Geometrie II
"Hallo Welt!" für Fortgeschrittene - 2010

Thorsten Wißmann

2. Juli 2010

Stand: 1. Juli 2010

Inhalt

Inhalt

Datentypen

Punkt im $\mathbb{R}^2$:

```
1 class Point {
2 public:
3     double x;
4     double y;
5 };
```

Rechteck im $\mathbb{R}^2$:

```
1 class Rect {
2 public:
3     Point bl; //bottom left
4     Point tr; //top right
5 };
```

Datentypen

Knoten eines Binärbaums:

```
1 class Tree {
2 public:
3     Point p;
4     Tree* left;
5     Tree* right;
6 };
```

Datentypen

Element einer einfach verketten Liste von Punkten:

```
1 class List {
2 public:
3     Point p;
4     List* next;
5 };
```

Inhalt

Bereichssuche

Definition: Berreichssuche

Gegeben:

- Menge von Punkten im k-dimensionalen Raum
- Zu jeder Dimension i ein Intervall, d.h. Bereich

Gesucht:

- Teilmenge an Punkten, bei der die i-te Koordinate im i-ten Intervall liegt.

Bereichssuche

Definition: Berreichssuche

Gegeben:

- Menge von Punkten im k-dimensionalen Raum
- Zu jeder Dimension i ein Intervall, d.h. Bereich

Gesucht:

- Teilmenge an Punkten, bei der die i-te Koordinate im i-ten Intervall liegt.

Beispiel: 2D

- Punkte $A, B, C, D, E \in \mathbb{R}^2$
- $x \in [2; 5] \wedge y \in [4; 7]$ entspricht Rechteck $(2, 4)(5, 7)$

Bereichssuche

Naiver Ansatz

Über alle Punkte iterieren und die zur Ergebnismenge hinzufügen, die im gesuchten Bereich liegen.

Bereichssuche

Naiver Ansatz

Über alle Punkte iterieren und die zur Ergebnismenge hinzufügen, die im gesuchten Bereich liegen.
$\Rightarrow$ Laufzeit: $\mathcal{O}(|\text{Punkte}|)$

Inhalt

Gitterverfahren

Idee

Im Vorfeld:

- Suchraum gitterartig in Quadrate zerlegen
- jedes Quadrat entspricht Menge von Punkten
- im Vorfeld alle Punkte in Quadrate einsortieren

Bei Bereichssuche:

- nur noch Quadrate im Bereich durchsuchen

Gitterverfahren

Punkt in das Gitter einfügen:

```
 1 double width; // Breite eines Gitterquadrates
 2 List* grid[GRIDSIZE][GRIDSIZE];
 3
 4 void insert_point(Point p) {
 5     List* el = create_list(p);
 6     int gx = p.x / width;
 7     int gy = p.y / width;
 8     el->next = grid[gx][gy];
 9     grid[gx][gy] = el->next;
10 }
```

Gitterverfahren

Punkte im Bereich finden:

```
 1 List* bereichssuche(Rect r) {
 2     List* erg = empty_list();
 3     int f_x = r.bl.x/width, t_x = r.tr.x/width;
 4     int f_y = r.bl.y/width, t_y = r.tr.y/width;
 5     for (int x = f_x; x <= t_x; x++)
 6         for (int y = f_y; y <= t_y; y++) {
 7             List* el = grid[x][y];
 8             while (el != NULL)
 9                 if (is_in_rect(r, el->p))
10                     append_list(erg, el->p);
11         }
12     return erg;
13 }
```

Gitterverfahren

Nachteil

Gittergröße und -auflösung muss vorher bekannt sein

Gitterverfahren

Nachteil

Gittergröße und -auflösung muss vorher bekannt sein

Nachteil

Schlechte Laufzeit bei:

- zu vielen Quadraten $\Rightarrow$ viele leere Quadrate
- zu wenigen Quadraten $\Rightarrow$ zu viele Punkte pro Quadrat

Gitterverfahren

Nachteil

Gittergröße und -auflösung muss vorher bekannt sein

Nachteil

Schlechte Laufzeit bei:

- zu vielen Quadraten $\Rightarrow$ viele leere Quadrate
- zu wenigen Quadraten $\Rightarrow$ zu viele Punkte pro Quadrat

$\Rightarrow$ keine eindeutige minimale Laufzeitkomplexität in $\mathcal{O}$-Notation

Inhalt

kD-Trees

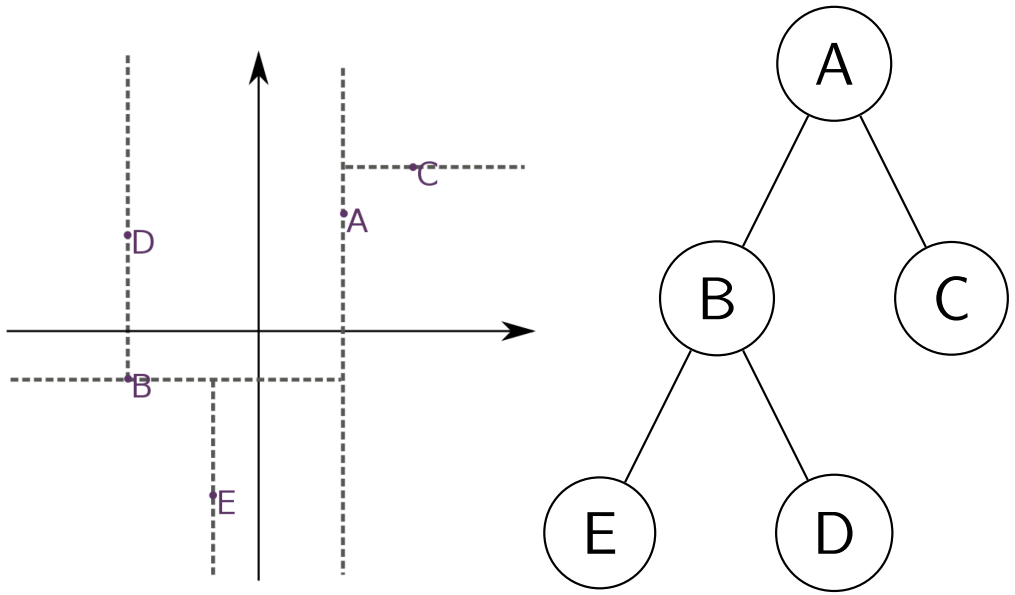
Idee

Punkte in Binären Suchbaum einsortieren.
Pro Ebene andere Dimension als Vergleichswert nutzen.

kD-Trees

Idee

Punkte in Binären Suchbaum einsortieren.
Pro Ebene andere Dimension als Vergleichswert nutzen.



kD-Trees

k-Dimensionalen Punkt in Baum einfügen:

```
 1 void tree_insert(Point p, Tree* t, int d)
 2 {
 3     Tree* n;
 4     bool lower = p[d] < t->p[d];
 5     if (lower) n = t->l;
 6     else       n = t->r;
 7     if (n != NULL) {
 8         tree_insert(p, n, (d+1) % k);
 9     } else {
10         if (lower) t->l = create_node(p);
11         else       t->r = create_node(p);
12     }
13 }
```

kD-Trees

Problem

Bei ungünstiger Einfügreihenfolge $\Rightarrow$ Entartung des Baums (vgl. Binärbaum)

kD-Trees

Problem

Bei ungünstiger Einfügreihenfolge $\Rightarrow$ Entartung des Baums (vgl. Binärbaum)

Lösung

jeweils Untermedian einfügen

Bereichssuche: Punkte innerhalb des k-Dimensionalen Rechtecks:

```
 1 List* bs(Rect r, Tree* t, int d)
 2 {
 3     if (t == NULL) return empty_list();
 4     List* erg = empty_list();
 5     bool lower = r.p1[d] < t->p[d];
 6     bool upper = t->p[d] <= r.p2[d];
 7     if (lower)
 8         append_list(erg, bs(r, t->l, (d+1)%k));
 9     if (is_in_rect(r, t->p))
10         append_list(erg, t->p);
11     if (upper)
12         append_list(erg, bs(r, t->r, (d+1)%k));
13     return erg;
14 }
```

Inhalt

Closest Pair

Definition: Closest Pair

Gegeben: Menge von Punkten M

Gesucht: das Punktepaar $P1, P2 \in M$ mit dem kleinsten Abstand

Closest Pair

Naiver Ansatz

Über alle Punktepaare iterieren und mit bisher bekanntem Closest Pair vergleichen.
Wenn Abstand kleiner, dann Closest Pair auf aktuelles Punktepaar setzen.

Closest Pair

Naiver Ansatz

Über alle Punktepaare iterieren und mit bisher bekanntem Closest Pair vergleichen.
Wenn Abstand kleiner, dann Closest Pair auf aktuelles Punktepaar setzen.
$\Rightarrow$ Laufzeit: $\mathcal{O}(|M|^2)$

Closest Pair

Bessere Lösung

Teile und Herrsche: (vgl. Merge-Sort)

- Problem in kleinere Teilprobleme zerlegen
- Teilprobleme (rekursiv) lösen
- Lösungen der Teilprobleme zusammenfügen (mergen)

Closest Pair

Bessere Lösung

Teile und Herrsche: (vgl. Merge-Sort)

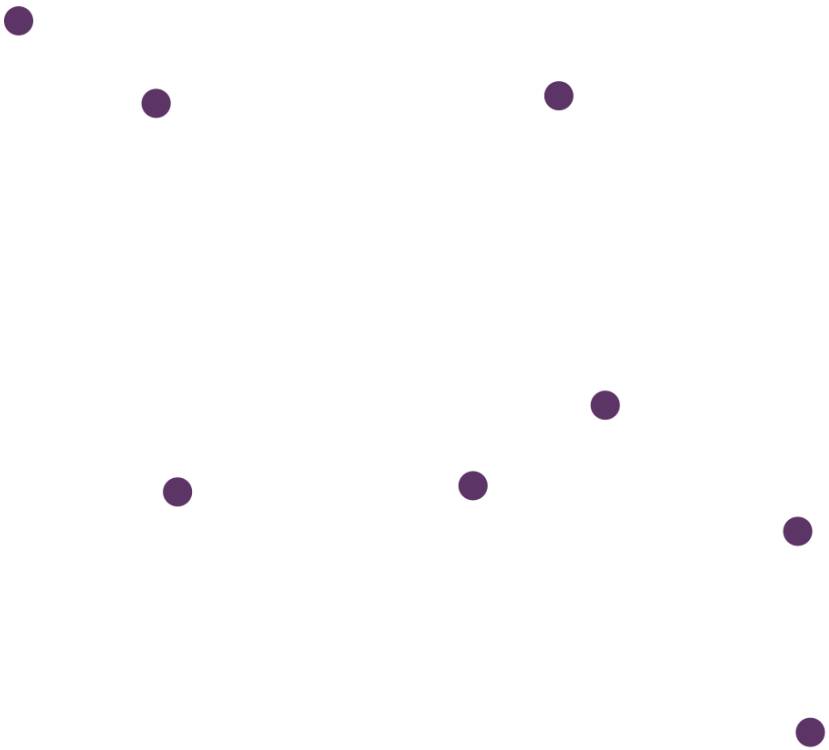
- Problem in kleinere Teilprobleme zerlegen
- Teilprobleme (rekursiv) lösen
- Lösungen der Teilprobleme zusammenfügen (mergen)

Bei Closest Pair

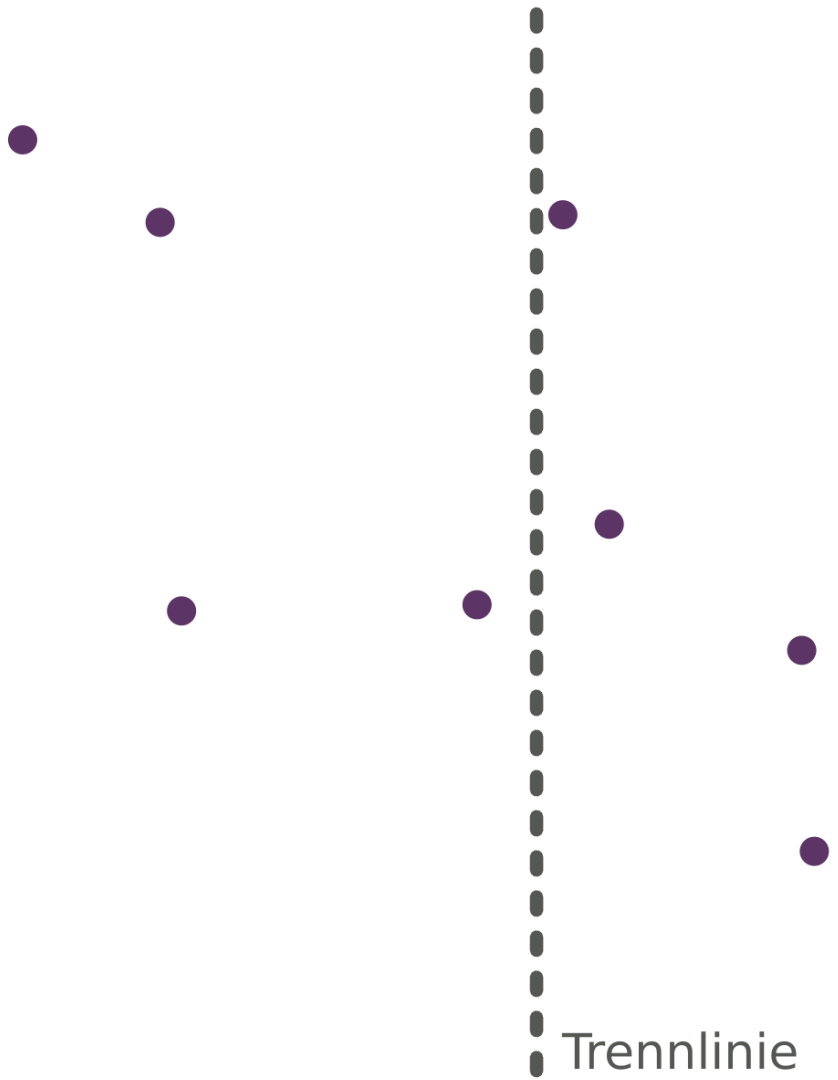
Nach x-Koordinate vorsortieren

- Punktebereich nach x-Koordinate in der Mitte zerteilen
- Closest Pair (rekursiv) in linker und rechter Hälfte finden
- Closest Pair in Grenzbereich finden
- Endergebnis ist minimales Closest Pair.

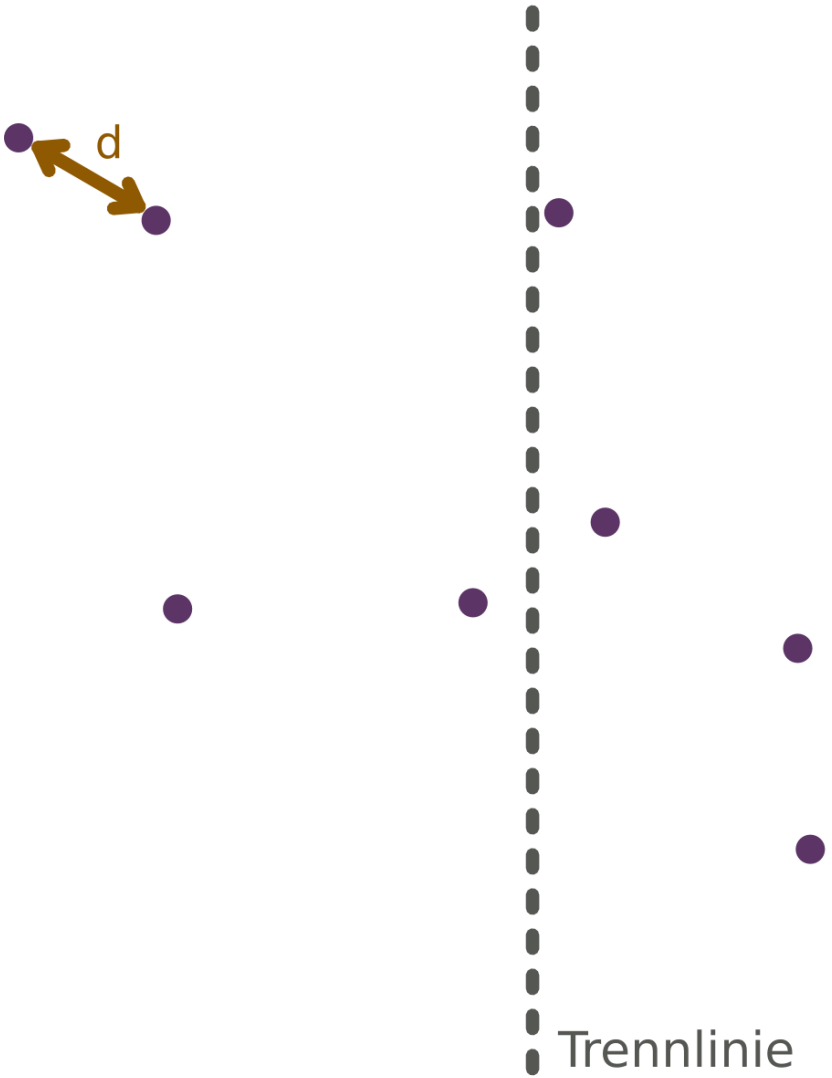
Closest Pair



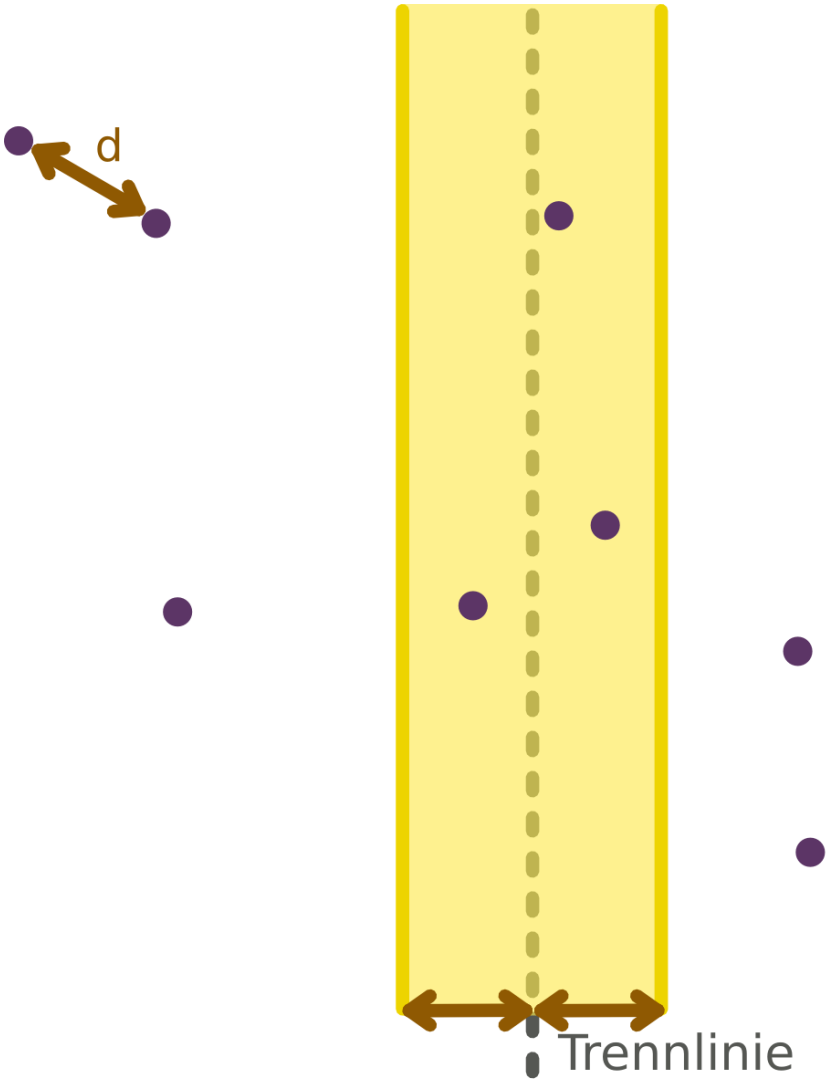
Closest Pair



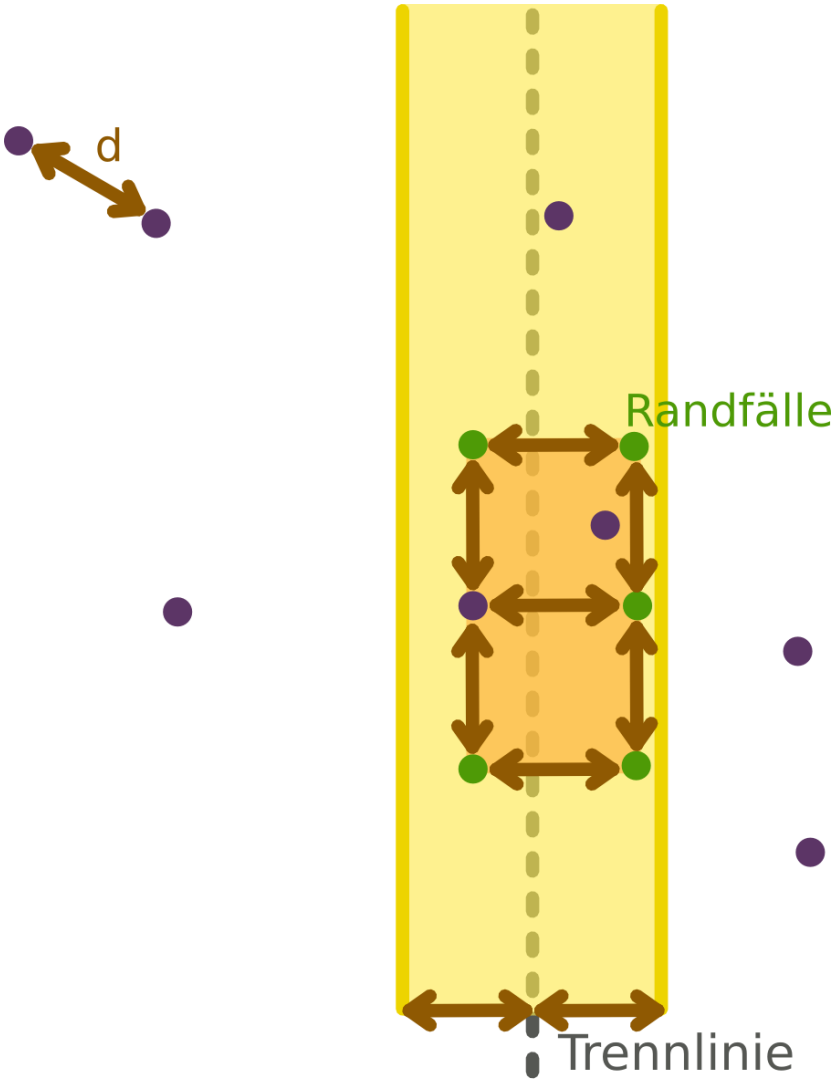
Closest Pair



Closest Pair



Closest Pair



Closest Pair

Implementierung für $M \subset \mathbb{R}^2$

- Punkte in einfach verlinkter Liste
- `check(P1, P2)`: bei Bedarf CP neu setzen
- `merge(L1, L2)`: sortierte Listen mergen (vgl. Merge-Sort)
- `cp(l, N)`: CP in den ersten `N` Elementen der Liste `L`

Closest Pair

Implementierung für $M \subset \mathbb{R}^2$

- Punkte in einfach verlinkter Liste
- `check(P1, P2)`: bei Bedarf CP neu setzen
- `merge(L1, L2)`: sortierte Listen mergen (vgl. Merge-Sort)
- `cp(l, N)`: CP in den ersten `N` Elementen der Liste `L`

Sortieren

- Anfangs nach x-Koordinate sortieren
- Beim Verlassen der Rekursion schrittweise nach y sortieren

Closest Pair

Bisheriges Closest Pair bei Bedarf neu setzen

```
 1 Point cp1, cp2;
 2 double mindist;
 3
 4 void check(Point* p1, Point* p2)
 5 {
 6     if (p1 == NULL || p2 == NULL) return;
 7     double dist = distance(p1, p2);
 8     if (dist < mindist) {
 9         cp1 = p1;
10         cp2 = p2;
11         mindist = dist;
12     }
13 }
```

Closest Pair

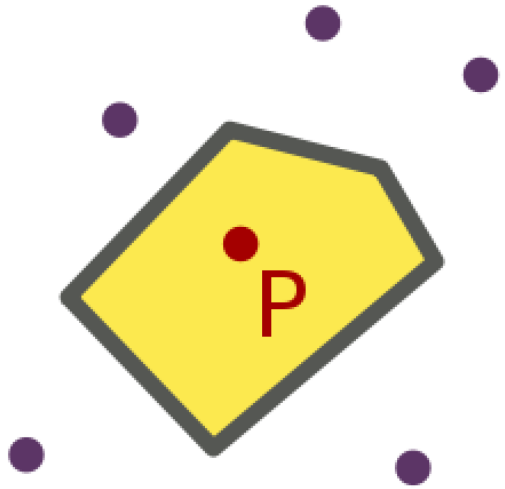
```
 1 List* cp(List* l, int N)
 2 {
 3     if (l == NULL || l->next == NULL) return l;
 4     List* part1 = l; List* part2;
 5     for (int i = 0; i < N/2; i++) l = l->next;
 6     part2 = l->next; l->next = NULL;
 7     double middle = part2->p.x;
 8     l = merge(cp(part1, N/2), cp(part2, N-N/2));
 9     List* cur = l;
10     Point *p1, *p2, *p3, *p4; p1 = p2 = p3 = p4 = NULL;
11     while (cur != NULL) {
12         if (abs(cur->p.x - middle) < mindist) {
13             check(&cur->p, p1);
14             check(&cur->p, p2);
15             check(&cur->p, p3);
16             check(&cur->p, p4);
17             p1 = p2; p2 = p3; p3 = p4; p4 = &cur->p;
18         }
19         cur = cur->next;
20     }
21     return l;
22 }
```

Inhalt

Voronoi-Diagramme

Definition: Voronoi-Polygon

Gegeben sei eine Menge an Punkten M.
Voronoi-Polygon um $P \in M$:
Randlinie der Menge aller Punkte x, deren Abstand zu P kleiner als zu allen anderen Punkten



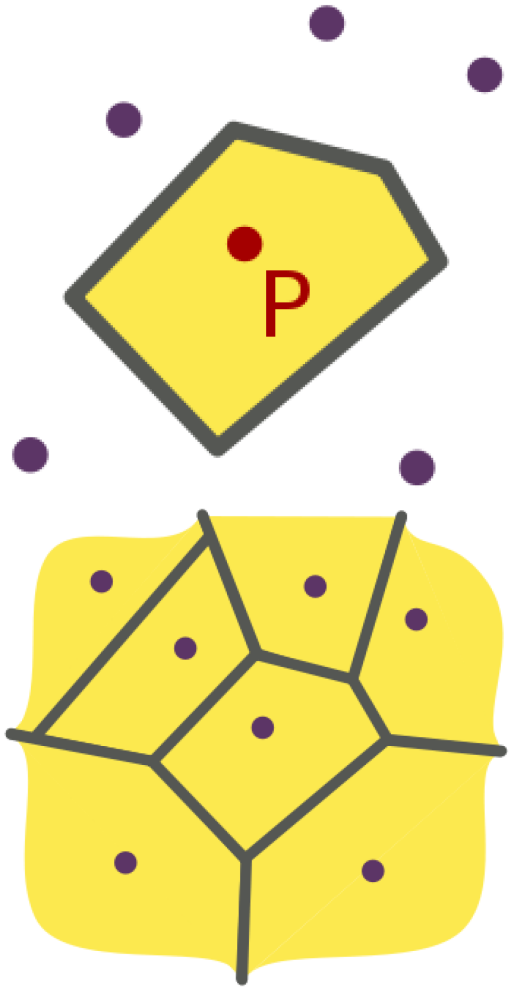
Voronoi-Diagramme

Definition: Voronoi-Polygon

Gegeben sei eine Menge an Punkten M.
Voronoi-Polygon um $P \in M$:
Randlinie der Menge aller Punkte x, deren Abstand zu P kleiner als zu allen anderen Punkten

Definition: Voronoi-Diagramm

Menge aller Voronoi-Polygone zu allen $P \in M$.



Konstruktion: Sweep-Line-Algorithmus

Allgemeine Eigenschaften:

- Events (d.h. Punkte) in Ebene in Warteschlange Q
- Horizontale Gerade (Sweep Line) wandert von oben nach unten
- Wenn Gerade Event überquert, wird Event ausgeloest
- Status τ: Liste von zuletzt ausgeführter Events

Voronoi: Typen von Events

- Site-Event:
  - neuer Punkt auf Sweep-Line
  - neuer Punkt in τ
- Circle-Event
  - neuer Knotenpunkt des Voronoi-Diagramms bestimmbar
  - zugehöriger Punkt nicht mehr zu betrachten (aus τ zu entfernen)

Voronoi-Diagramme

Fortune's algorithm

Eingabe: Punktemenge P

Ausgabe: Voronoi Diagramm

- $Q := \{\text{Alle site-events}\}$, $\tau =$ leer
- solange Q nicht leer
  - $p :=$ entferne Event mit höchster y-Koordinate aus Q
  - falls Site-Event $\Rightarrow$ HandleSiteEvent(p)
  - sonst $\Rightarrow$ HandleCircleEvent(γ), mit $\gamma :=$ zu p zugehöriger Knoten in τ
- restliche Knoten in τ entsprechen Halbgeraden am Rand

Fortune's algorithm

HandleSiteEvent(p)

- falls τ leer $\Rightarrow$ p in τ einfügen und returnen
- suche in τ nach p' direkt über p und entferne p' und zugehöriges CircleEvent aus Q falls vorhanden
- füge p an Stelle von p' ein
- füge Knoten in Voronoi-Diagramm ein, welcher p und p' trennt
- erstelle CircleEvents für p und dessen Nachbarn
- füge CircleEvents in Q ein

Fortune's algorithm

HandleCircleEvent(γ)

- entferne γ aus τ,
- entferne CircleEvents, bei denen γ beteiligt ist
- füge Mittelpunkt des Kreises als Knoten in Q hinzu
- füge Kanten zu bereits bekannten Nachbarknoten hinzu
- füge neue CircleEvents für neue Nachbarn zu Q hinzu

Voronoi-Diagramme

Anmerkungen:

Implementierungsdetails:

- Q ist priority queue nach y sortiert
- τ ist Binärer Suchbaum nach x sortiert
- τ enthält Pointer auf zugehöriges CircleEvent in Q
- CircleEvent zeigt auf zugehörigen Knoten in τ
- Halbgeraden durch Strecken darstellen und Diagramm durch Rechteck abgrenzen (Bounding Box)
- Voronoi-Diagramm als doppelt-verlinkte Kantenliste

Literatur


Robert Sedgewick: Algorithmen (2. Auflage)
Pearson Studium 2002


Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry
Springer, Berlin; Auflage: 3rd ed. (7. März 2008)


Applet zu Fortune's Algorithmus
http://www.diku.dk/hjemmesider/studerende/duff/Fortune/