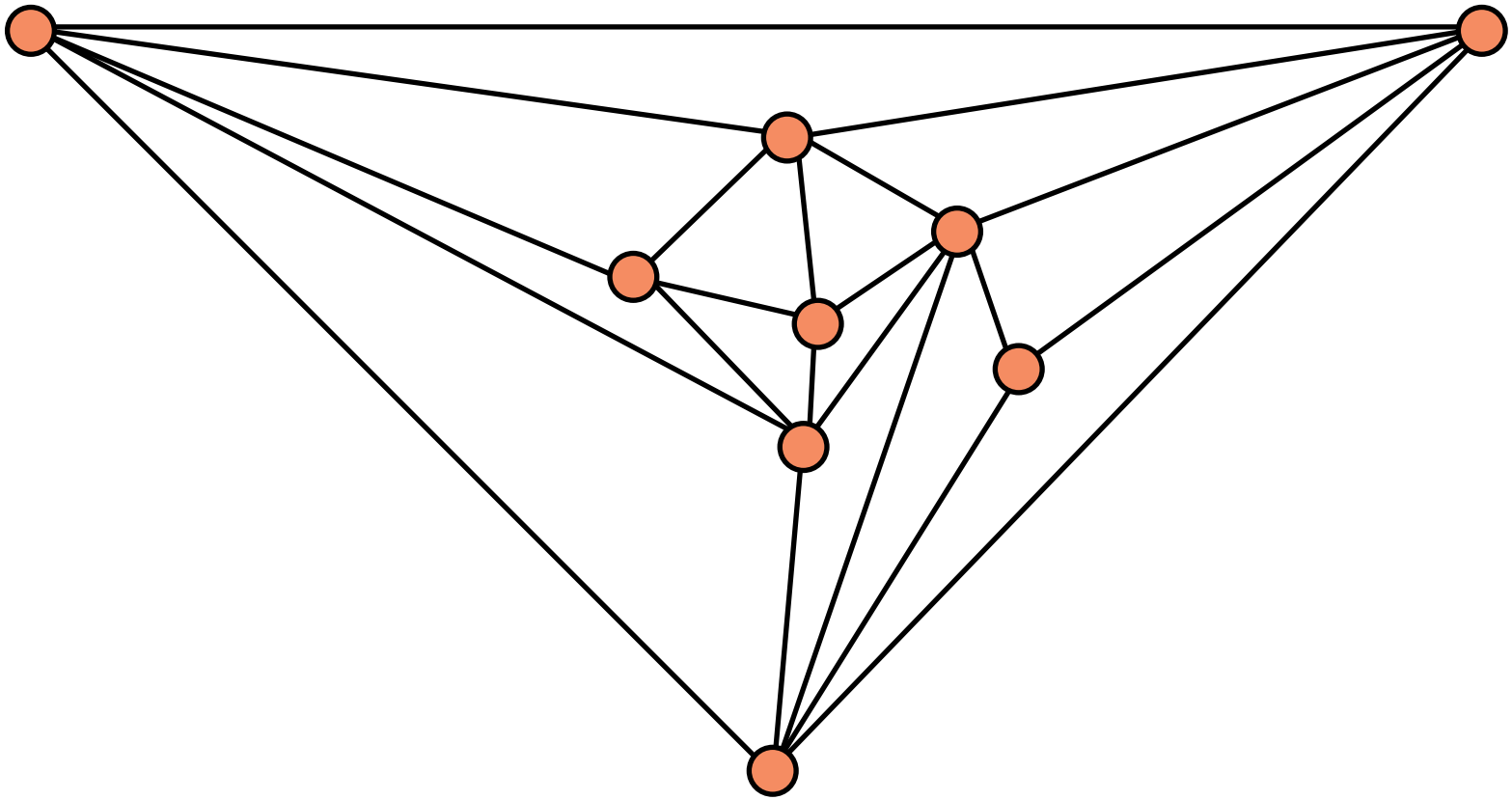
Inkrementeller Algorithmus



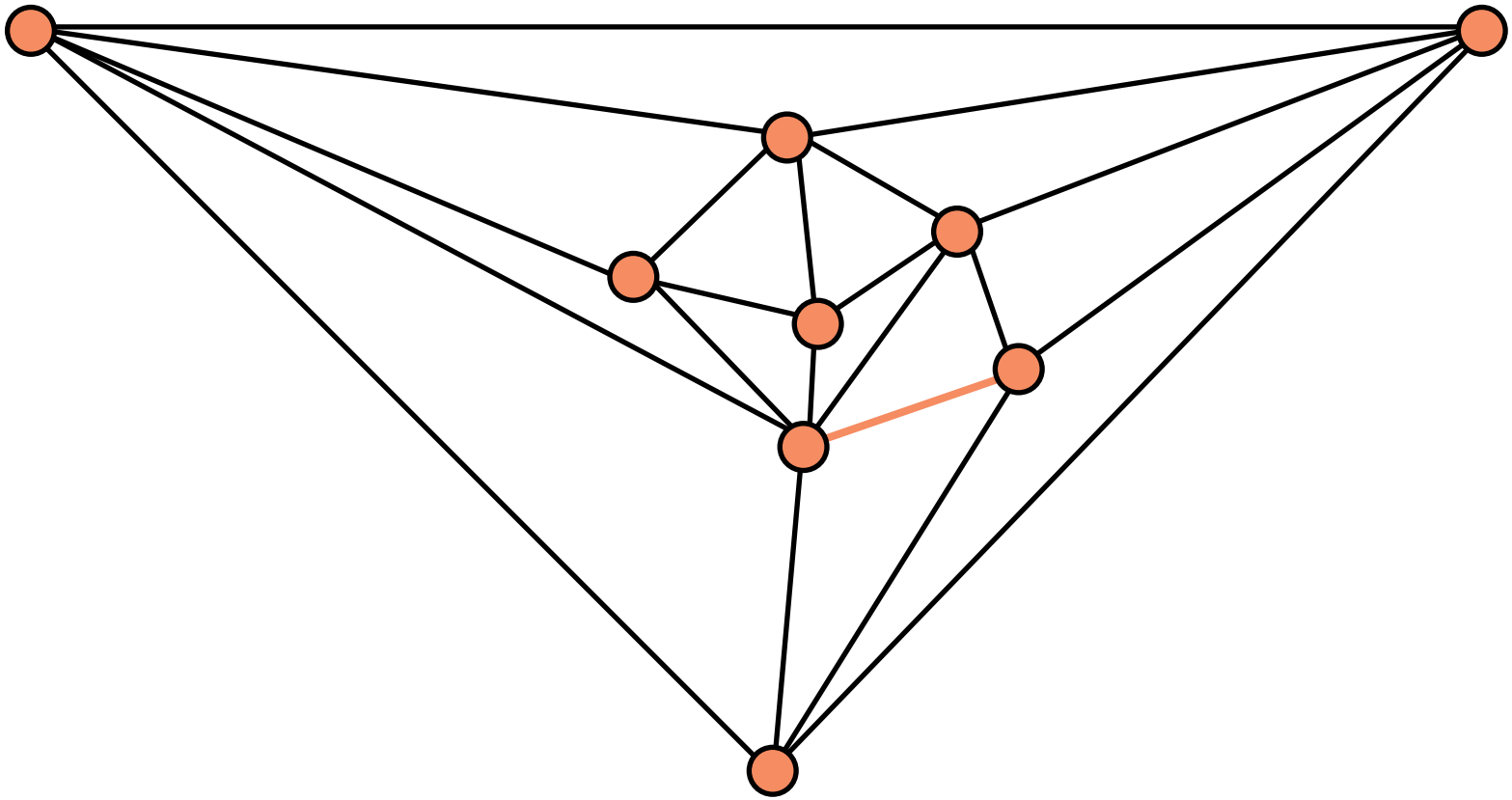
Inkrementeller Algorithmus



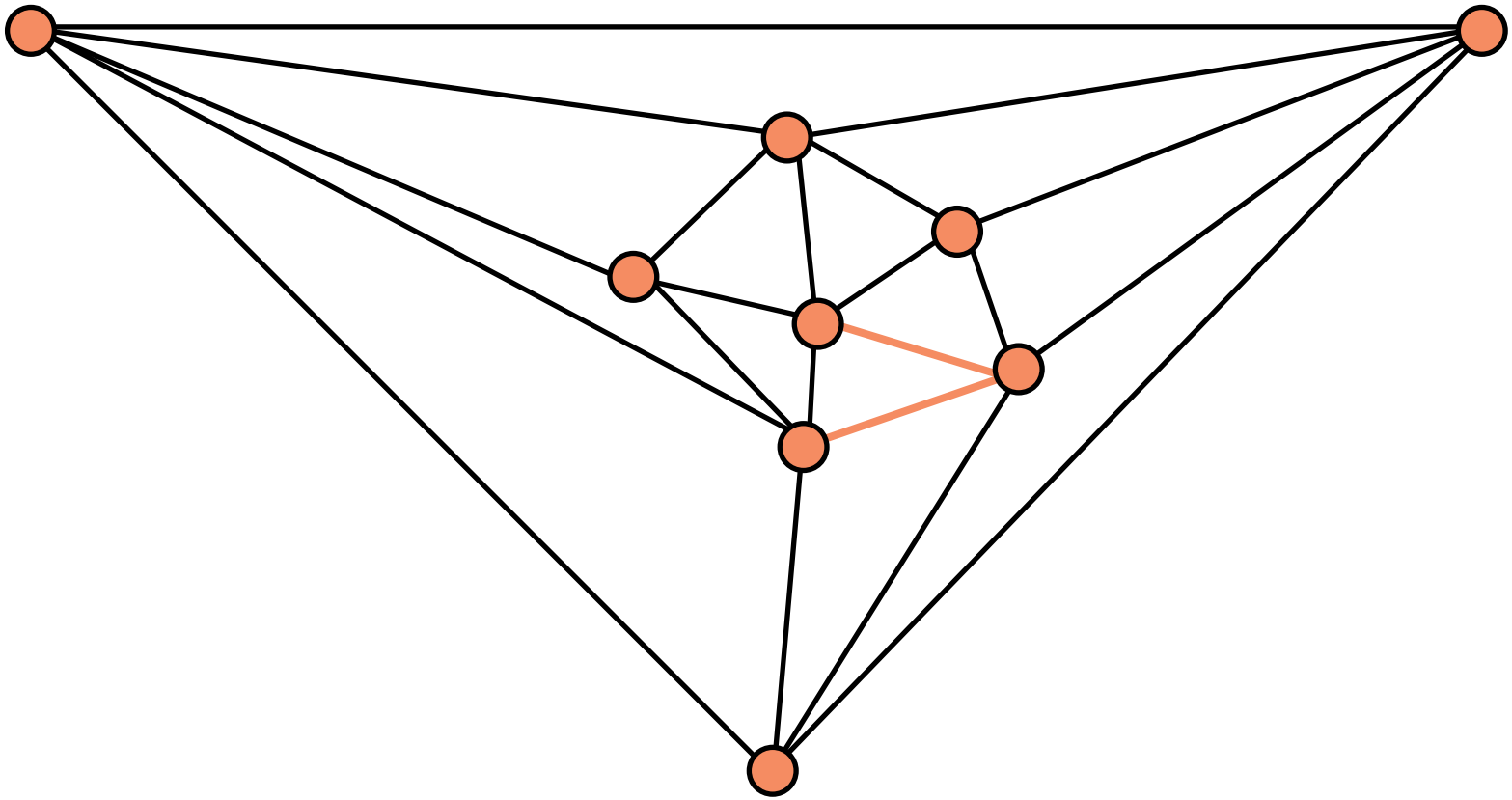
Inkrementeller Algorithmus



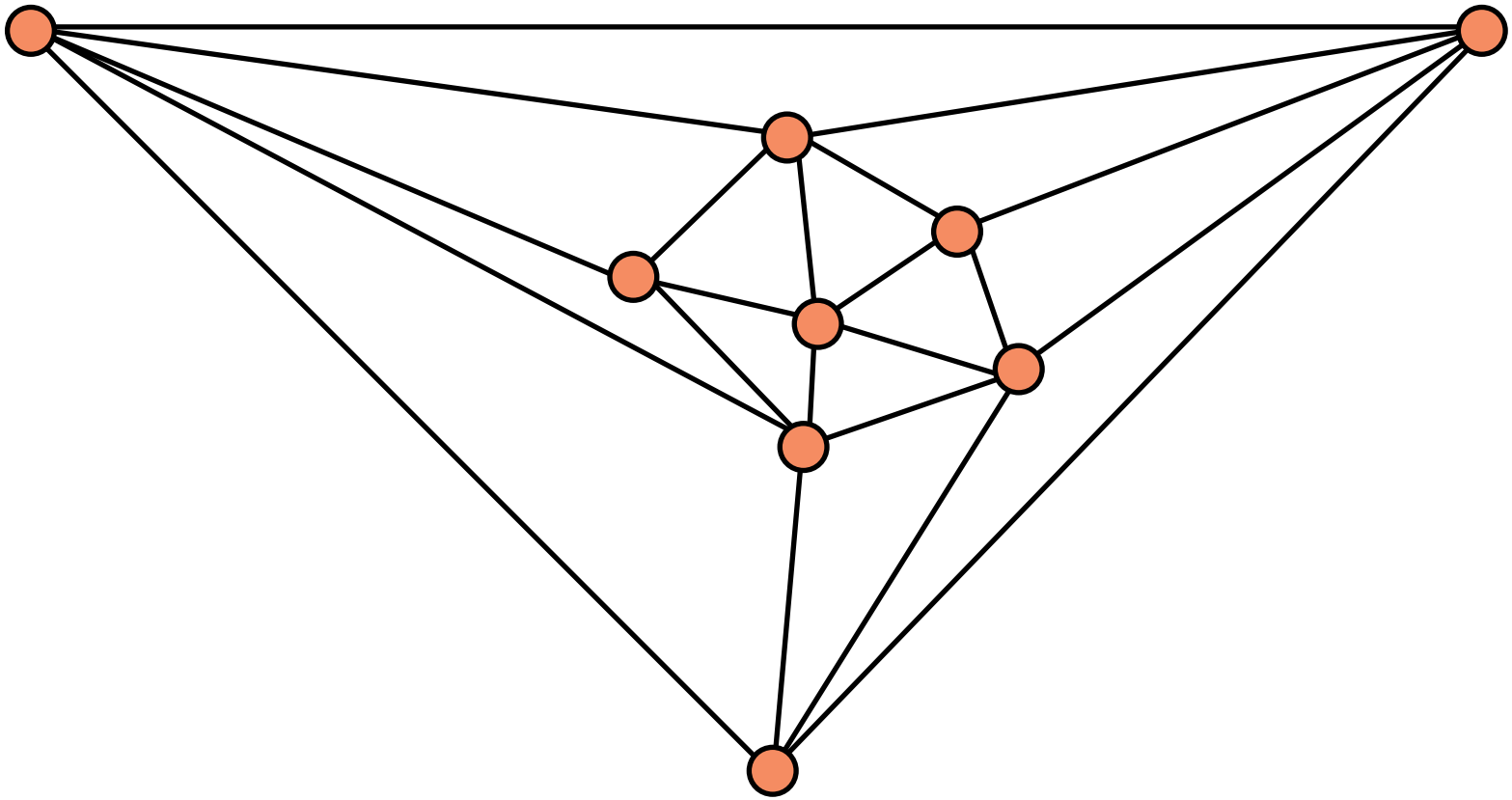
Inkrementeller Algorithmus



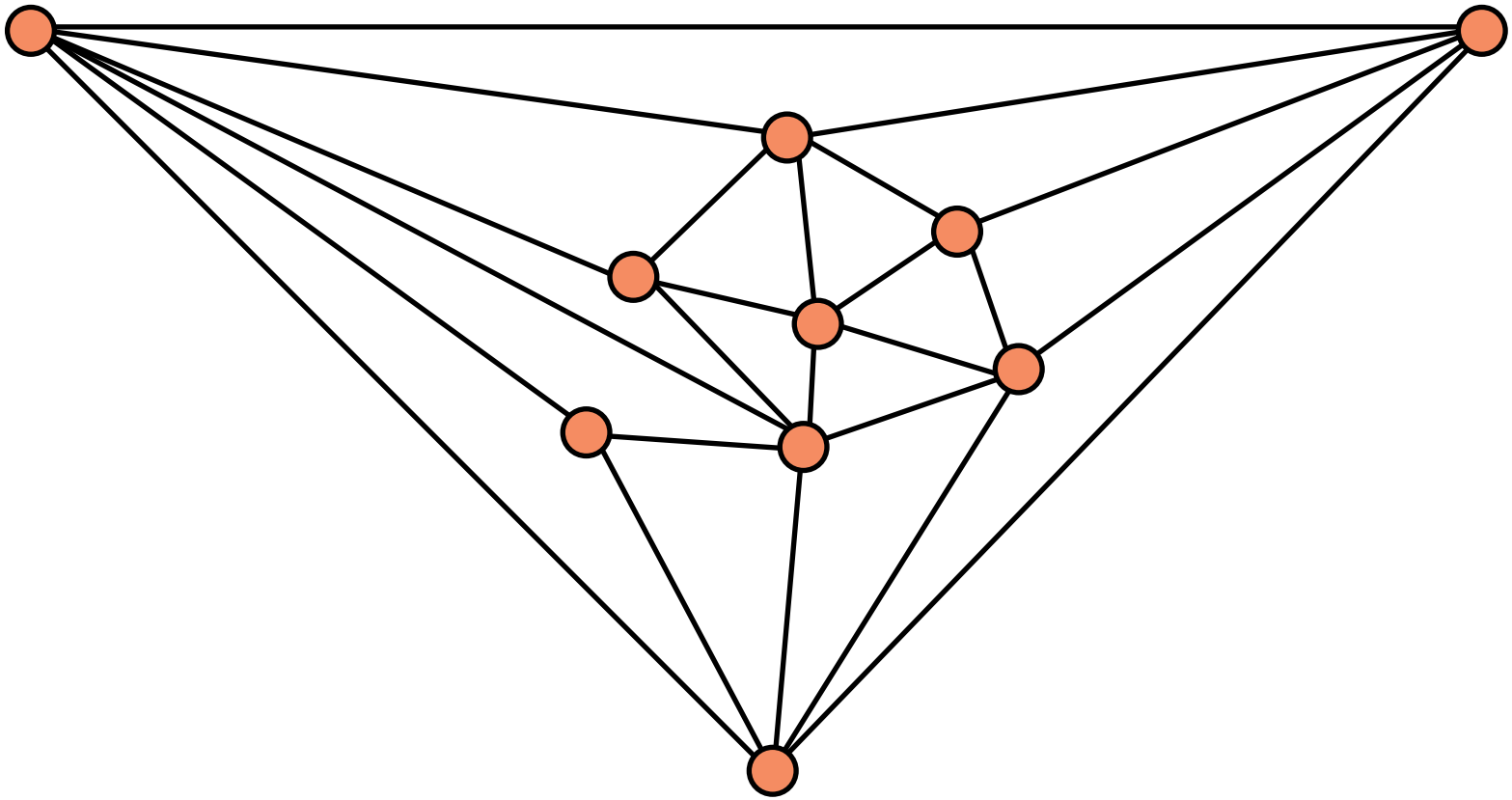
Inkrementeller Algorithmus



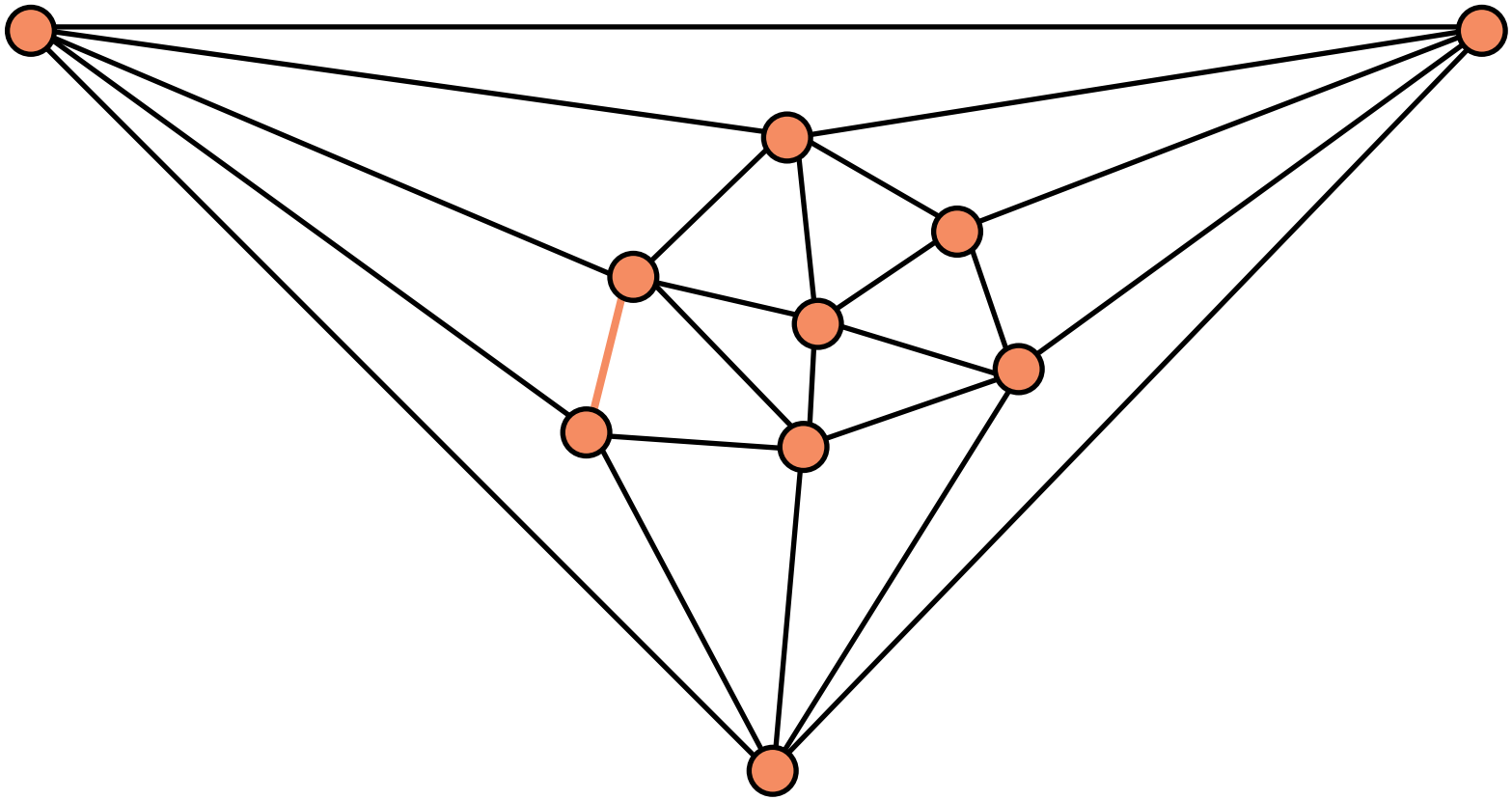
Inkrementeller Algorithmus



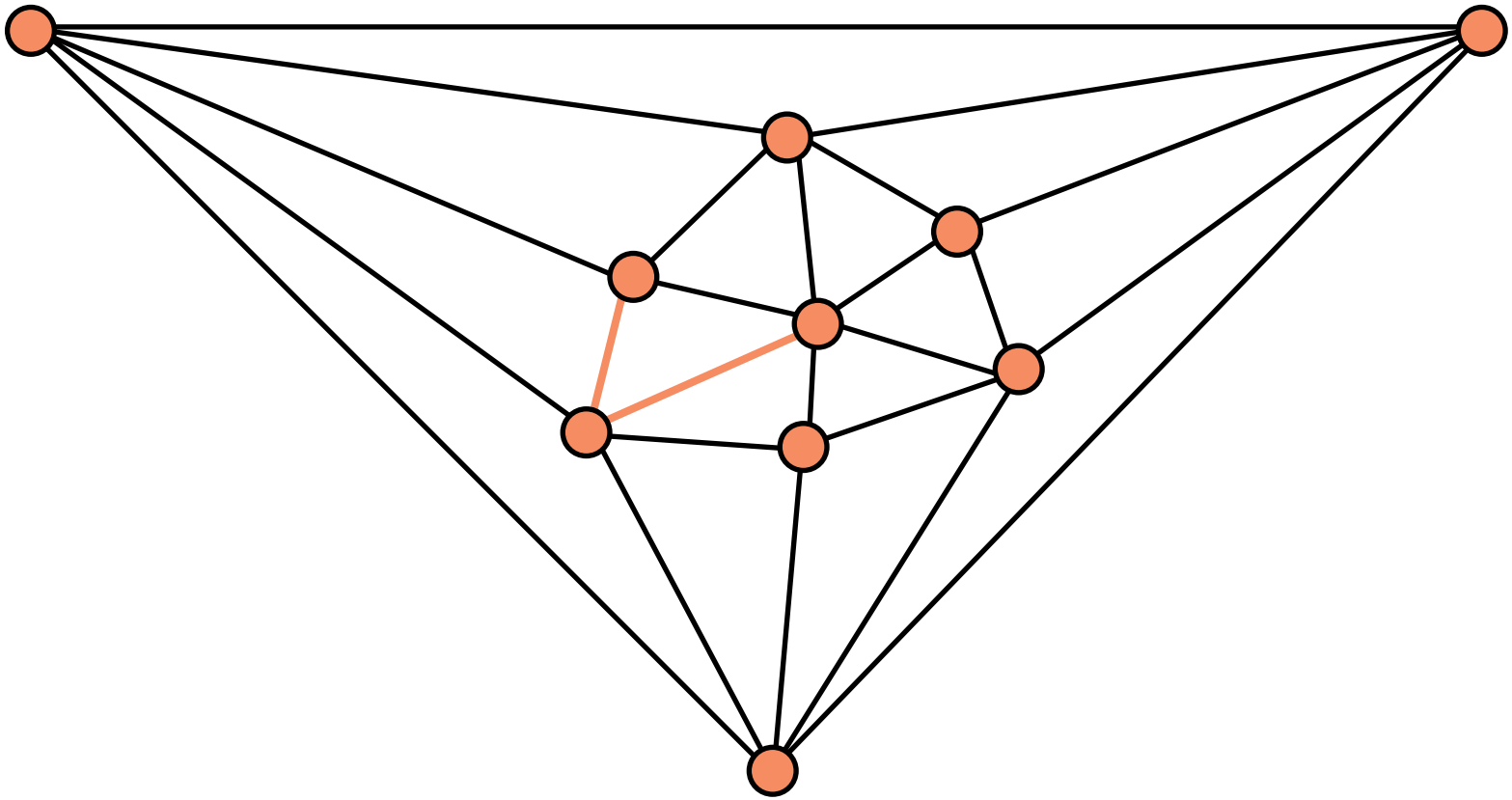
Inkrementeller Algorithmus



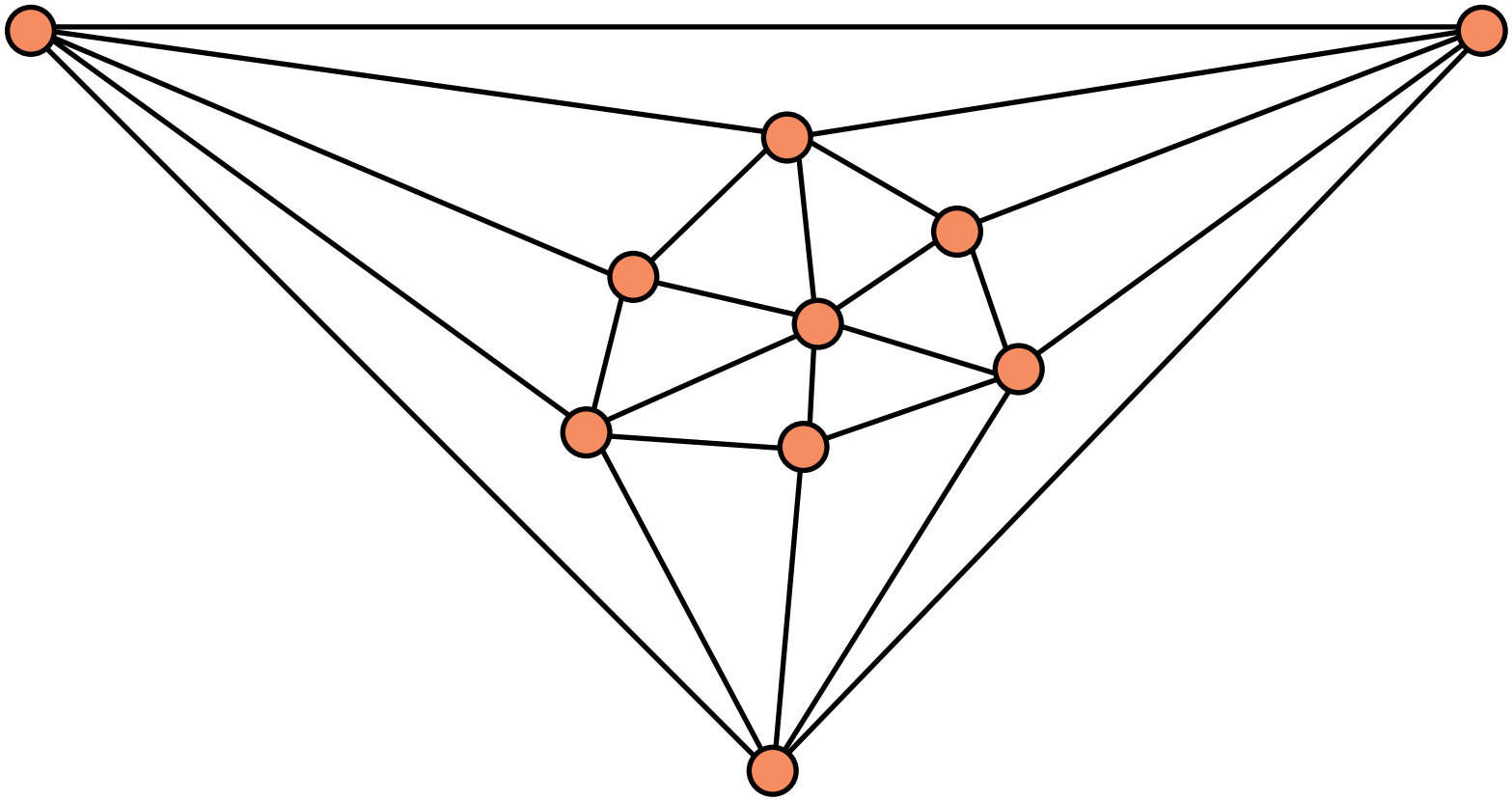
Inkrementeller Algorithmus



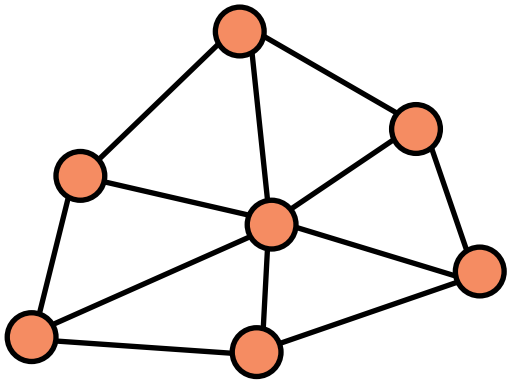
Inkrementeller Algorithmus



Inkrementeller Algorithmus



Inkrementeller Algorithmus



Inkrementeller Algorithmus

- Für jeden Punkt p_i — O(n)
  - Suche das entsprechende Dreieck — O(n)
  - Füge den neuen Punkt ein — O(1)
  - Flippe Kanten bis die Delaunay-Bedingungen wieder hergestellt sind. — O(k)

Aufwandsabschätzung

- Der maximale Knotengrad k (Valenz) kann normalerweise als beschränkt angenommen werden (zumindest als unabhängig von n)
- Gesamtaufwand $O(n^2)$
- Beschleunigung durch bessere Suche

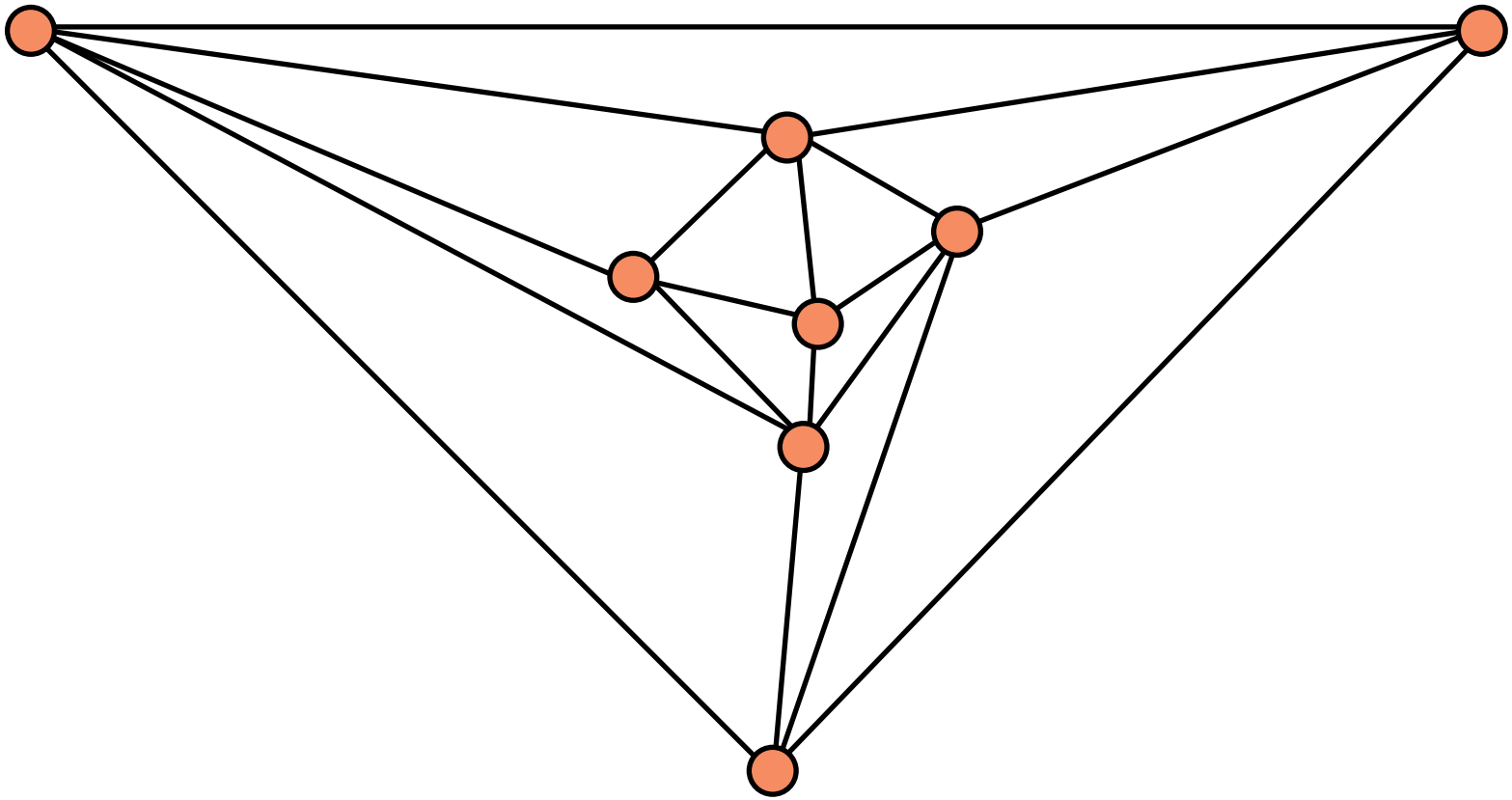
Beschleunigte Suche

- Standard Ansatz:
  Verwalte die Dreiecke in einem Suchbaum, um schneller auf die Elemente zuzugreifen
- Verbesserung: $O(\log n)$ statt $O(n)$

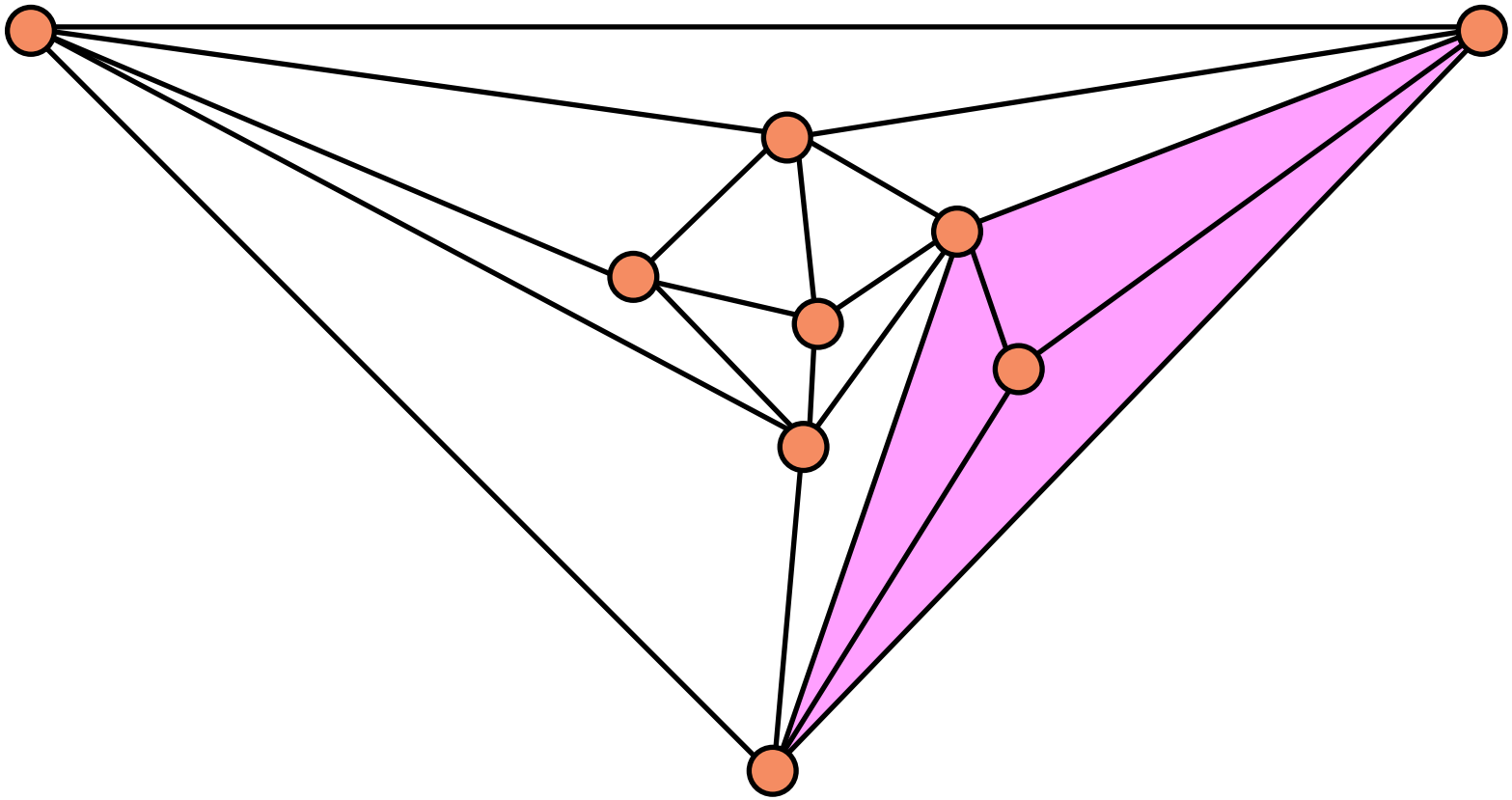
Beschleunigte Suche

- Am Anfang existiert nur ein Dreieck
- Bei jedem Einfügeschritt werden m Dreiecke entfernt und m+2 Dreiecke ergänzt

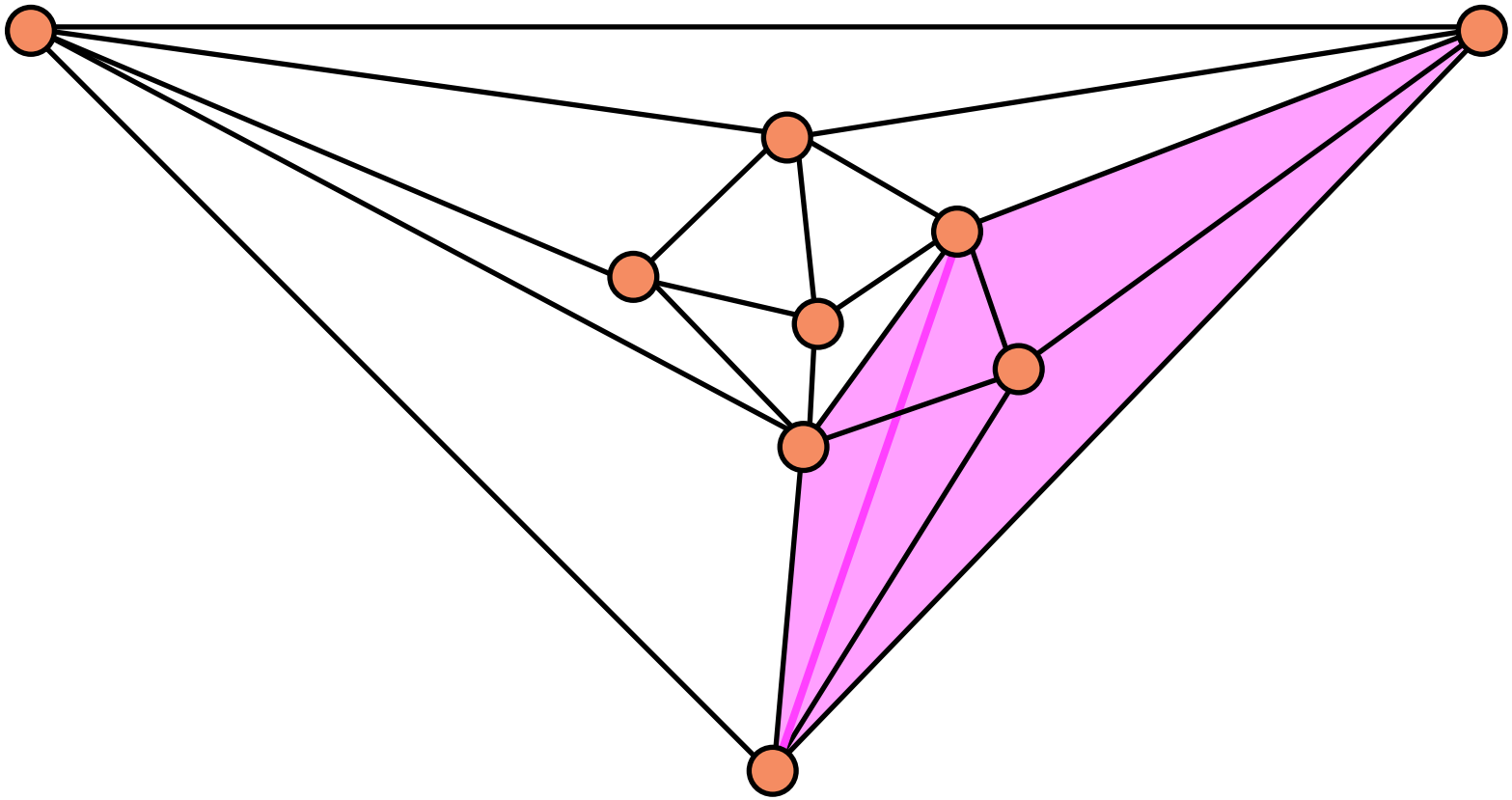
Inkrementeller Algorithmus



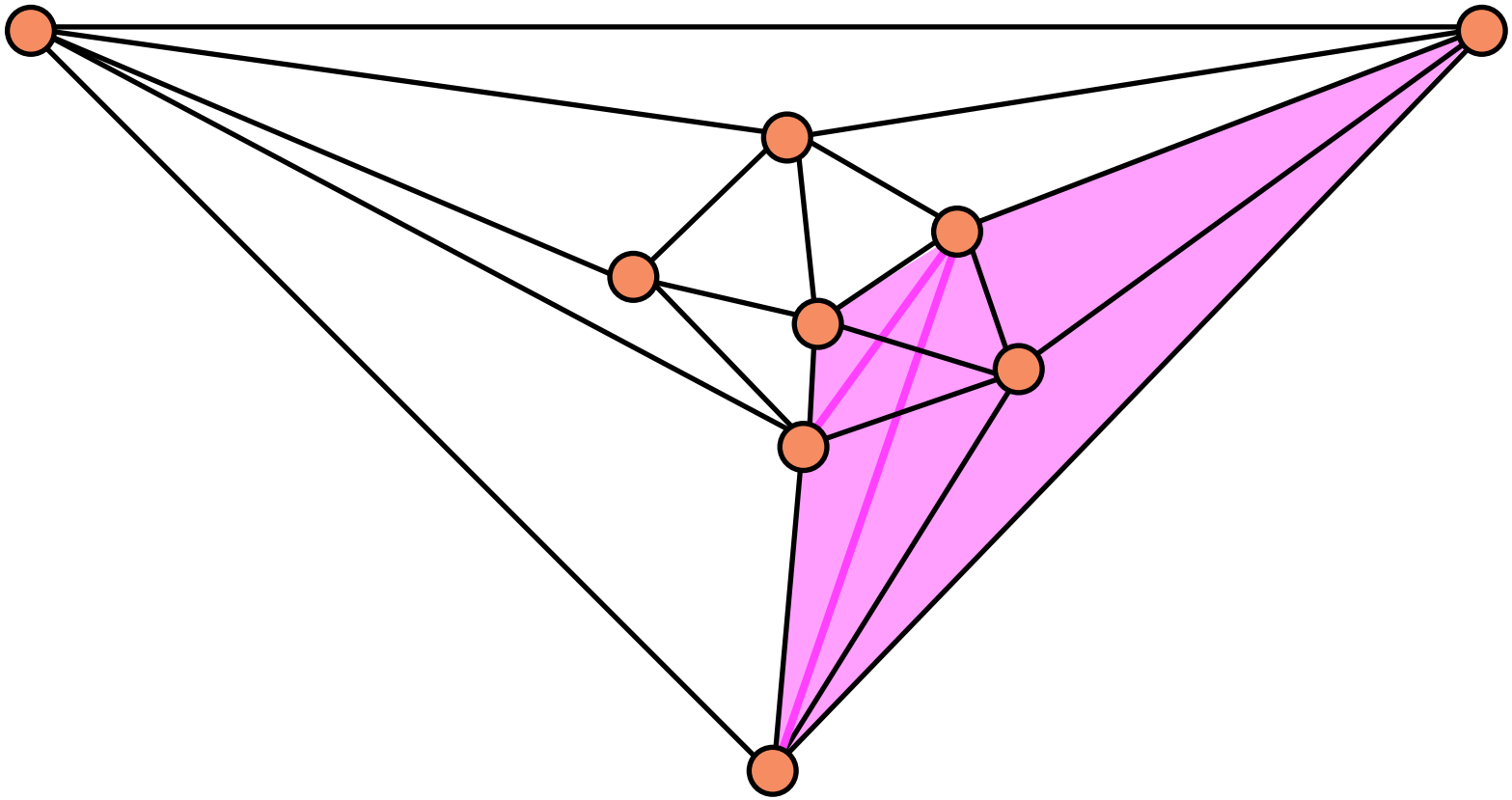
Inkrementeller Algorithmus



Inkrementeller Algorithmus



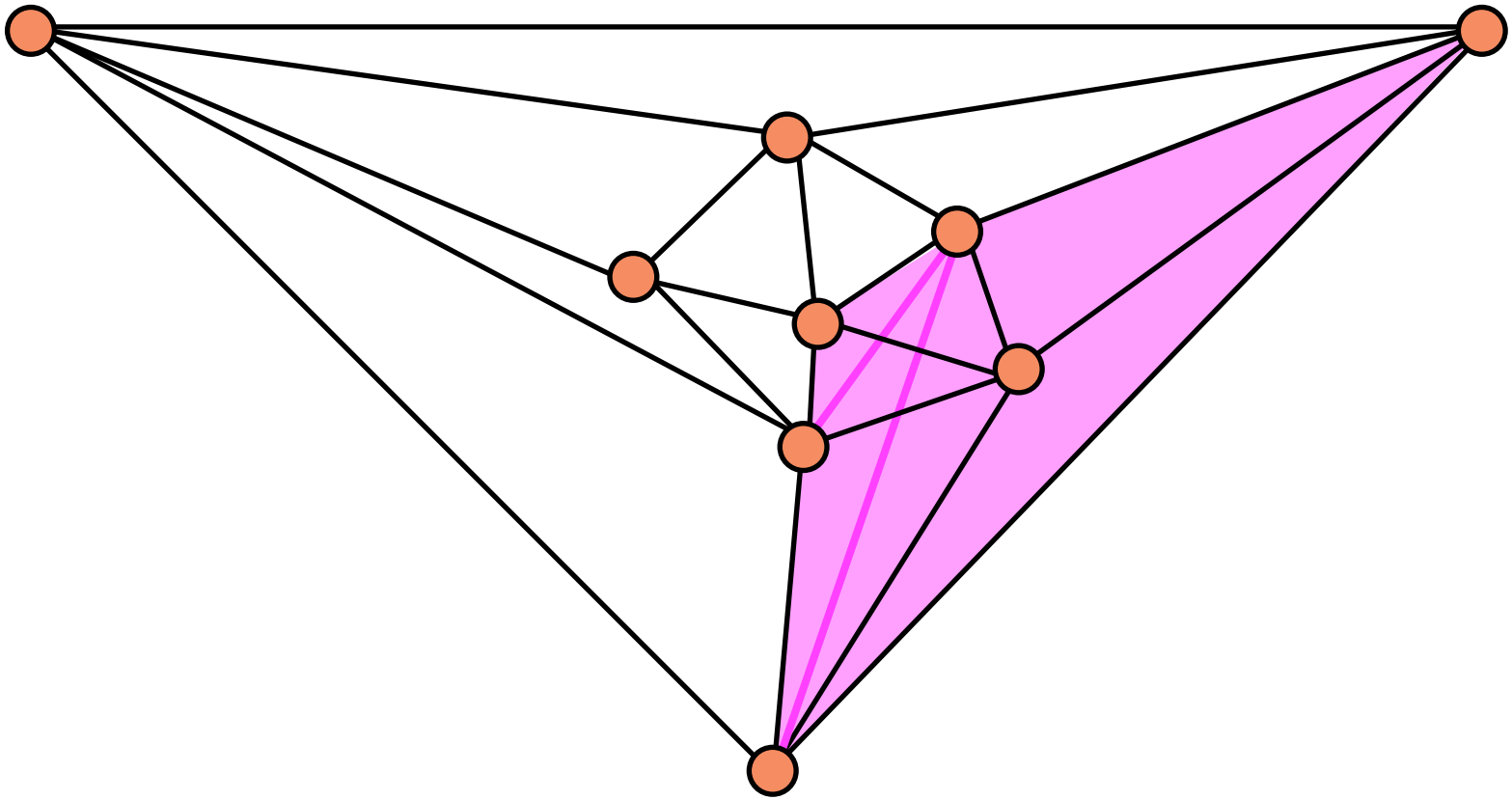
Inkrementeller Algorithmus



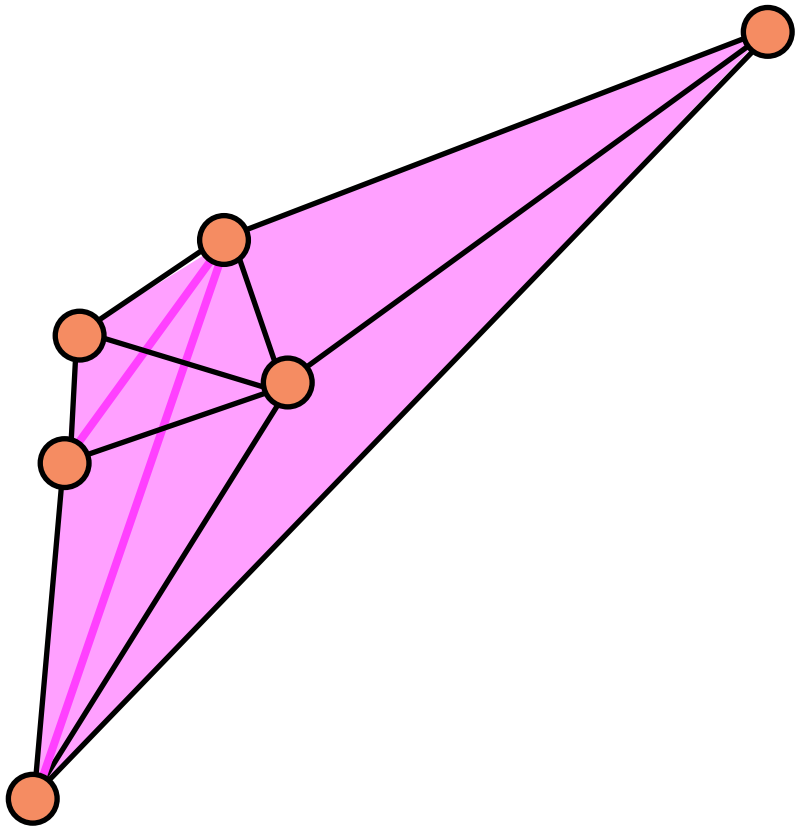
Beschleunigte Suche

- Am Anfang existiert nur ein Dreieck
- Bei jedem Einfügeschritt werden m Dreiecke entfernt und $m+2$ Dreiecke ergänzt
- $m+2 \leqslant k$ … die maximale Valenz
- $m+2 : m \geqslant c > 1$ … minimaler Verfeinerungsfaktor

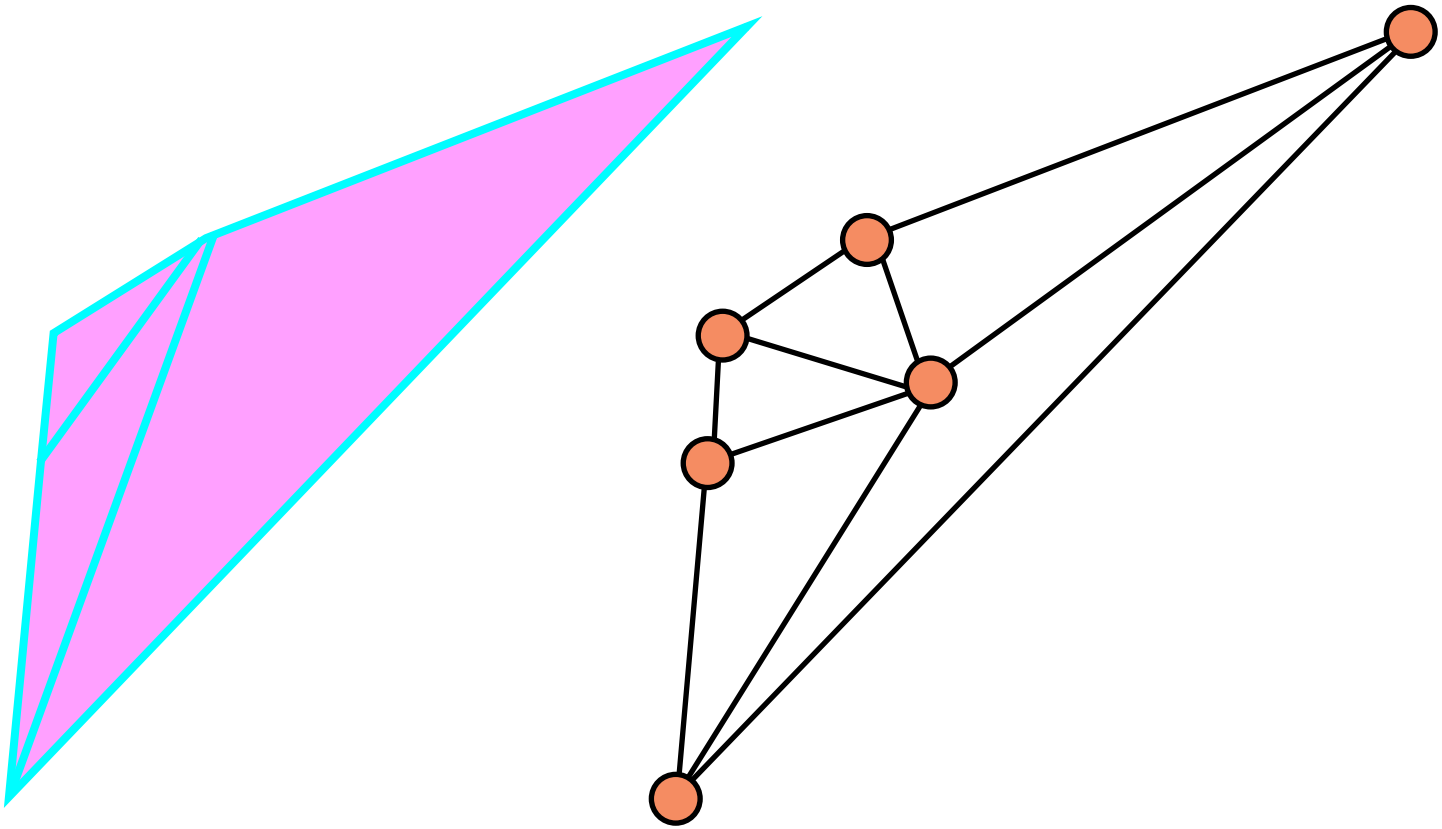
Inkrementeller Algorithmus



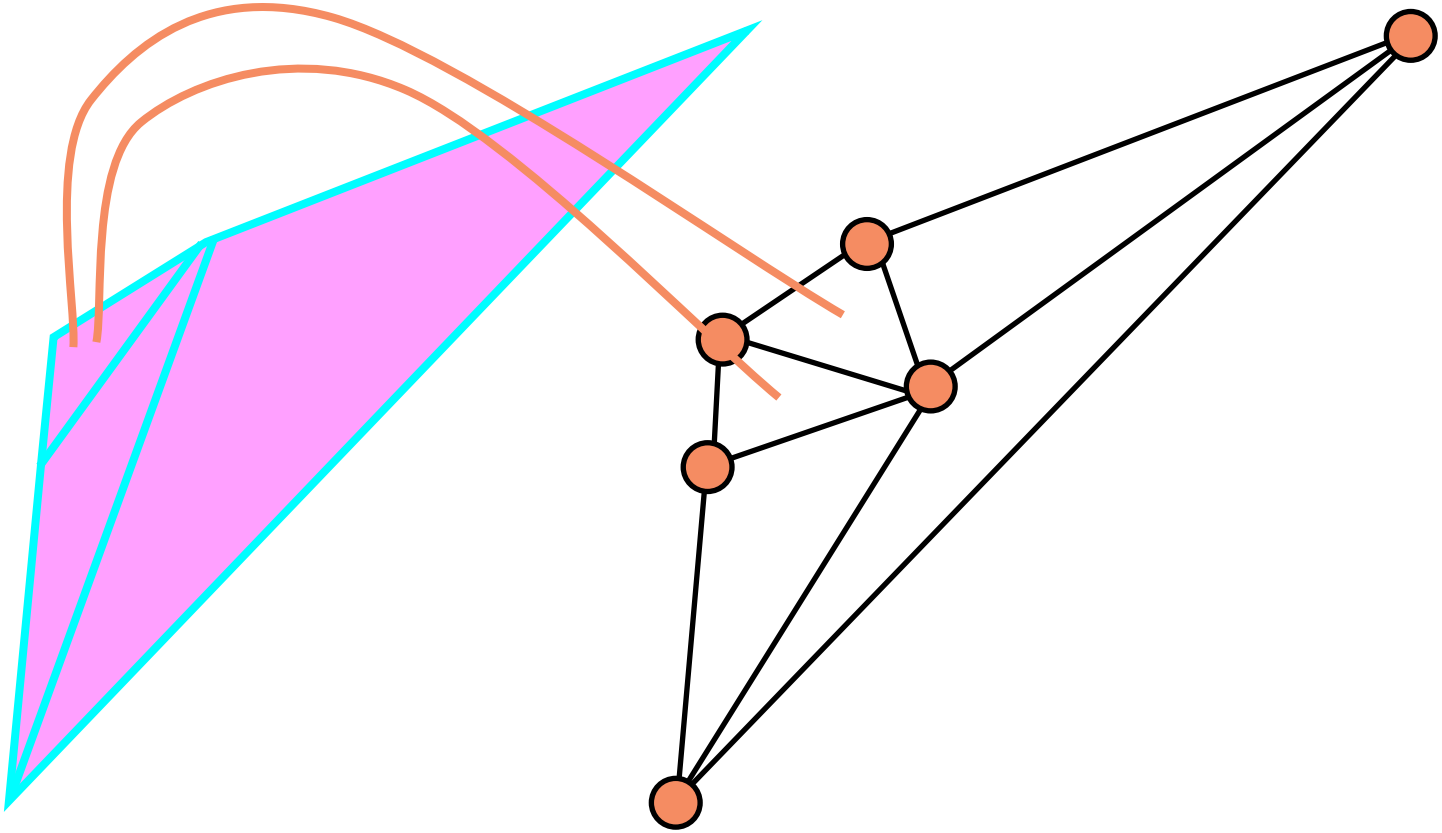
Inkrementeller Algorithmus



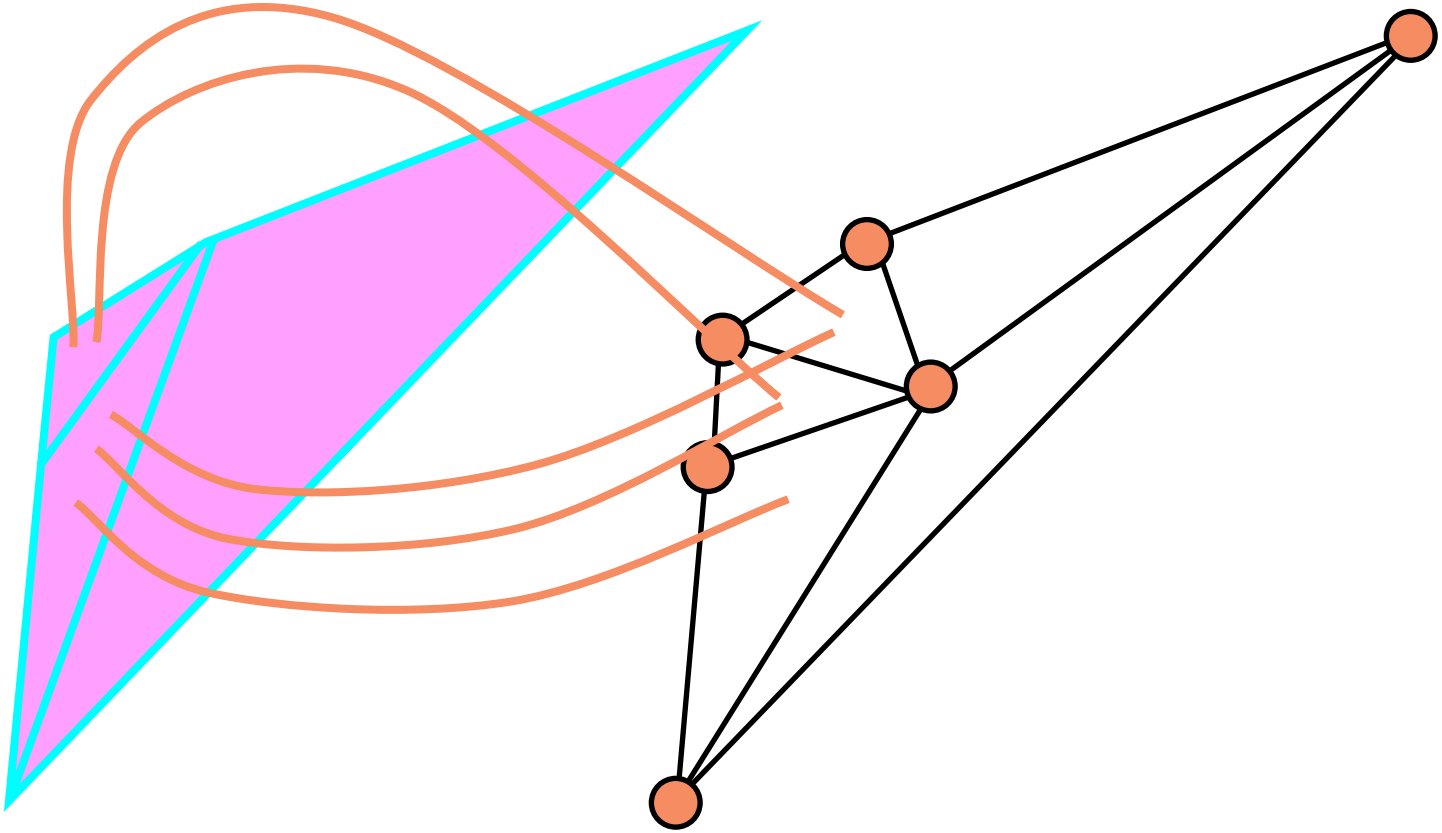
Inkrementeller Algorithmus



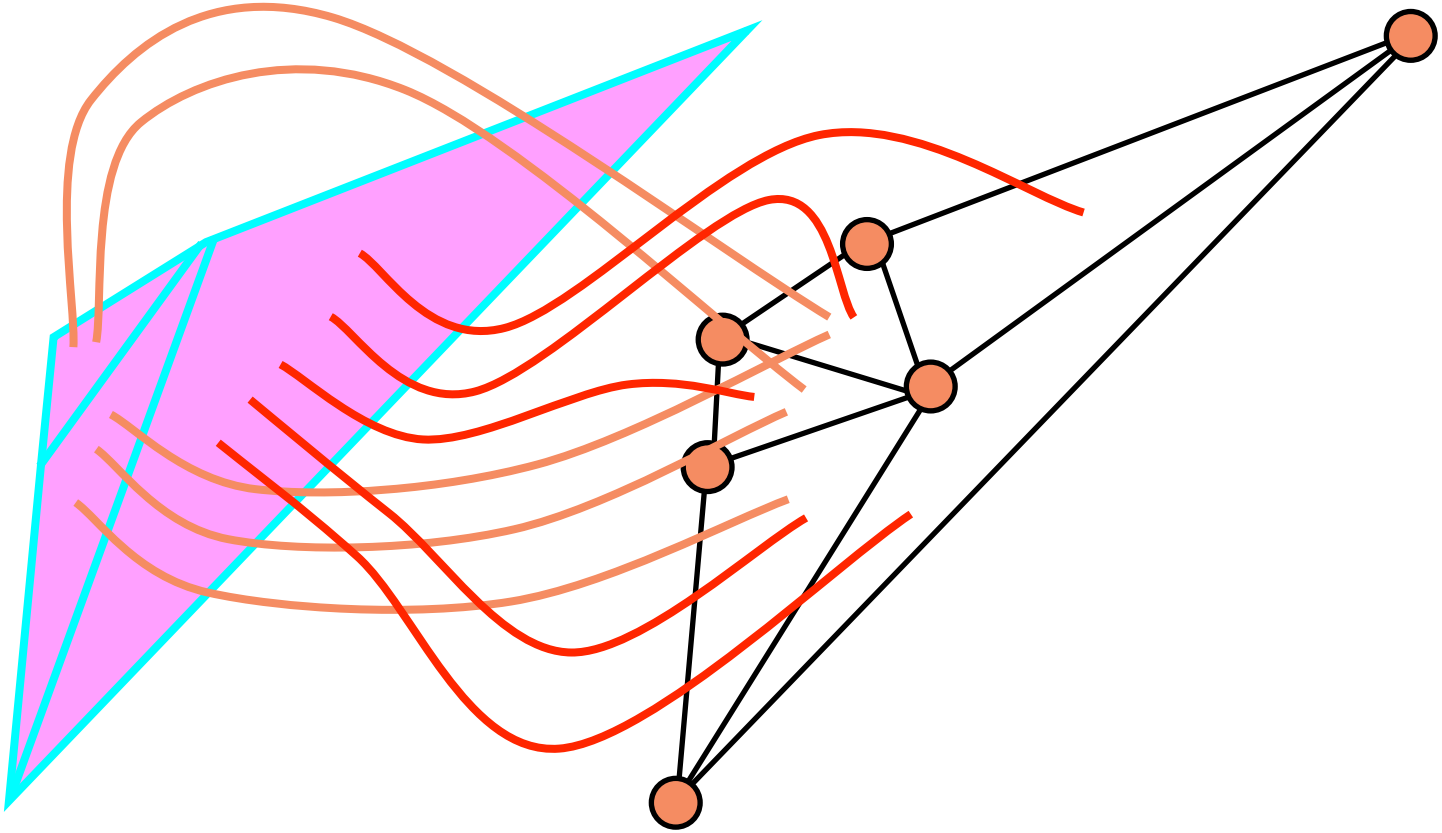
Inkrementeller Algorithmus



Inkrementeller Algorithmus



Inkrementeller Algorithmus



Suchbaum für Dreiecke

- Seien die Dreiecke, die im Laufe des Algorithmus generiert werden die Knoten des Suchbaumes
- Durch den minimalen Verfeinerungsfaktor $m+2 : m \geqslant c > 1$ ist garantiert, dass die Anzahl von Knoten von Level zu Level exponentiell steigt

Suchbaum für Dreiecke

- Obwohl diese Datenstruktur keinen (zyklenfreien) Baum darstellt, ergibt sich trotzdem ein gerichteter Graph, dessen Pfade maximal logarithmische Länge haben, wenn die Punkte hinreichend gleichmäßig verteilt sind (und die maximale Valenz beschränkt ist).
- Kosten:
  - Aufbau des Baumes $O(n)$
  - Suche im Baum $O(\log n)$

Inkrementeller Algorithmus

- Für jeden Punkt p_i — O(n)
  - Suche das entsprechende Dreieck — O(log n)
  - Füge den neuen Punkt ein — O(1)
  - Flippe Kanten bis die Delaunay-Bedingungen wieder hergestellt sind. — O(k)
  - Gesamtaufwand: O(n log n)