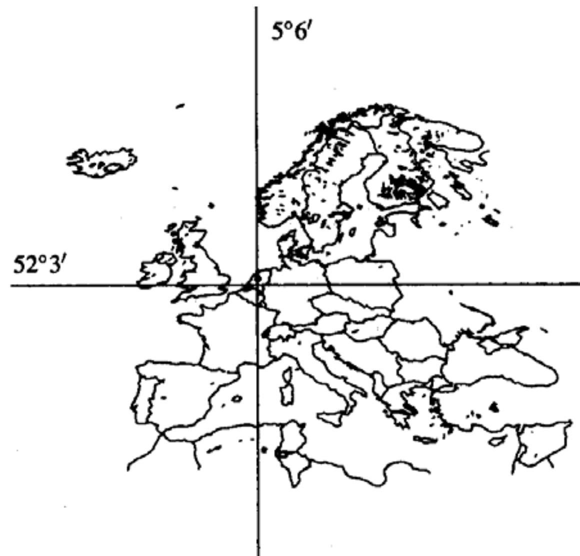
Geometrische Algorithmen

Geometrische Algorithmen



Geometrische Algorithmen

Geometrische Problemstellungen

- *Computational Geometry* als Spezialgebiet der Informatik/Mathematik beschäftigt sich mit der Lösung von geometrischen Problemstellungen mittels des Computers
- Typische Anwendungsbereiche
  - CAD
  - VLSI-Schaltungen
  - Geographie
  - Virtuelle Welten (Virtual Reality)
  - alle weiteren Anwendungen, bei denen die Form von physikalischen Objekten eine Rolle spielt
- Schwierigkeiten bei geometrischen Problemstellungen
  - einfachste Dinge, wie z.B. Enthaltensein eines Punktes in einem Polygon, müssen auf Berechnungen zurückgeführt werden
  - viele Sonderfälle sind zu berücksichtigen
  - numerische Rechengenauigkeit

Geometrische Algorithmen

Inhalt

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Geometrische Algorithmen
Geometrische Figuren

ADT Point

- Allgemeine ADT `Point` ist durch x- und y-Koordinate charakterisiert
  - `double getX()` - x-Koordinate des Punktes
  - `double getY()` - y-Koordinate des Punktes
  - `double getAngle()` - Winkel des Punktes bei Polarkoordinatendarstellung
  - `double getDistance()` - Distanz des Punktes bei Polarkoordinatendarstellung
- Im folgenden wollen wir der Einfachheit halber aber annehmen, dass
  - Punkte immer x, y-Koordinaten dargestellt sind
  - Koordinaten `int`-Werte sind
  - direkt auf die Variablen x und y zugriffen wird
- Damit ist ein ADT `Point` gegeben durch
  - `int x` - x-Koordinate des Punktes
  - `int y` - y-Koordinate des Punktes

ADT Line

- ADT `Line` ist durch zwei Punkte festgelegt
  - `Point getP1()` - erster Punkt der Linie
  - `Point getP2()` - zweiter Punkt der Linie
  - `double length()` - Länge der Linie
- der Einfachheit halber werden wir im Folgenden bei `Line` direkt mit den Variablen p1 und p2 arbeiten
  - `Point p1` - erster Punkt der Linie
  - `Point p2` - zweiter Punkt der Linie

ADT Polygon, Triangle, Rectangle

Geometrische Algorithmen
Geometrische Figuren

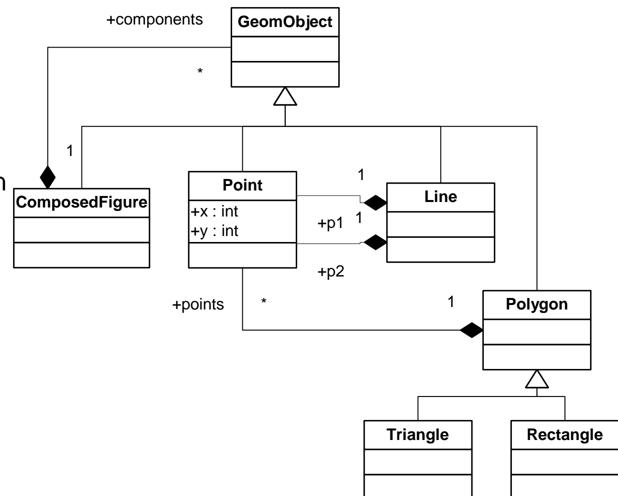
- ADT `Polygon` ist durch eine Reihe von Punkten dargestellt
  - `Point[] points()` - liefert die Eckpunkte des Polygons
  - `Line[] lines()` - liefert die Seitenlinien des Polygons
  - `double area()` - Fläche des Polygons
- ADT `Triangle` erweitert `Polygon` und definiert Zugriff auf die drei Punkte und drei Seiten
  - `Point getP1()` - liefert den 1. Eckpunkt des Dreiecks
  - `Point getP2()` - liefert den 2. Eckpunkt des Dreiecks
  - `Point getP3()` - liefert den 3. Eckpunkt des Dreiecks
  - `Line getLine1()` - liefert die 1. Seitenlinie des Dreiecks
  - `Line getLine2()` - liefert die 2. Seitenlinie des Dreiecks
  - `Line getLine3()` - liefert die 3. Seitenlinie des Dreiecks
  - ...
- ADT `Rectangle` erweitert `Polygon` und und definiert Zugriff auf die vier Punkte oder vier Seiten
  - `Point getP(int i)` - liefert den i-ten Eckpunkt des Rechtecks
  - `Line getLine(int i)` - liefert die i-te Seitenlinie des Rechtecks
  - `double getWidth()` - liefert die Breite des Rechtecks
  - `double getHeight()` - liefert die Höhe des Rechtecks
  - ...

ADT ComposedFigure

Aus elementaren Figures lassen sich durch einfache Komposition zusammengesetzte Figuren bilden

Composite Pattern

- `GeomObject` bildet abstrakte Basisklasse für geometrische Figuren
- davon abgeleitet sind die unterschiedlichen Objekttypen wie `Point`, `Line`, `Polygon` etc.
- `ComposedFigure` ist ebenso von `GeomObject` abgeleitet
- `ComposedFigure` hat eine Reihe von *beliebigen* `GeomObject` als Komponenten (d.h. Komponenten können auch vom Typ `ComposedFigure` sein)



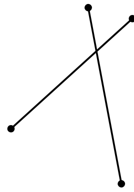
- ADT `ComposedFigure` ist durch die Subkomponenten dargestellt
  - `GeomObject[] components()` – liefert die geometrischen Objekte aus dem die zusammengesetzte Figur besteht

Inhalt

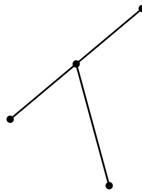
- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Schnitt von Strecken

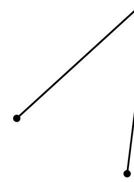
- Frage, ob sich zwei Stecken schneiden



schneiden sich



schneiden sich



schneiden sich nicht

- Lösung 1: Schnittpunkt der Geraden berechnen und dann testen, ob dieser auf beiden Strecken liegt

  ==> aufwendig!!

- Lösung 2: Ohne Schnittpunkt und unter Verwendung einer Hilfsfunktion, die für 3 Punkte bestimmt, ob sie im Uhrzeigersinn oder gegen den Urzeigersinn angeordnet sind

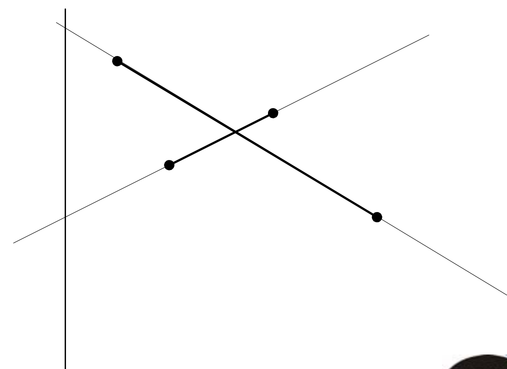
Lösung 1: Schnittpunkt der Geraden

- Geradengleichungen y = k*x +d für beide Strecken aufstellen
- Schnittpunkt der Geraden berechnen
- Prüfen, ob Schnittpunkt innerhalb beider Strecken liegt

Beispiel:

Strecke1 gegeben durch Punkte (2, 4) und (4, 5)
Strecke2 gegeben durch Punkte (1, 5) und (6, 3)

Berechnung des Schnittes ==
Übungsaufgabe !



Lösung 2: Funktion ccw (counter clockwise)

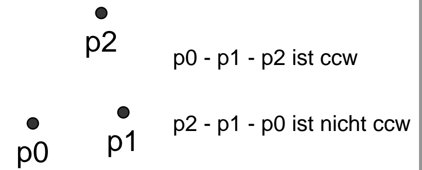
Geometrische Algorithmen — Schnitt von Strecken

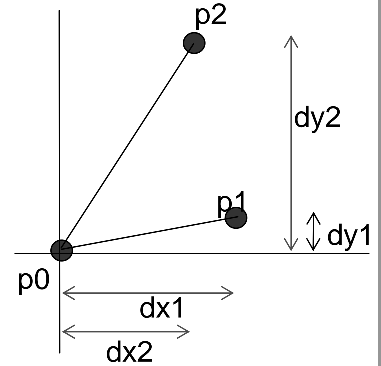
- Methode `ccw` bestimmt, ob drei Punkte gegen den Uhrzeigersinn angeordnet sind
- Verfahren (am Beispiel der Anordnung p0 - p1 -p2):
  - Man betrachtet die Geraden von p0 zu p1 und p0 zu p2
  - Dann vergleicht man die Steigungen dy/dx der beiden Geraden
    - ist die Steigung von p0 zu p1 kleiner als von p0 zu p2 dann ccw
    - sonst nicht ccw

```
dx1 = p1.x - p0.x
dy1 = p1.y - p0.y
dx2 = p2.x - p0.x
dy2 = p2.y - p0.y
if (dy1/dx1 < dy2/dx2) then ccw
if (dy1/dx1 > dy2/dx2) then not ccw
```

  - Durch Multiplikation der Ungleichung mit dx1*dx2 erhält man die Ungleichung

```
if (dy1*dx2 < dy2*dx1) then ccw
if (dy1*dx2 > dy2*dx1) then not ccw
```

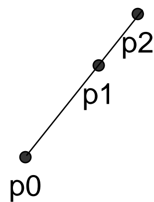




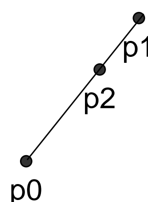
Funktion ccw: Sonderfall kollinear Punkte

Geometrische Algorithmen — Schnitt von Strecken

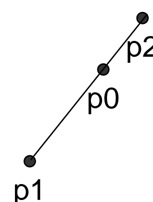
- Ein Sonderfall ergibt sich, wenn die drei Punkte auf einer Geraden liegen
- Die Funktion `ccw` wird dreiwertig (-1, 0, 1) mit
  - 1 bedeutet ccw
  - -1 bedeutet cw
  - und folgender Annahmen bei kollinearen Punkten



p1 zwischen p0 und p2 = 1



p2 zwischen p0 und p1 = 0



p0 zwischen p1 und p2 = -1

Algorithmus ccw

Geometrische Algorithmen
Schnitt von Strecken

```
int ccw (Point p0, Point p1, Point p2) {
   int dx1, dx2, dy1, dy2;

   dx1 = p1.x - p0.x;
   dy1 = p1.y - p0.y;
   dx2 = p2.x - p0.x;
   dy2 = p2.y - p0.y;

   if (dy1*dx2 < dy2*dx1) {
      return 1;
   }
   else if (dy1*dx2 > dy2*dx1) {
      return -1;
   }
   else {    // dy1*dx2 == dy2*dx1 ==> kollinear

      if (dx1*dx2 < 0 || dy1*dy2 < 0) { // p0 in der Mitte
         return -1;
      } else if (dx1² + dy1² >= dx2² + dy2²) { // p2 in der Mitte
         return 0;
      } else {  // p1 in der Mitte
         return 1;
      }
   }
}
```

Schnitt von Strecken

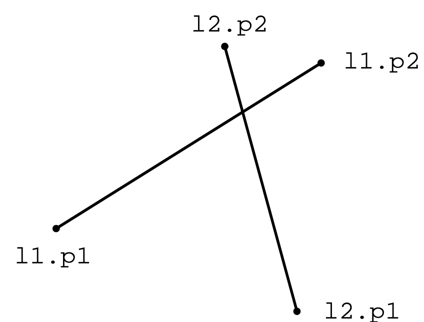
Geometrische Algorithmen
Schnitt von Strecken

Schnitt von Strecken kann man mit ccw einfach berechnen:

Falls die beiden Endpunkte beider Strecken sich auf verschiedenen Seiten der anderen Strecke befinden (ccw-Werte mit unterschiedlichen Vorzeichen), müssen die Strecken einander schneiden.

```
boolean intersect (Line l1, Line l2) {

   if ((ccw(l1.p1, l1.p2, l2.p1) * ccw(l1.p1, l1.p2, l2.p2)) <= 0) &&
       (ccw(l2.p1, l2.p2, l1.p1) * ccw(l2.p1, l2.p2, l1.p2)) <= 0)) {
      return true;
   } else {
      return false;
   }
}
```

Hinweis: Wenn beide Endpunkte auf unterschiedlichen Seiten liegen, haben die beiden ccw-Werte unterschiedliches Vorzeichen und Multiplikation ergibt Wert <= 0.



Inhalt

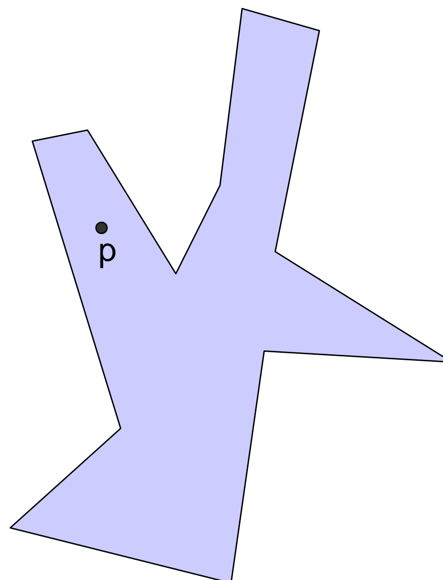
Geometrische Algorithmen

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Enthaltensein in einem Polygon

Geometrische Algorithmen
Enthaltensein in einem Polygon

- Problemstellung:
  - gegeben ein Punkt und ein beliebiges Polygon gegeben durch eine Reihe von Punkten
  - Frage, liegt der Punkt innerhalb oder ausserhalb des Polygons



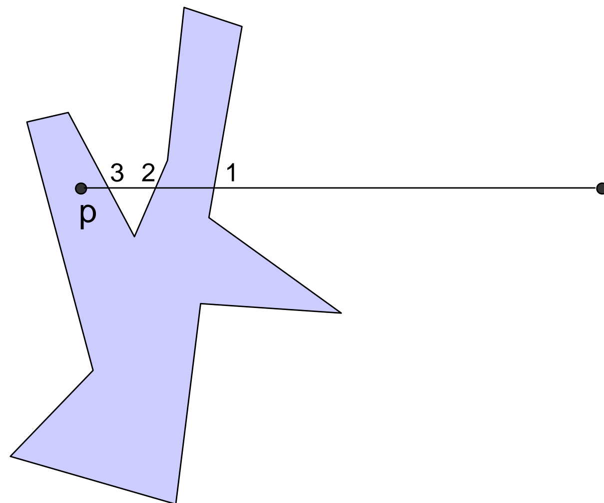
Enthaltensein in einem Polygon: Lösungsidee

Geometrische Algorithmen

Enthaltensein in einem Polygon

- Man verwendet eine **Teststrecke**, deren eine Eckpunkt der betrachtete Punkt ist und deren andere Eckpunkt beliebig aber sicher ausserhalb des Polygons ist.
- Man überprüft für alle Seiten des Polygons, ob sie mit dieser Teststrecke schneiden und zählt die Anzahl der Seiten, die sich mit der Teststrecke schneiden

==> ist die Anzahl ungerade liegt der Punkt innerhalb

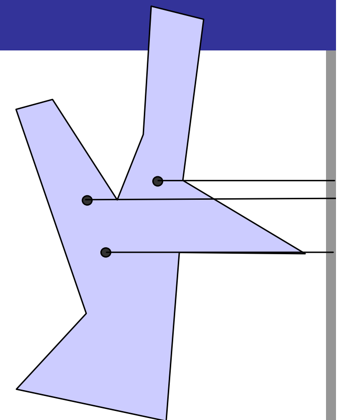


Enthaltensein in einem Polygon: Sonderfall

Geometrische Algorithmen

Enthaltensein in einem Polygon

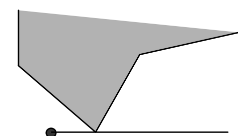
- Sonderbehandlung bedarf es, wenn die Teststrecke einen Eckpunkt berührt, bzw. die Teststrecke mit einer Seite zusammenfällt



Lösungsidee:

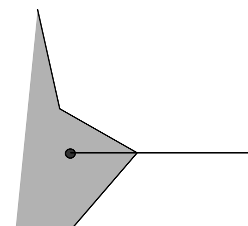
- Berührt die Teststrecke einen Punkt p[i] unterscheidet man wie folgt:
  - der vorangehende Punkt p[i-1] und der nächste Punkt p[i+1] liegen auf gleicher Seite der Teststrecke (beide oberhalb oder unterhalb)

    ==> Schnitt *nicht* zählen



  - der vorangehende Punkt p[i-1] und der nächste Punkt p[i+1] liegen auf unterschiedlichen Seiten der Teststrecke (einer oberhalb, einer unterhalb)

    ==> Schnitt *einmal* zählen



Enthaltensein in einem Polygon: Erklärungen zum Algorithmus (1)

Der Algorithmus arbeitet wie folgt:

- Argumente
  - `polygon` ist das Polygon gegeben durch eine Reihe von Punkten
  - `t` ist der Testpunkt, für den bestimmt werden soll, ob er innerhalb oder außerhalb des Polygons liegt.
- Variablen und Initialisierung
  - `lt` sei die Teststrecke, die bei Testpunkt `t` beginnt, waagrecht ist und bis zu einer x-Koordinate geht, die auf jeden Fall rechts vom Polygon liegt (groß ist).
  - n == Anzahl der Punkte des Polygons
  - p ein Array vom Typ `Point[]` der Länge `n+2`
  - `p[1..n]` enthält die Punkte des Polygons
  - `p[0] = p[n]` enthält den letzten Punkt des Polygons
  - `p[n+1] = p[1]` enthält den ersten Punkt des Polygons
  - `j` speichert den Index des zuletzt besuchten Punktes in p, der nicht auf der Teststrecke lag; Initialisierung `j == 0`, d.h. `p[0] == p[n]` darf *nicht* auf der Teststrecke (sonst wähle andere Teststrecke)
  - `lastAtTestline` ist ein boolesches Flag, das angibt, ob der zuletzt besuchte Punkt auf der Teststrecke lag; initial `false`

Enthaltensein in einem Polygon: Erklärungen zum Algorithmus (2)

- Schleife über alle Punkte `p[i]` an Stellen `i = 1` bis n

  prüfen ob der Punkt `p[i]` auf der Teststrecke liegt

    wenn ja, vormerken dass dieser Punkt auf Teststrecke (Flag `lastPointAtTestline`)

  sonst

    wenn vorheriger Punkt auf Teststrecke, prüfen ob letzter Punkt `p[j]` und dieser Punkt `p[i]` auf verschiedenen Seiten der Teststrecke liegen

      wenn ja, Zähler um eins erhöhen

    sonst auf Schnitt zwischen Teststrecke und Strecke zwischen letzten Punkt `p[j]` und diesen Punkt `p[i]` prüfen

      wenn ja, Zähler um eins erhöhen

- Ergebnis

  wurde eine ungerade Anzahl von Schnitten gezählt, liegt der Punkt innerhalb, sonst außerhalb

Enthaltensein in einem Polygon: Algorithmus

Geometrische Algorithmen
Enthaltensein in einem Polygon

```
boolean inside (Polygon polygon, Point t) {
    int count, i, j, n;
    Line lt, lp;

    boolean lastAtTestline = false;
    count = 0;
    j = 0;
    n = polygon.length; // Anzahl der Punkte
    Point[] p = new Point[n+2];
    p[1..n] = polygon.points();  p[0] = p[n];  p[n+1] = p[1];
    lt = new Line(t, new Point(big_integer, t.y)); // horizontale Linie von t

    for (i = 1.. n) {
        if (t.y == p[i].y && t.x <= p[i].x) { // Punkt auf Teststrecke
            lastAtTestline = true;
        } else {
            if (lastAtTestline) {
                if (p[i] und p[j] auf unterschiedlichen Seiten von lt ) {
                    count = count + 1;
                }
            } else {
                lp = new Line(p[i], p[j]); // Linie p[i]-p[j]
                if (intersect(lt, lp))  {
                    count = count + 1;
                }
            }
            lastAtTestline = false;
            j = i;
        }
    }
    return ((count %2) == 1);
} // end inside
```

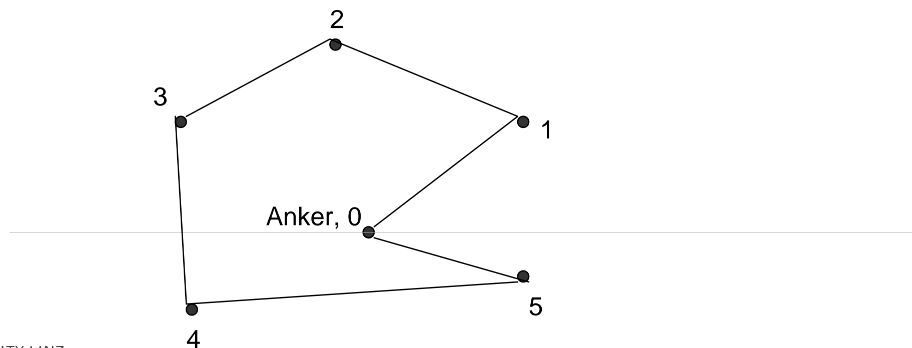
Inhalt

Geometrische Algorithmen

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Einfach geschlossener Pfad einer Punktmenge

- Für eine gegeben Punktmenge ist ein *einfach geschlossener Pfad* ein Pfad, der alle Punkte verbindet, zum Ausgangspunkt zurückkehrt (Zyklus) und der sich nicht überschneidet.
- Ein einfaches Verfahren, um einen einfach geschlossener Pfad zu berechnen, ist folgend:
  - wähle einen beliebigen Punkt als "Anker"
  - berechne dann für alle anderen Punkte den Winkel der Verbindung zum Anker
  - Bilde den Pfad, indem die Punkte in der Reihenfolge gegeben durch den Winkel besucht werden

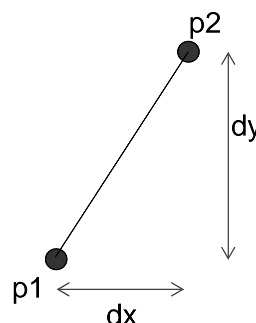


Funktion `theta` statt Winkelberechnung (2)

- Um die aufwendige Winkelberechnung zwischen zwei Punkten zu vermeiden, kann man eine einfachere Funktion verwenden, die die gleiche Ordnungsrelation wie der Winkel hat.
- Diese basiert auf der Term

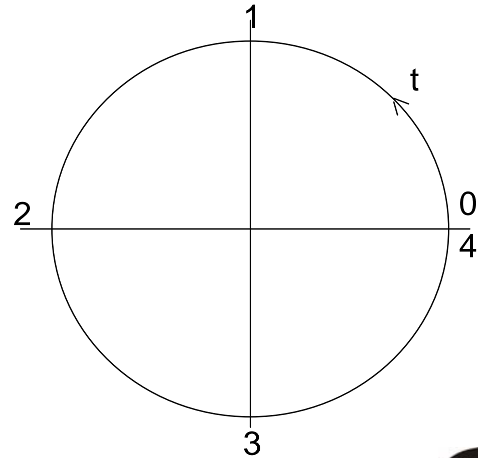
  `dy/(|dy| + |dx|)`

  wobei dx und dy die x- und y-Abstände zwischen den Punkten sind. Der Wert dieses Terms liegt zwischen -1 und 1.
- Folgende Funktion liefert für zwei Punkte p1 und p2 ein Zahl zwischen 0 und 360, die *nicht* der Winkel ist aber die gleiche Ordnungsrelation hat.



Funktion theta statt Winkelberechnung

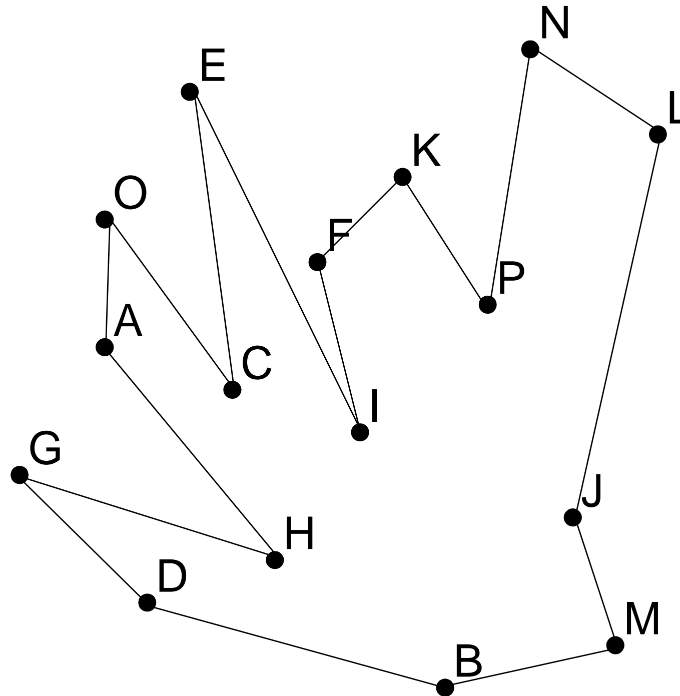
```
double theta (Point p1, Point p2) {
    int dx, dy, ax, ay;
    double t;
    dx = p2.x - p1.x;
    dy = p2.y - p1.y;
    ax = abs(dx);
    ay = abs(dy);
    if (ax == 0 && ay == 0) t = 0;
    else t = dy/(ax+ay);
    if (dx < 0) t = 2 - t;
    else if (dy < 0) t = 4 + t;
    return t * 90.0;
}
```



Beispiele zu theta

	dy/ (ax+ay)	dx dy	Berechnung t	t	theta
p1 → p2	1/2	dx > 0 dy > 0	t	0.5	45°
p1 → p2	1/2	dx < 0 dy > 0	t = 2 - t	1.5	135°
p1 → p2	-1/2	dx < 0 dy < 0	t = 2 - t	2.5	225°
p1 → p2	-1/2	dx > 0 dy < 0	t = 4 + t	3.5	315°

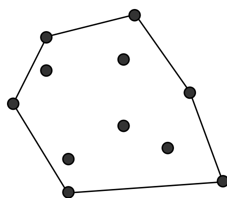
Einfach geschlossener Pfad: Arbeitsblatt



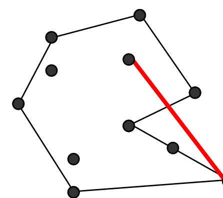
Konvexe Hülle

- Für eine gegeben Punktmenge ist die Grenze interessant, die alle Punkte miteinschließt
- Die konvexe Hülle einer Punktmenge schließt alle Punkte ein, wobei die Grenze minimal ist (man erhält die konvexe Hülle, wenn man einen "Faden" um die Punkte spannt)
- Die konvexe Hülle definiert ein konvexes Polygon

Eigenschaft Konvexität: Jede Linie, die zwei Punkte innerhalb des Polygons verbindet, liegt vollständig innerhalb des Polygons



konvex



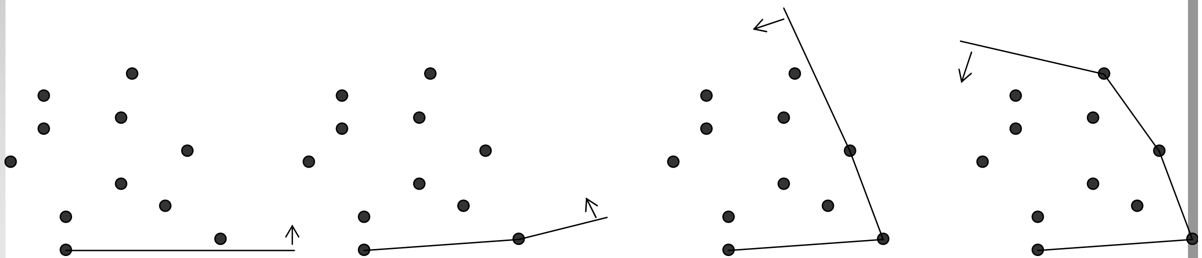
nicht konvex !!

Geometrische Algorithmen
Konvexe Hülle

Einwickeln

Verfahren des Einwickelns funktioniert, wie wenn man mit einem Faden von einem Ausgangspunkt (*Anker*) um die Punktmenge laufen würde

- Man beginne mit einem Punkt der garantiert zur konvexen Hülle gehört, z.B. dem Punkt mit kleinster y-Koordinate.
- Man schwenke einen Strahl beginnend von waagrecht solange nach oben, bis der Strahl den nächsten Punkt erreicht; dieser ist der nächste Punkt der konvexen Hülle.
- Man gehe zu diesem nächsten Punkt und verfahre in analoger Weise.



Geometrische Algorithmen
Konvexe Hülle

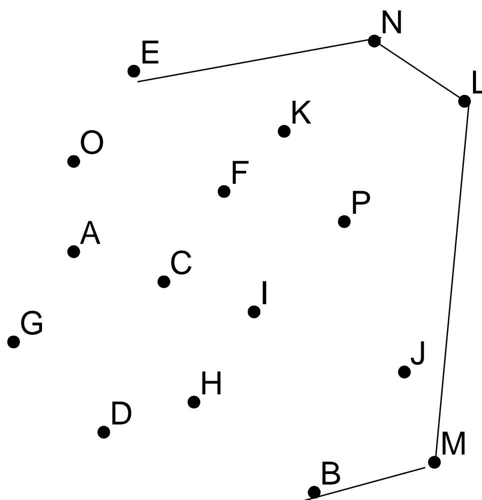
Einwickeln: Erklärungen zum Algorithmus

- Algorithmus funktioniert ähnlich wie Sortieralgorithmus `InsertionSort` (mit vertauschen der Elemente)
- Der Algorithmus arbeitet wie folgt:
  - Argument
    - im Array pa werden die zu umhüllenden Punkte übergeben
  - Variablen und Initialisierung
    - die Punkte pa werden ins Array p kopiert, wobei p um eine Stelle größer als pa ist.
    - es wird der Punkt mit kleinster y-Koordinate gesucht (`min`)
    - dieser Punkt kommt an erste Stelle `p[0]` als auch an letzte Stelle `p[n]`
  - Schleife solange nicht einen vollständigen Kreis gegangen
    - suche den Punkt der zum letzten besuchten Punkt (Stelle `m`) den kleinsten Winkel hat; dieser ist der nächste Punkt in der konvexen Hülle
    - vertausche den Punkt an Stelle `m+1` mit diesem minimalen Punkt
  - Ergebnis
    - die Punkte `p[0..m-1]` definieren die konvexe Hülle (an Stelle `m` ist Ausgangspunkt `p[0]`)

Einwickeln: Algorithmus

```
static Polygon wrap(Point[] pa) {
    int min, m, n;
    double minangle, v, theta;
    Point t;

    n = pa.length;                                    // erzeugen von p und kopieren
    Point[] p = new Point[n+1];                       //   von pa in p
    p[0..n-1] = pa[0..n-1];

    min = 0;                                          // suchen des Punktes mit
    for (i = 1 to n-1) {                              //   kleinster y-Koordinate
        if (p[i].y < p[min].y) min = i;
    }                                                 // stellen des Punktes an erste
    t = p[0]; p[0] = p[min]; p[min] = t; min = 0;     //   Stelle 0 als auch an letzte
    p[n] = p[0];                                      //   Stelle n (vertauschen der
                                                      //   Punkte an Stellen 0 und min)
    m = 0;  minangle = 0.0;
    do  {                                             // Suchen des Punktes mit
        min = n;  v = minangle;  minangle = 360;      //   minimalen Winkel zu Punkt p[m]
        for (int i= m+1; i <= n; i++) {               //   (winkel muss aber >0 als
            theta = theta(p[m], p[i]);                //   bisheriger Winkel sein)
            if (theta >= v  && theta < minangle) {
                min = i;  minangle = theta;           // stellen des minimalem Punkts
            }                                         //   an Stelle m+1 (vertauschen der
        }                                             //   Punkte an Stellen m+1 und min)
        m = m+1;
        t = p[m];  p[m] = p[min];  p[min] = t;        // solange nicht wieder am
    }  while  (p[m] != p[0]);                         //   Ausgangspunkt angelangt

    return new Polygon(p[0, m-1]); ;                  // Ergebnis ist das Polygon der
}                                                     //   Punkte 0 bis m-1
```

Einwickeln: Arbeitsblatt



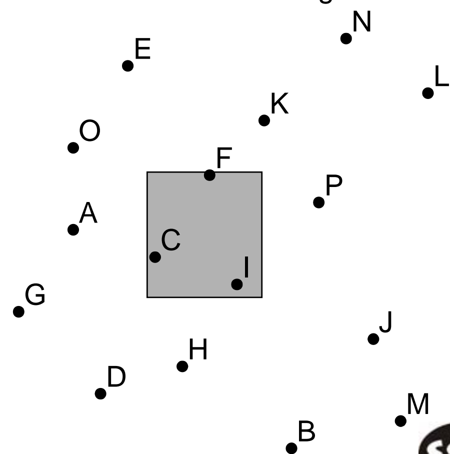
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
B	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P	B
B	M	C	D	E	F	G	H	I	J	K	L	A	N	O	P	B
B	M	L	D	E	F	G	H	I	J	K	C	A	N	O	P	B
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P	B
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P	B

Inhalt

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Motivation

- Viele geometrische Probleme sind Suchprobleme im 2-dimensionalen Raum
  - Welche Orte liegen in einem Umkreis von 100km von Linz
  - Welche Punkte liegen innerhalb vorgegeber Koordinaten (rechteckiger Bereich)
  - usw.
- Bekannte Suchverfahren und Datenstrukturen zur Suche lassen sich auf 2- und mehr-dimensionale Probleme verallgemeinern
- Im Folgenden wollen wir folgende Verfahren für die Bereichssuche angeben
  - 1-dimensionale Suche
  - Gitterverfahren
  - 2-dimensionale Bäume



Wiederholung: Suchen im 1-dimensionalen Raum

In einem 1-dimensionalen Problemraum kann die Bereichssuche folgend erfolgen

- einfache sequentielle Suche
- binäre Suche bei sortierten Elementen
- binäre Suchbäume

Geometrische Algorithmen
Geometrische Datenstrukturen und Bereichssuche

Wiederholung: Binäre Suche

- gegeben:
  - ein Feld mit allen Elementen aufsteigend sortiert
  - ein Suchbereich `(from, to)`
- gesucht:
  - alle Elemente, die im Bereich `(from, to)` liegen
- Lösung
  - binäre Suche nach kleinsten Element `min`, welches >= `from` ist
  - binäre Suche nach größtem Element `max`, welches =< `to` ist
  - alle Elemente von `min` bis `max` liegen im Bereich

0	1	2	3	4	5	6	7	8	9
2	5	7	10	12	17	19	23	26	27

`from 10` `to 20`

Geometrische Algorithmen
Geometrische Datenstrukturen und Bereichssuche

1-dimensionale binäre Suche bei 2-dimensionalen Problemen

- Prinzip:
  - 1-dimensionale binäre Suche entlang einer Koordinate (z.B. x-Koordinate)
  - reine sequentielle Suche entlang der anderen Koordinate (z.B. y-Koordinate)
- Punkte werden nach der x-Koordinate aufsteigend sortiert in einem Feld gespeichert

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
G	O	A	D	E	C	H	F	I	K	B	P	N	j	M	L

from to

Geometrische Algorithmen
Geometrische Datenstrukturen und Bereichssuche

Gitterverfahren für 2-dimensionale Suche

- Idee:
  - Man legt ein künstliches Gitter über die Fläche und teilt so die gesamte Fläche in einzelne Quadrate gleicher Größe ein
  - für die Quadrate bestimmt man alle Punkte, die im Quadrat enthalten sind
  - die Mengen von Punkten in den Quadraten speichert man in einer Matrix

Matrix

	0	1	2	3
0	E		K,N	L
1	O,A	F	P	
2	G	C,I,H		J
3	D		B	M

Gitterverfahren für 2-dimensionale Suche: Parameter

Im folgenden sei angenommen, dass sich die Punkte in einem quadratischen Bereich von `(0,0)` bis `(max, max)` befinden

folgende Parameter sind für das Verfahren wichtig

- sei `N` die Anzahl der Punkte insgesamt
- sei `max` die größte Koordinate eines Punktes in x- oder y-Richtung
- `size` ist die Größe der Quadrate
  - soll so bestimmt werden dass durchschnittlich kleine Anzahl von Elementen in einem Quadrat auftreten (z.B. durchschnittlich 1),
  - Berechnung folgend

    $$size = \lceil \max / \sqrt{N / M} \rceil$$

    mit N = Anzahl der Punkte, M gewünschte durchschnittliche Anzahl der Punkte pro Gitterelement
- `n` ist die Anzahl der Quadrate je Richtung == Größe der Matrix
  - ergibt sich aus `n = max / size + 1`
- daraus ergibt sich für einen Punkt `(x, y)`, dass der Punkt im Matrixelement `(i, j)` gespeichert wird mit

```
i = x / size
j = y / size
```

(ganzzahlige Division !!!!)

Gitterverfahren für 2-dimensionale Suche: Erzeugung der Matrix

Folgender Algorithmus erzeugt die Matrix von Mengen von Punkten

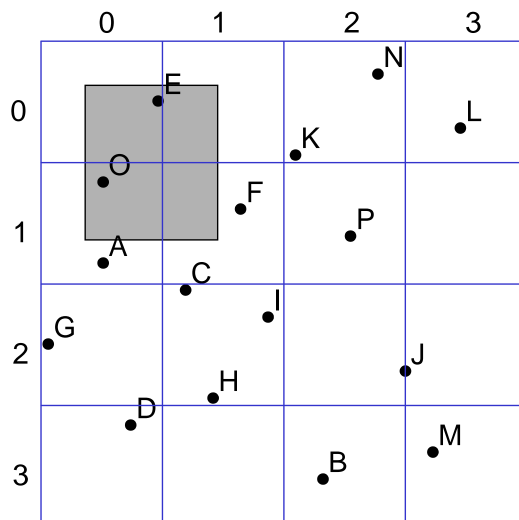
```
Set[][] computeGrid(Point[] points) {

    int max = größte x- oder y-Koordinate der Punkte in points
    int size = ⌈max / sqrt(points.length)⌉; // M == 1
    int n = max / size + 1;
    Set[][] grid = new Set[n][n];

    for (int i = 0 to n-1) {
        for (int j = 0 to n-1) {
            grid[i, j] = new Set();
        }
    }

    for all p in points {
        int i = p.x / size;
        int j = p.y / size;
        grid[i, j].add(p);
    }
    return grid;
}
```

Gitterverfahren für 2-dimensionale Suche: Bereichssuche

Folgender Algorithmus sucht alle Punkte in einem bestimmten Bereich gegeben durch ein Rechteck

```
Set pointsInRange (Set[][] grid, Rectangle rect, int size) {

   int minX, maxX, minY, maxY;
   Set resultSet = new Set();

   minI  = rect.leftCoord() / size;
   minJ  = rect.upperCoord() / size;
   maxI  = rect.rightCoord() / size;
   maxJ  = rect.lowerCoord() / size;

   for (int i  = minI  to maxI) {
      for (int j  = minJ to maxJ) {
         for all Points p in grid[i, j] {
            if (insiderect(p, rect) {
               resultSet.add(p);
            }
         }
      }
   }
   return resultSet;
}
```

Gitterverfahren für 2-dimensionale Suche: Beispiel Bereichssuche

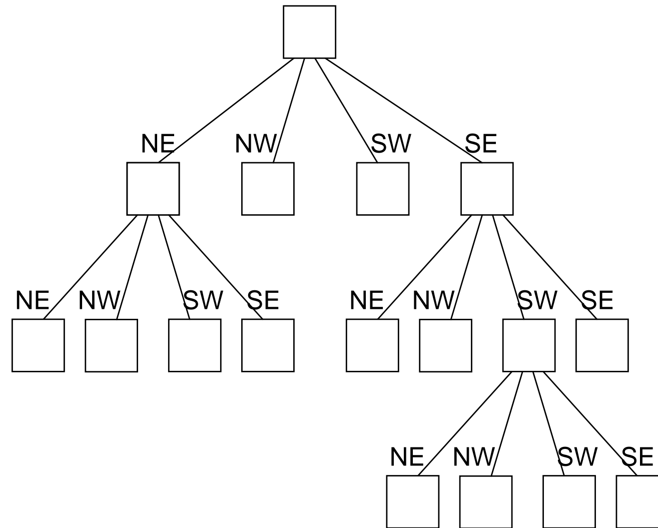


Matrix

	0	1	2	3
0	E		K,N	L
1	O,A	F	P	
2	G	C,I,H		J
3	D		B	M

Quadtrees

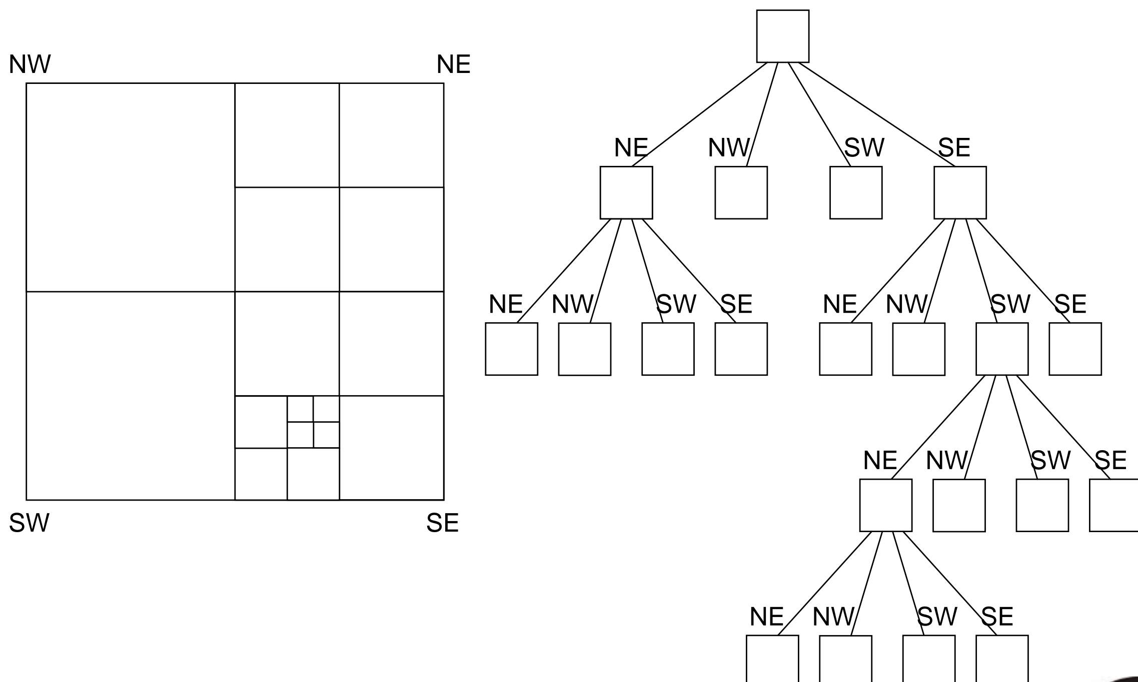
- Quadtrees sind Bäume mit einer vierfachen Verzweigung
  - North East (NE)
  - North West (NW)
  - South West (SW)
  - South East (SE)



Quadtrees zur Einteilung von Gebieten

Geometrische Algorithmen

- Gebiete können beliebig fein unterteilt werden, indem man jedes Quadrat nach Bedarf in 4 weitere Quadrate unterteilt



Geometrische Algorithmen

ADT Quadtrees

```
enum Orientation { NE, NW, SW, SE }
class Rect {
  int x, y, w, h;
  Rect(int x, int y, int w, int h) { … }
}
class Quadtree {
  Rect rect;
  Set<Point> points;
  Quadtree ne, nw, se, sw;

  Quadtree(Rect rect)

  Rect getRect()                       // Rechteck
  Set<Point> getPoints()               // Punkte im Knoten (nur gesetzt wenn isLeaf())
  boolean isLeaf()                     // Test, ob keine weitere Zerlegung

  Quadtree getSubtree(Orientation o)   // Zugriff auf Unterbaum
  Quadtree northEast()                 // nord-östlicher Unterbaum
  Quadtree northWest()                 // nord-westlicher Unterbaum
  Quadtree southWest()                 // süd-westlicher Unterbaum
  Quadtree southEast()                 // süd-östlicher Unterbaum

  void setSubtree(Orientation o, Quadtree qtr) // Setzen des Unterbaums
  void setNorthEast(Quadtree qtr)      // Setzen des nord-östlichen Unterbaums
  void setNorthWest(Quadtree qtr)      // Setzen des nord-westlicher Unterbaums
  void setSouthWest(Quadtree qtr)      // Setzen des süd-westlicher Unterbaums
  void setSouthEast(Quadtree qtr)      // Setzen des süd-östlicher Unterbaums
}
```

Geometrische Algorithmen

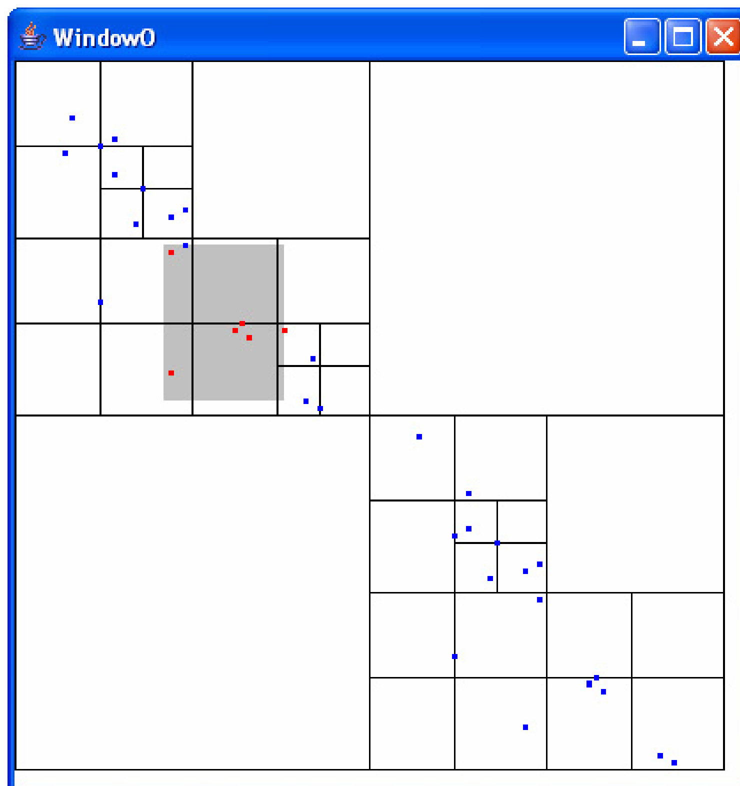
Aufbau eines Quadtrees

```
Quadtree buildQuadtree(Rect r, Set<Point> ps) {
  Quadtree qtr = new Quadtree(r);
  if (ps.size() >= maximale Anzahl der Elemente in Quadrat) {
    int w2 = r.w / 2;
    int h2 = r.h / 2;
    // NE
    Rect neRect = new Rect(r.x + w2, r.y, r.w-w2, h2);
    qtr.setNorthEast(buildQuadtree(neRect, pointsInRect(ps, neRect)));
    // NW
    Rect nwRect = new Rect(r.x, r.y, w2, h2);
    qtr.setNorthWest(buildQuadtree(nwRect, pointsInRect(ps, nwRect)));
    // SW
    Rect swRect = new Rect(r.x, r.y + h2, w2, r.h-h2);
    qtr.setSouthWest(buildQuadtree(swRect, pointsInRect(ps, swRect)));
    // SE
    Rect seRect = new Rect(r.x + w2, r.y + h2, r.w-w2, r.h-h2);
    qtr.setSouthEast(buildQuadtree(seRect, pointsInRect(ps, seRect)));
  } else {
    qtr.points = ps;
  }
  return qtr;
}

Set<Point> pointsInRect(Point[] p, Rect r) {
  // Berechnung der Punkte in r
  …
}
```

Bereichssuche mit Quadtrees

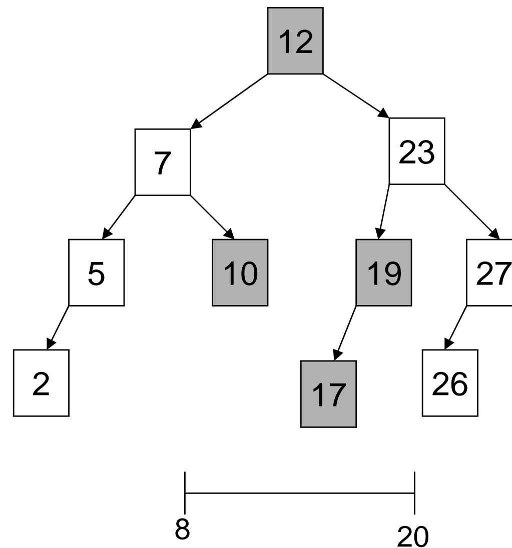
```
Set<Point> lookup(Quadtree qtr, Rect area) {

  Set<Point> result;
  if (qtr.isLeaf()) {
    for (p in  qtr.getPoints()) {
      if (inside(p, area)) {
         Add p to result
      }
    }
  } else {
    for (o in {NE, NW, SW, SE}) {
      Quadtree subQtr = qtr.getSubtree(o);
      if (intersect(subQtr.getRect(), area)) {
        Set<Point> pointsInSubQtr = lookup(subQtr, area);
        result = result united with pointsInSubQtr
      }
    }
  }
  return result;
}
```

Quadtrees Beispiel



Wiederholung: Binäre Suchbäume

- In einem binären Suchbaum sind bei den Knoten die Werte so gespeichert, dass für einen Knoten
  - die Werte aller Knoten im linken Unterbaum kleiner als der Wert des Knotens sind
  - die Werte aller Knoten im rechten Unterbaum größer als der Wert des Knotens sind



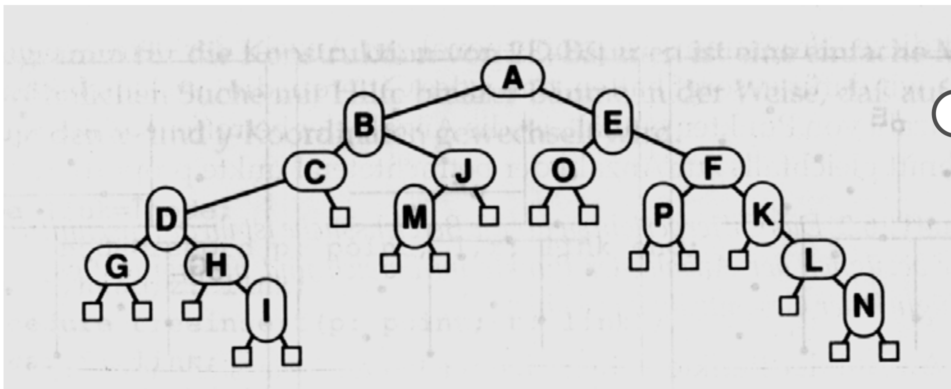
Wiederholung: Bereichssuche in binären Suchbäumen

- Damit kann eine Bereichssuche leicht rekursiv realisiert werden.
- Folgender rekursive Algorithmus
  - erhält einen Knoten node und einen Bereich in Form von zwei Werten `from` und `to`
  - sammelt alle Knoten aus Unterbaum von node, die im Bereich liegen, in der globalen Variablen nodesInRange

```
static List nodesInRange = new ArrayList();

void treeRange(Node node, int from, int to) {

    if (node == null) return;

    if (from <= node.value() <= to) {
        nodesInRange.add(node.value())
    }
    if (node.value() > from) {
        treeRange(node.left(), from, to);
    }
    if (node.value() <= to) {
        treeRange(node.right(), from, to);
    }
}
```

2D-Suchbäume: Prinzip

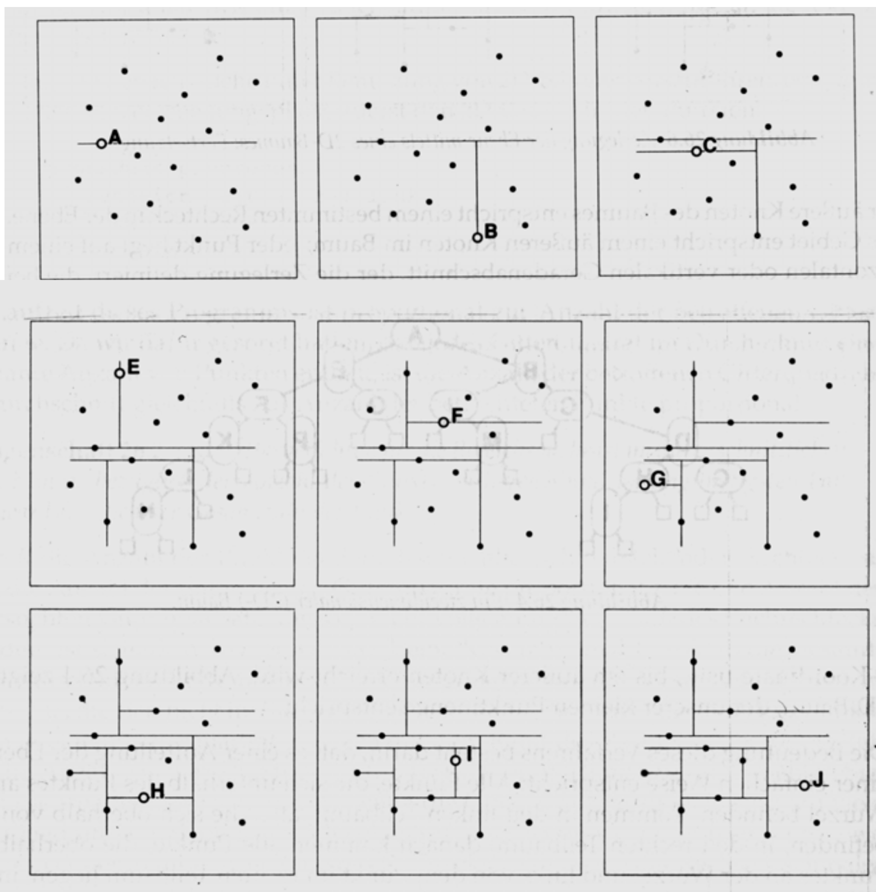
- 2D-Suchbäume sind eine direkte Erweiterung der binären Suchbäume auf 2-dimensionale Probleme
- im 2D-Suchbaum werden die Punkte bei den Knoten gespeichert, wobei **alternierend** die x- und y-Koordinaten als Schlüssel verwendet werden
  - wenn x-Koordinate Schlüssel,
    dann Knoten im linken Unterbaum alle kleinere x-Koordinaten
    und Knoten im rechten Unterbaum alle größere x-Koordinaten
  - wenn y-Koordinate Schlüssel,
    dann Knoten im linken Unterbaum alle kleinere y-Koordinaten
    und Knoten im rechten Unterbaum alle größere y-Koordinaten



vertikal
(y-Teilung)

horizontal
(x-Teilung)

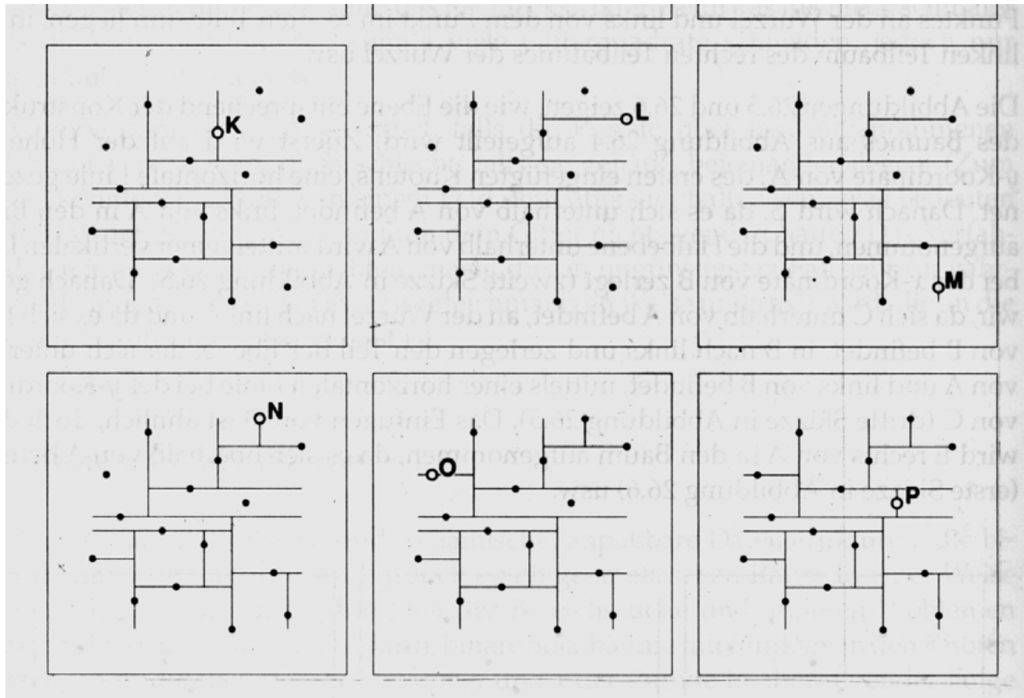
2D-Suchbäume: Beispiel



2D-Suchbäume: Beispiel (2)

Geometrische Algorithmen

Geometrische Datenstrukturen und Bereichssuche



2D-Suchbäume: Erzeugen des 2D-Baumes

Geometrische Algorithmen

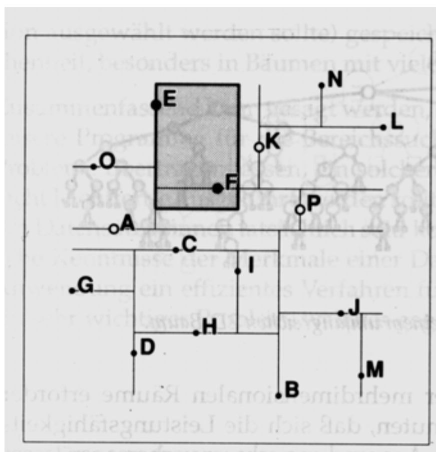
Geometrische Datenstrukturen und Bereichssuche

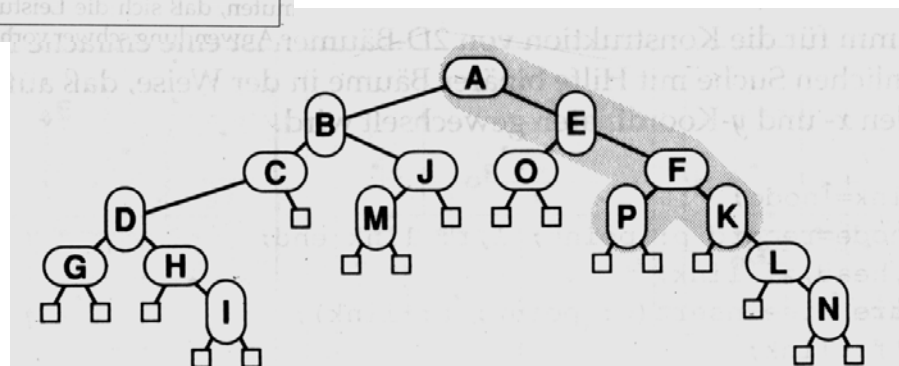
```
treeInsert2D(Point p, BinaryTree tree) {

   BinaryTreeNode node, lastNode;
   node = tree.root();
   boolean nodeDividesY = true;
   boolean goLeft;

   do {
      if (nodeDividesY ) goLeft = p.y < ((Point)node.value).y;
      else               goLeft = p.x < ((Point)node.value).x;
      lastNode = node;
      if (goLeft )    node = node.left();
      else            node = node.right();
      nodeDividesY = ! nodeDividesY; // alternierend x und y
   } until (node = null);

   node = new BinaryTreeNode(p);
   if (goLeft )    lastNode.setLeft(node);
   else            lastNode.setRight(node);
}
```

2D-Suchbäume: Bereichssuche mit 2D-Baum

```
static List pointsInRange;

treeRange2D(BinaryTreeNode node, Rectangle rect, boolean nodeDividesY)
{
   if (node == null) {
      return;
   } else {
      Point p = (Point) node.value;
      if (insiderect(p, rect))
         pointsInRange.add(p);
      if (nodeDividesY) {
         if (p.y > rect.lowerCoord())
            treeRange2D(node.left(), rect, ! nodeDividesY);
         if (p.y <= rect.upperCoord ())
            treeRange2D(node.right(), rect, ! nodeDividesY);
      } else {
         if (p.x > rect.leftCoord())
            treeRange2D(node.left(), rect, ! nodeDividesY);
         if (p.x <= rect.rightCoord())
            treeRange2D(node.right(), rect, ! nodeDividesY);
      }
   }
}
```

2D-Suchbäume: Beispiel Bereichssuche mit 2D-Baum





Literatur

Geometrische Algorithmen

- R. Sedgewick, Algorithmen, Addison-Wesley, 2002, Seiten 427 - 441
- Ottmann, Wiedemayer, *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996, Seiten 444 - 471

Inhalt

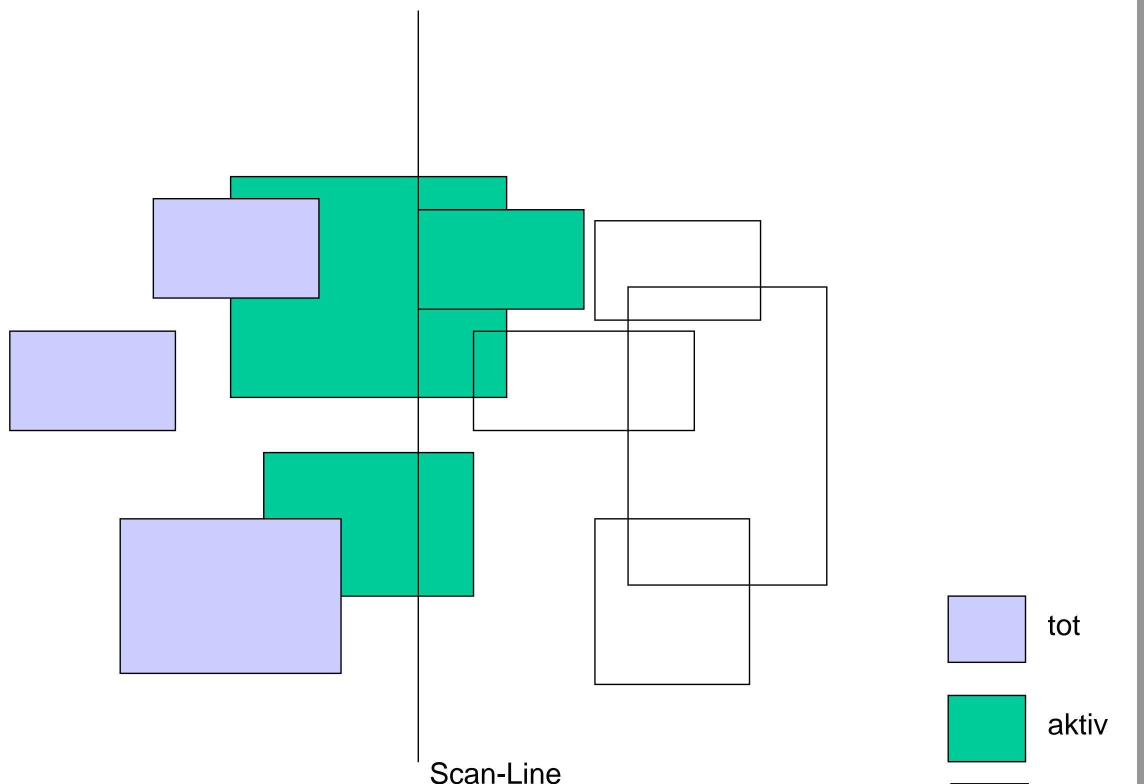
Geometrische Algorithmen

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Scan-Line-Methode

- Die Scan-Line-Methode ist ein allgemeines Verfahren zur Lösung von geometrischen Problemen mit vielen Anwendungsmöglichkeiten.
- Das Prinzip ist ein 2-dimensionales Problem in eine *dynamische* Abfolge von 1-dimensionalen Problemen zu zerlegen.
- Die Vorstellung ist eine vertikale (oder horizontale) Linie über die geometrische Szene zu schwenken.
- Dadurch können die Menge der geometrischen Objekte der Szene zu jedem Zeitpunkt in 3 unterschiedliche Klassen eingeteilt werden
  - die *toten* Objekte, die bereits vollständig von der Scan-Line überstrichen wurden (bereits hinter der Scan-Line liegen)
  - die *aktiven* Objekte, die derzeit von der Scan-Line geschnitten werden
  - die *inaktiven* oder *schlafenden* Objekte, die noch vor der Scan-Line liegen
- In den Algorithmen sind diese drei Klassen folgend zu interpretieren
  - die toten Objekte sind die bereits behandelten Objekte
  - die aktiven Objekte sind die gerade behandelten Objekte
  - die inaktiven Objekte sind die noch zu behandelnden Objekte

Beispielszene



Basisalgorithmus

- Scan-Line wird nicht kontinuierlich über die Szene geschwenkt

  sondern schrittweise von einem Haltepunkt zum nächsten, wobei die Haltepunkte so sind, dass sich interessante Änderungen der Szene ergeben.
  (z.B. sich die Menge der toten, aktiven oder inaktiven Objekte ändert)

- Folgende Mengen werden verwendet

  Q ist objekt- oder problemabhängige Folge von Haltepunkten in aufsteigender x-Reihenfolge

  L ist die Menge der jeweils aktiven Objekte, eventuell **problemorientiert angeordnet**

```
Scan-Line-Algorithm(GeomObject[] objs) {
   Q = objekt- od. problemabhängige Folge von Haltepunkte von objs
   L = { };
   while ( Q != { } ) {
        nimmt nächsten Haltepunkt aus Q und entferne ihn
        update L und gib (problemabhängige) Teilantwort aus
   }
}
```

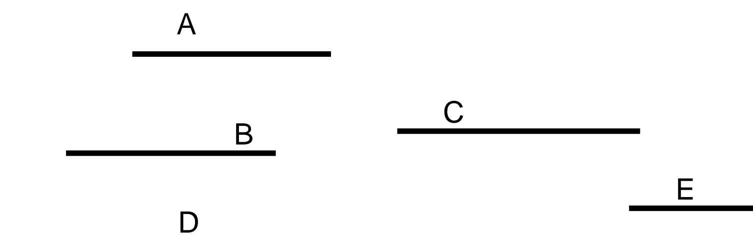
Anwendung bei Sichtbarkeitsproblem

- Gegeben:
  - eine Reihe von horizontalen Linien
- Frage:
  - welche Linien sind direkt benachbart, d.h. teilen sich gemeinsamen x-Bereich und sind nicht durch andere Linie getrennt.
- Lösung durch Scan-Line-Methode:
  - alle Paare von Linien, die gleichzeitig in Menge L der aktiven Objekte auftreten und bezüglich y-Koordinate direkt benachbart sind

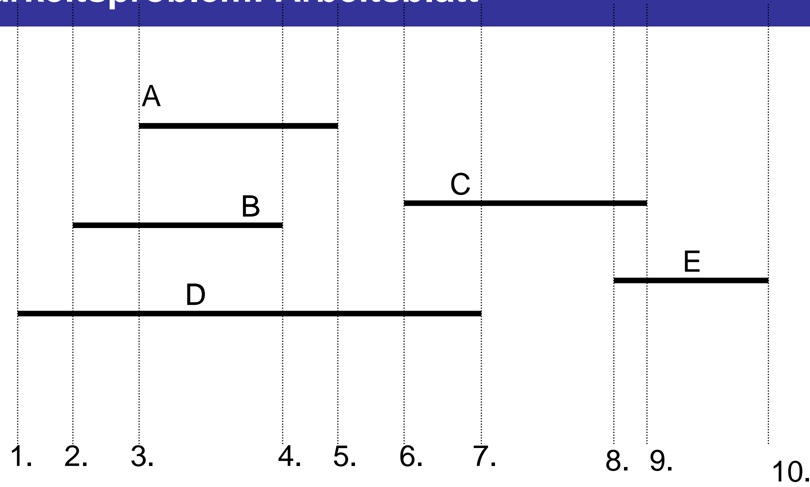
Beispiel:

bei der folgenden Szene sind dies die Paare:

(A, B), (A, D), (B, D), (C, D), (C, E)



Sichtbarkeitsproblem: Arbeitsblatt




```
1.  L = {D}
2.  L = {B, D}    sichtbar BD
3.  L = {A, B, D} sichtbar AB
4.  L = {A, D}    sichtbar AD
5.  L = {D}
6.  L = {C, D}    sichtbar CD
7.  L = {C}
8.  L = {C, E}    sichtbar CE
9.  L = {E}
10. L = {}
```

Sichtbarkeitsproblem: Algorithmus

- Algorithmus verwendet eine Sortierung der horizontalen Linien in L nach ihrer y-Koordinate
- Dadurch ergibt sich die Sichtbarkeit von 2 Linien durch die direkte Nachbarschaft in L,

  d.h. sind 2 Linien s1 und s2 einmal in L direkte Nachbarn, sind sie sichtbar.

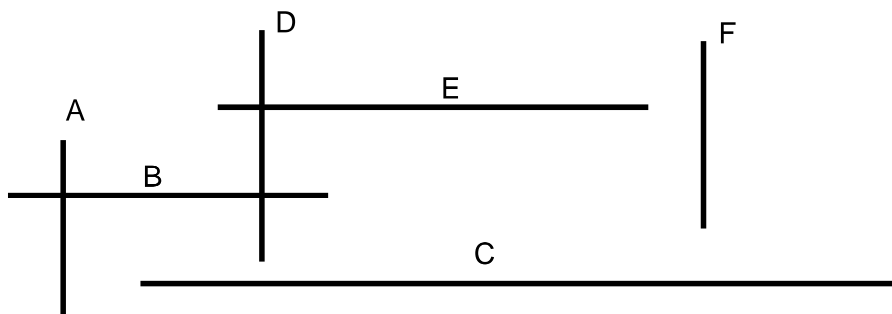
```
Sichtbarkeit(Line[] seg) {

   Q = Folge der x-Koordinaten der Anfangs- und Endpunkte der
       Liniensegmente seg

   L = { };  // Menge der jeweils aktiven Liniensegmente
                in aufsteigender y-Reihenfolge

   while ( Q != { } ) {
        p = nächster Haltepunkt in Q; entferne p aus Q
        if (p ist linker Endpunkt einer Linie s in seg) {
            füge s in L ein, sodass Linien in L nach y-Koordinate sortiert;
            bestimme Vorgänger v und Nachfolger n von s in L und
            gib (v, s) und (s, n) als Paare von sichtbaren Segmenten aus
        } else { // p ist rechter Endpunkt einer Linie s in seg
            bestimme Vorgänger v und Nachfolger n von s in L
            entferne s aus L
            gib (v, n) als Paare von sichtbaren Segmenten aus
        }
   }
}
```

Anwendung bei Linienschnittproblem (iso-orientierte Liniensegemente)

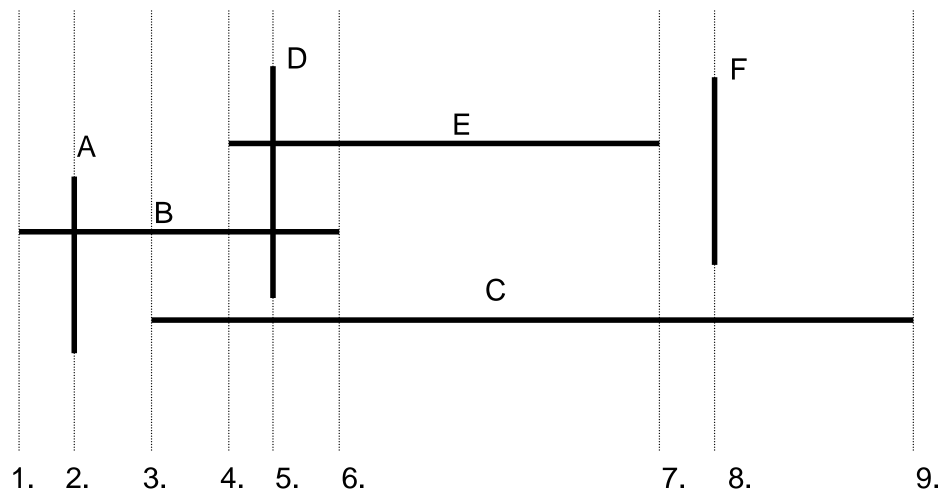
- Gegeben:
  - eine Reihe von horizontalen und vertikalen Liniensegmenten
- Frage:
  - welche Linien schneiden sich
- Lösung:
  - man betrachtet die horizontalen Linien und nimmt diese in die Menge L der aktiven Elemente auf
  - trifft man auf eine vertikale Linien, prüft man alle aktiven horizontalen Linien auf Schnitt

<u>Beispiel</u>:

Schnitte : `(A, B), (B, D), (D, E)`



Linienschnittproblem (iso-orientierte Liniensegemente): Arbeitsblatt




```
1. L = {B}
2. Schnitt A mit Elementen aus L = {B} prüfen
3. L = {B, C}
4. L = {B, C, E}
5. Schnitt D mit Elementen aus L = {B, C, E} prüfen
6. L = {C, E}
7. L = {C}
8. Schnitt F mit Elementen aus L = {C} prüfen
9. L = {}
```

Linienschnittproblem (iso-orientierte Liniensegemente): Algorithmus

Geometrische Algorithmen

Scan-Line Verfahren und Linienschnittproblem

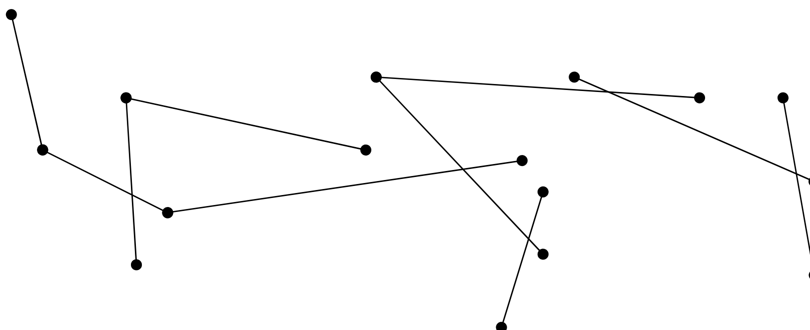
```
Schnittproblem(Line[] segments) {

    Q = Folge der x-Koordinaten von Anfangs- und Endpunkten der horizontalen
    Liniensegmente und x-Koordinaten der vertikalen Linien in segments

    L = { };   // Menge der jeweils aktiven horizontaler Liniensegmente

    while ( Q != { } ) {
        p = nächster Haltepunkt in Q; entferne p aus Q
        if (p ist linker Endpunkt einer horizontalen Linie s) {
            füge s in L ein, sodass Linien in L nach y-Koordinate sortiert;
        } else if (p ist rechter Endpunkt einer horizontalen Linie s) {
            entferne s aus L
        } else {  // p ist x-Koordinate einer vertikalen Linie
            finde vertikale Linie v mit Endpunkten (p, y1) und (p, y2)
            for all horizontalen Linien  h = ((x1, y), (x2, y)) in L {
                if (y1 <= y <= y2) {
                    gib (h, v) als schneidende Liniensegmente aus;
                }
            }
        }
    }
}
```

Allgemeines Linienschnittproblem

Geometrische Algorithmen

Scan-Line Verfahren und Linienschnittproblem

- Schnitt von beliebigen Liniensegmenten
- Anwendung bei vielen Schnittproblemen (bei Darstellung von Verläufen durch Segmente)
  - Schnitt von Verläufen wie Straßenzüge und Flüsse
  - Schnitt von Gebieten (siehe später)



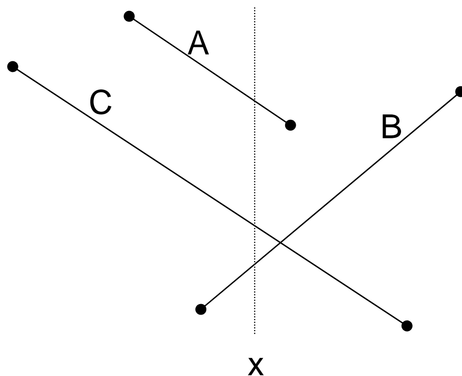
x-oberhalb Ordnungsrelation

Das allgemeine Linienschnittverfahren verwendet die x-oberhalb Ordnungsrelation wie folgt:

- Liniensegmente lassen sich folgend ordnen
  - gegeben eine beliebige x-Koordinate
  - ein Segment s1 heisst *x-oberhalb* eines Segments s2

    s1 ↑x s2

    wenn der Schnittpunkt der vertikalen x-Linie mit s1 oberhalb des Schnittpunkts der vertikalen x-Linie mit s2 liegt



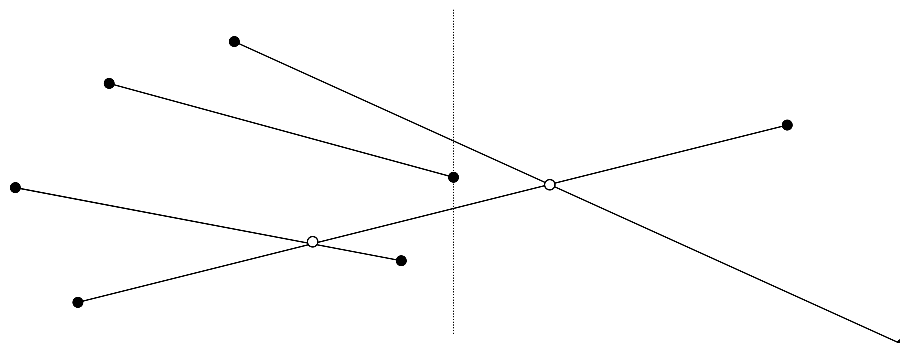
für dieses Beispiel gilt:

- A ↑x B
- A ↑x C
- C ↑x B

Allgemeines Linienschnittproblem: Prinzip

- Das allgemeine Linienschnittproblem arbeitet nach der Scan-Line-Methode
- wobei die aktiven Segmente in L nach der x-oberhalb Ordnung (entlang der Scan-Line) sortiert werden
- und man sich folgende Beobachtung zu nutze macht:
  - Für 2 beliebige Segmente A und B gilt: Wenn A und B sich schneiden, dann gibt es eine Stelle x links von Schnittpunkt, so dass A und B in der Ordnung ↑x unmittelbar aufeinander folgen

➔ und es folgt, dass man nur **benachbarte Segmente in L auf Schnitt prüfen muss**.



Allgemeines Linienschnittproblem: Vorgehen

- Schwenken der Scan-Line von links nach rechts über alle Haltepunkte in Q
  (anfänglich sind die Haltepunkte alle Anfangs- und Endpunkte aller Segmente)
- an jedem Haltepunkt x der Scan-Line wird die Ordnung $\uparrow x$ für alle Segmente in L aufrecht erhalten
- Durch die Umsortierung von L gibt es immer wieder neue direkte Nachbarn in L
  - für diese neuen Nachbarn muss man Schnittpunkte berechnen (nur die rechts der Scan-Line sind zu betrachten)
  - und falls vorhanden diese Schnittpunkte in die Menge der Haltepunkte Q einsortieren

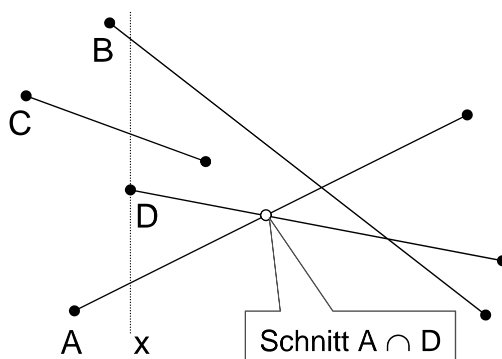
Geometrische Algorithmen
Scan-Line Verfahren und Linienschnittproblem

Vorgehen bei Haltepunkt linker Endpunkt

Haltepunkt: linker Endpunkt eines Liniensegments s

Vorgehen:

- Einsortieren des Segments s in L nach der x-oberhalb Ordnung
- Vorgänger v und Nachfolger n mit s auf Schnitt prüfen
- Schnittpunkte $v \cap s$ und $n \cap s$ (falls vorhanden) in Q einsortieren



Beispiel: linker Eckpunkt von D

- Einsortieren von D
  - vorher: L = (A, C, B)
  - nachher: L = (A, D, C, B)
- für neuen Vorgänger A und Nachfolger C Schnitte
  - $A \cap D$
  - $D \cap C$

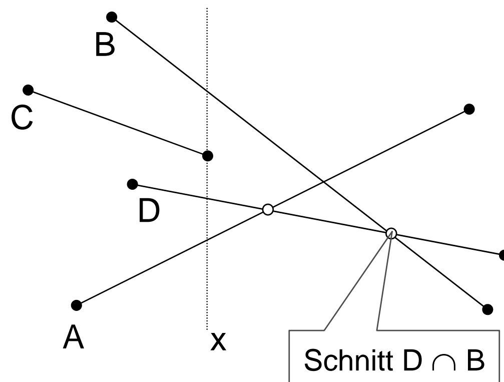
  prüfen
- Schnittpunkt $A \cap D$ in Q einsortieren

Vorgehen bei Haltepunkt rechter Endpunkt

Haltepunkt: rechter Endpunkt eines Liniensegments s

Vorgehen:

- Löschen des Segments s in L
- Dadurch werden Vorgänger v und Nachfolger n von s nun direkte Nachbarn
- Schnittpunkt v ∩ n (falls vorhanden) in Q einsortieren



Beispiel: rechter Eckpunkt von C

- Löschen von C
  - vorher: L = (A, D, C, B)
  - nachher: L = (A, D, B)
- Vorgänger D und Nachfolger B von C auf Schnitt
  - D ∩ B

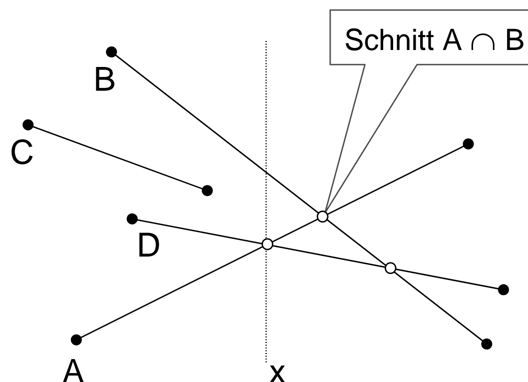
  prüfen
- Schnittpunkt D ∩ B in Q einsortieren

Vorgehen bei Haltepunkt Schnittpunkt zweier Liniensegmente

Haltepunkt: Schnittpunkt s1∩s2 zweier Liniensegmente s1 und s2

Vorgehen:

- Vertauschen von s1 und s2 in L
- Durch Vertauschen erhält s1 einen neuen Nachfolger n und s2 einen neuen Vorgänger v
- Schnittpunkte s1 ∩ n und s2 ∩ v (falls vorhanden) in Q einsortieren



Beispiel: Schnittpunkt A ∩ D

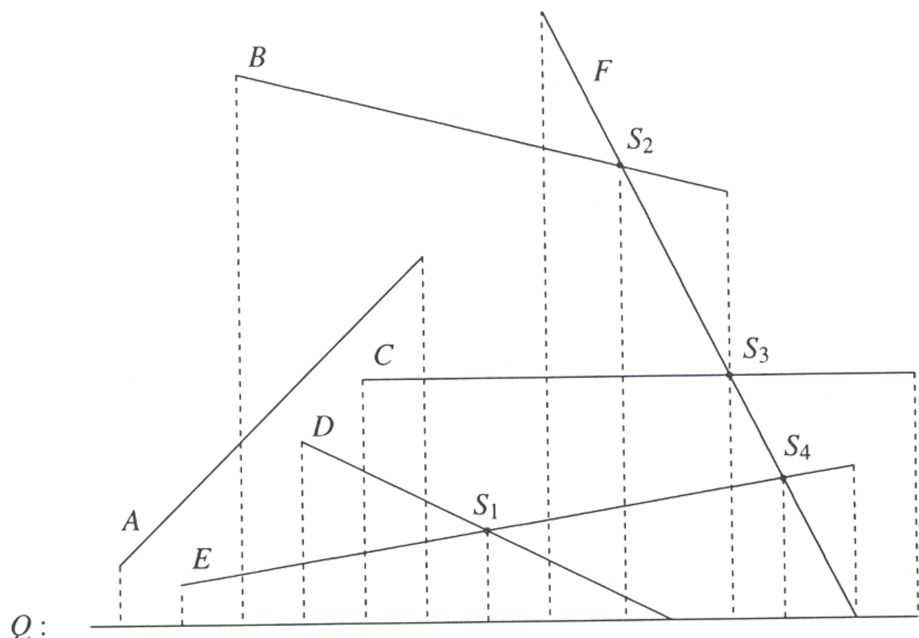
- Vertausen von A und D in L
  - vorher: L = (A, D, B)
  - nachher: L = (D, A, B)
- neuen Nachfolger B von A und A auf Schnitt
  - A ∩ B

  prüfen
- Schnittpunkt A ∩ B in Q einsortieren

Algorithmus allgemeines Linienschnittproblem

```
Schnittproblem(Line[] segments) {

    Q = Folge der Anfangs- und Endpunkte der Liniensegmente in segments nach x-
    Koordinate sortiert

    // Menge L der jeweils aktiven Liniensegmente nach x-oberhalb sortiert
    L = { };

    // Ergebnismenge R mit allen Schnittpunkten
    R = { };

    while ( Q != { } ) {
        p = nächster Haltepunkt in Q; entferne p aus Q
        if (p ist linker Endpunkt einer Linie s) {
            füge s in L ein, sodass Linien in L nach x-oberhalb sortiert;
            v = Vorgänger von s in L
            n = Nachfolger von s in L
            if (v∩s ≠ ∅ && v∩s rechts der Scan-line) Einfügen von v∩s in Q
            if (s∩n ≠ ∅ && s∩n rechts der Scan-line) Einfügen von s∩n in Q
        } else if (p ist rechter Endpunkt einer Linie s) {
            v = Vorgänger von s in L
            n = Nachfolger von s in L
            entferne s aus L
            if (v∩n ≠ ∅ && v∩n rechts der Scan-line) Einfügen von v∩n in Q

        } else {  // p ist Schnittpunkt von Linie s1 und s2
            R = R + s1∩s2
            vertause s1 und s2 in L
            v = Vorgänger von S2 in L
            n = Nachfolger von s1 in L
            if (v∩s2 ≠ ∅ && v∩s2 rechts der Scan-line) Einfügen von v∩s2 in Q
            if (s1∩n ≠ ∅ && s1∩n rechts der Scan-line) Einfügen von s1∩n in Q
        }
    }
}
```

Algorithmus allgemeines Linienschnittproblem: Beispiel



Scan-line-Methode: Weitere Anwendungen

- In ähnlicher Weise lassen sich viele weitere Probleme nach der Scan-Line-Methode effizient lösen, wie z.B.:
  - Schnittproblem bei Rechtecken
  - Schnitt von beliebigen Polygonen
  - Enthaltensein von Punkten in Rechtecken (*Inklusion*)
  - ...
- Näheres siehe:
  - Ottmann, Wiedemayer, *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996, Seiten 420 – 435
  - M. de Berg et al., *Computational Geometry*, Springer, 2000.

Literatur

- Ottmann, Wiedemayer, *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996, Seiten 420 - 435
- R. Sedgewick, Algorithmen, Addison-Wesley, 2002, Seiten 443 – 454
- M. de Berg et al., *Computational Geometry*, Springer, 2000.

Geometrische Algorithmen

Inhalt

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- **Geometrisches Divide-and-Conquer**
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Geometrische Algorithmen

Geometrisches Divide-and-Conquer

Prinzip von Divide-and-Conquer

- Divide-and-conquer (teile und herrsche) ist ein allgemeines rekursives Problemlösungsprinzip mit vielen Anwendungsmöglichkeiten
- Idee ist (wie der Name sagt)
  - ein großes Problem in kleinere Teilprobleme zu teilen (*Divide*)
  - die Teilprobleme zu lösen (*Conquer*)
  - und dann auf der Basis der Lösungen der Teilprobleme (in einfacherer Weise) eine Gesamtlösung zu erstellen (*Merge*)
- Im Folgenden ist das allgemeine Schema von Divide-and-conquer bei einem Problemraum S angegeben

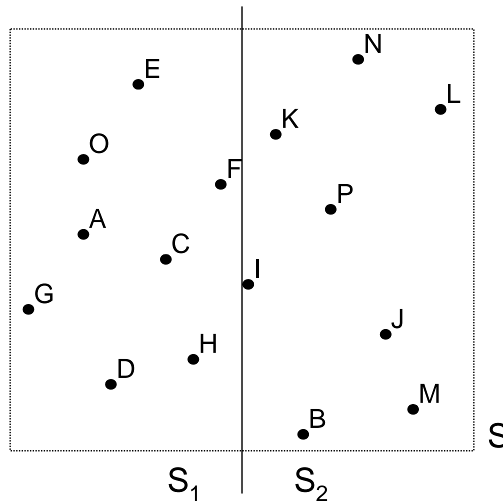
```
Loesung Divide-and-conquer (Problemraum S) {
  if (S ist klein) löse S direkt und gib Ergebnis zurück
  else {
        Divide: teile S in zwei Unterprobleme S1 und S2
        Conquer: löse S1 und S2 durch rekursive Aufrufe
            loesung1 = Divide-and-conquer (S1);
            loesung2 = Divide-and-conquer (S2);
        Merge: erstelle Gesamtlösung loesung für S
            unter Verwendung von loesung1 und loesung2
            return loesung
  }
}
```

Geometrisches Divide-and-Conquer

- Beim geometrischen Divide-and-conquer werden die geometrischen Objekte entlang einer vertikalen (oder horizontalen) Geraden in 2 (möglichst gleich große) Mengen von Objekten, die links bzw. rechts der Linie liegen, zerlegt.

Beispiel:

- Problemraum ist einer Reihe von Punkten
  S = {A, B, C, D, E, F, G, H, I, J, K, L, M, N, P}
- entlang einer vertikalen Linie werde die Punkte in zwei Mengen
  S1 = {A, C, D, E, F, G, H}
  S2 = {B, I, J, K, L, M, N, P}
  geteilt



Problem der nächsten Paare

- Problemstellung:
  - *Gegeben*: eine Menge von Punkten **points**
  - *Gesucht*: die minimale Distanz **min** zwischen allen Paaren von Punkten.

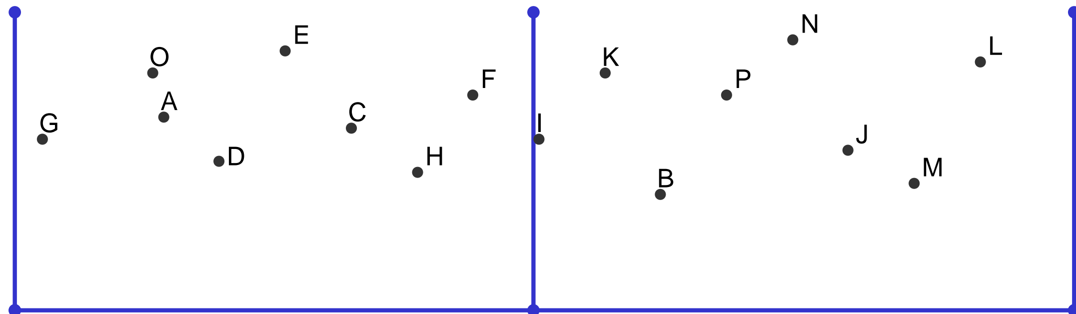
- Mit Divide-and-conquer läßt sich dieses Problem effizient lösen. Der Algorithmus arbeitet wie folgt:
  - teilt die Punktmenge **points** entlang von Teilungslinien **x** rekursiv in kleinere Mengen **points1** und **points2** bis die Mengen nur mehr ein Element beinhalten; für Mengen mit einem Element wird als minimale Distanz **unendlich** zurückgegeben
  - beim rekursiven Aufstieg werden die Distanzen für jene Punktpaare **p1**, **p2** berechnet,
    - deren eine Punkt **p1** in **points1** und der andere Punkt **p2** in **points2** liegt und
    - die nicht weiter als **min** von der Teilungslinie entfernt sind, wobei min die zur Zeit gefundene minimale Distanz ist (andere Punkte haben auf jeden Fall größere Distanz)
  - es wird als minimale Distanz das Minimum der Distanzen von **points1**, **points2** oder aller geprüften Paare zurückgegeben.

Problem der nächsten Paare: Algorithmus

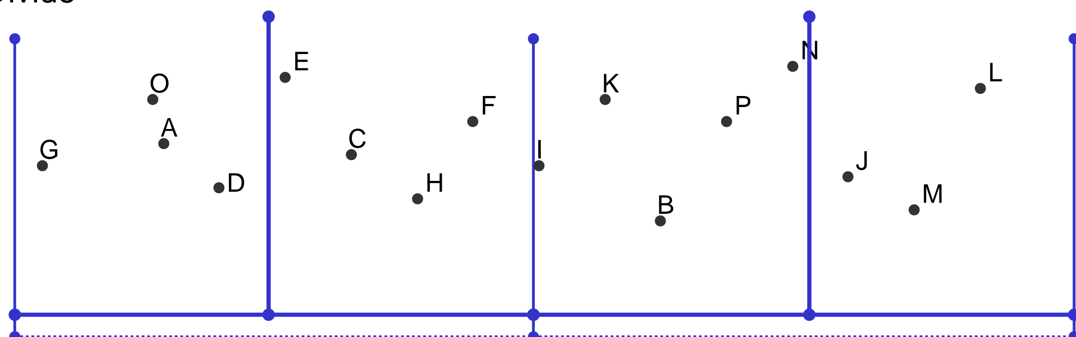
```
int closestPoints (Point[] points) {
    if (points.length <= 1) {
        // bei nur einem Element in points das Ergebnis unendlich
        return maxint;
    } else {
        Divide:
            berechne Teilungskoordinate x
                sodass sich points in zwei annähernd gleich große
                Teilmengen points1 und points2 teilt mit
                p.x <= x für alle p in points1
                p.x > x für alle p in points2
        Conquer:
            min1 = closestPoints(points1);
            min2 = closestPoints(points2);
        Merge:
            min = minimum {min1, min2}
            for all p1 in points1 mit p1.x > x - min {
                for all p2 in points2 mit p2.x < x + min {
                    if (distance(p1, p2) < min) {
                        min = distance(p1, p2)
                    }
                }
            }
            return min;
    }
}
```

Problem der nächsten Paare: Arbeitsblatt

Divide



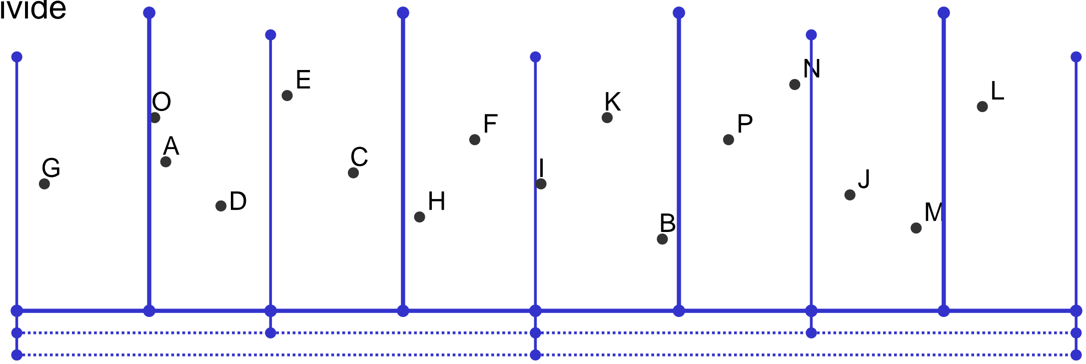
Divide



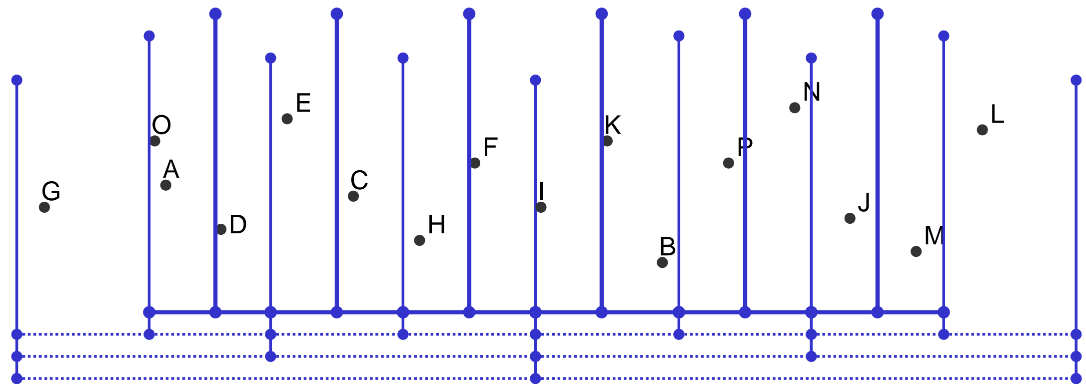
Problem der nächsten Paare: Arbeitsblatt

Geometrische Algorithmen
Geometrisches Divide-and-Conquer

Divide



Divide

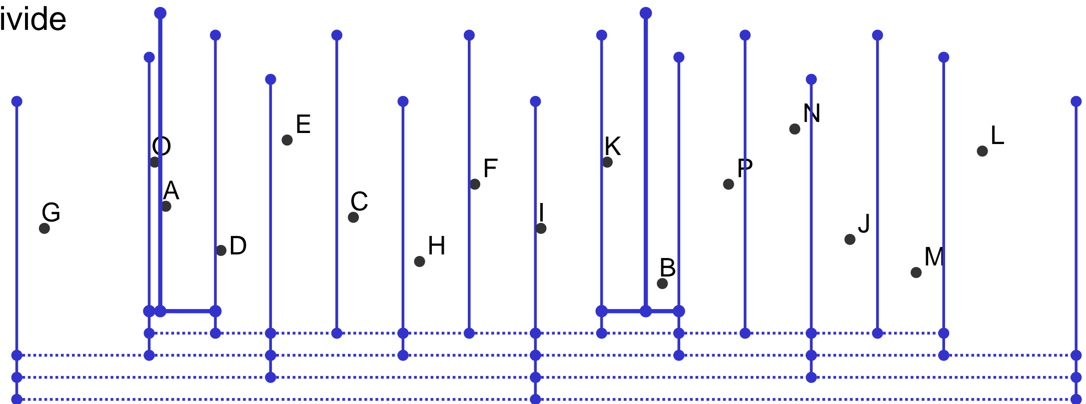


JOHANNES KEPLER UNIVERSITY LINZ
Research and teaching network

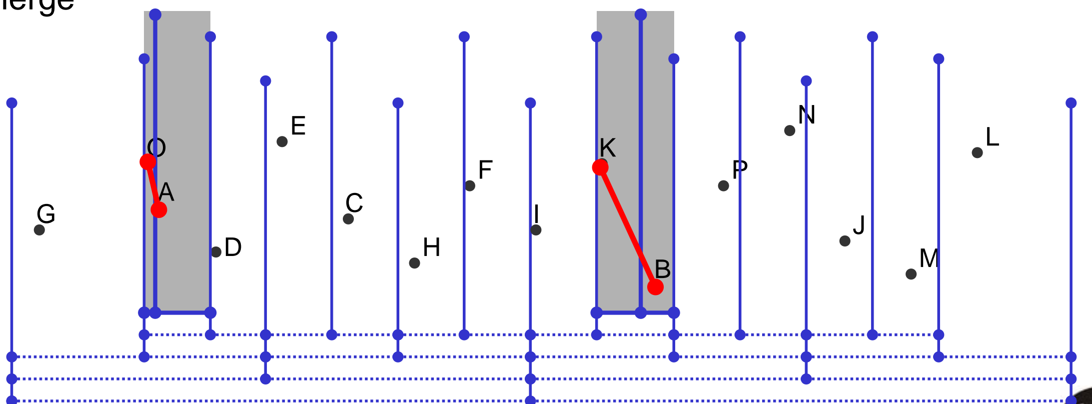
Problem der nächsten Paare: Arbeitsblatt

Geometrische Algorithmen
Geometrisches Divide-and-Conquer

Divide



Merge

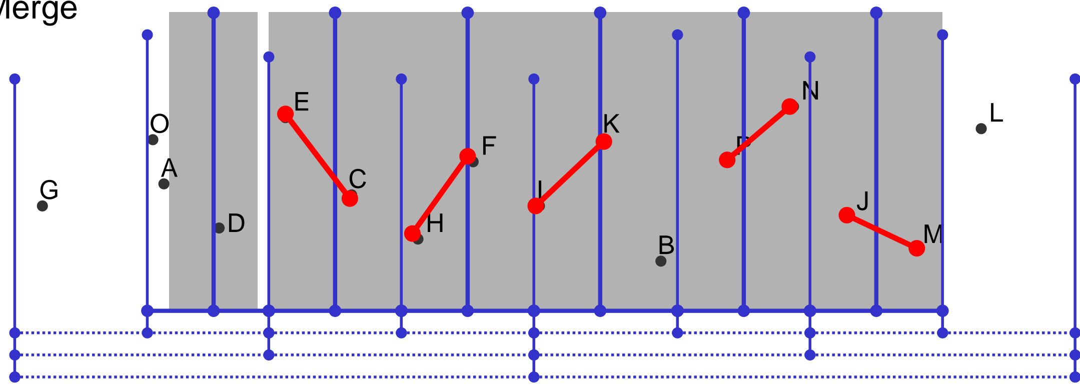


JOHANNES KEPLER UNIVERSITY LINZ
Research and teaching network

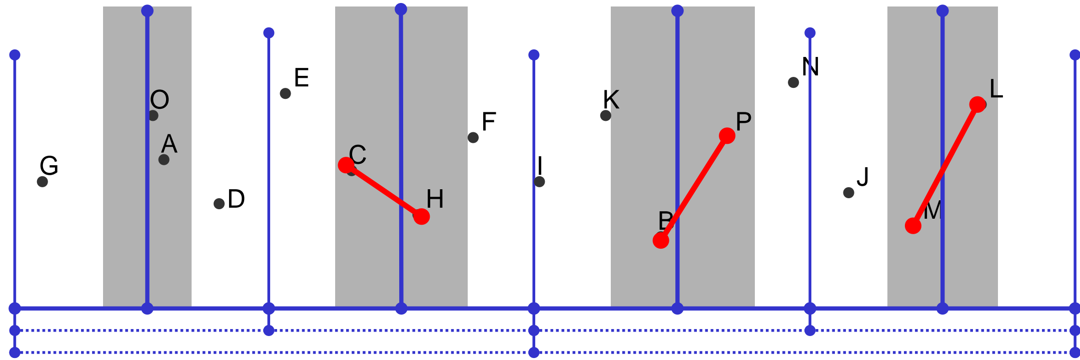
Problem der nächsten Paare: Arbeitsblatt

Geometrische Algorithmen
Geometrisches Divide-and-Conquer

Merge



Merge

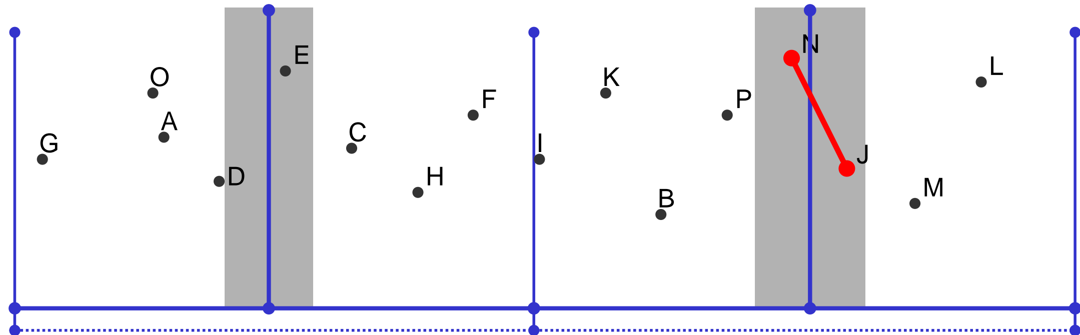


JOHANNES KEPLER UNIVERSITY LINZ
Research and teaching network

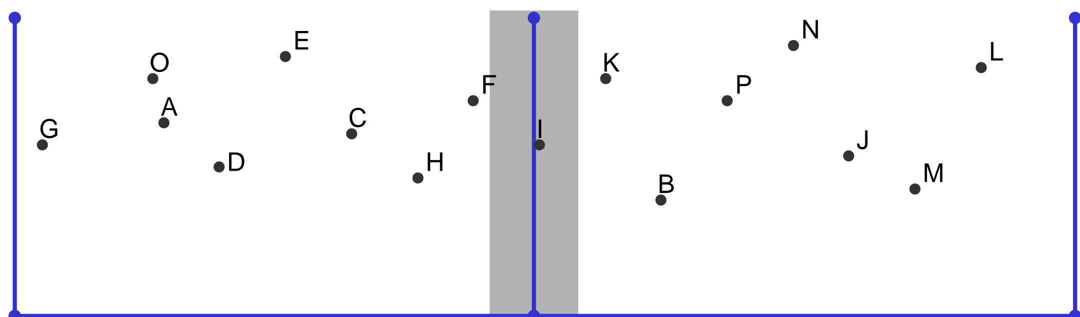
Problem der nächsten Paare: Arbeitsblatt

Geometrische Algorithmen
Geometrisches Divide-and-Conquer

Merge



Merge



JOHANNES KEPLER UNIVERSITY LINZ
Research and teaching network

Geometrisches Divide-and-Conquer: Weitere Anwendungen

Geometrische Algorithmen
Geometrisches Divide-and-Conquer

- In ähnlicher Weise lassen sich viele weitere Probleme nach dem Divide-and-Conquer-Prinzip effizient lösen, wie z.B.:
  - Linienschnittproblem von Liniensegmenten (siehe Scan-Line-Verfahren)
  - Schnittprobleme bei Rechtecken
  - Enthaltensein von Punkten in Rechtecken (*Inklusion*)

Literatur

Geometrische Algorithmen
Geometrisches Divide-and-Conquer

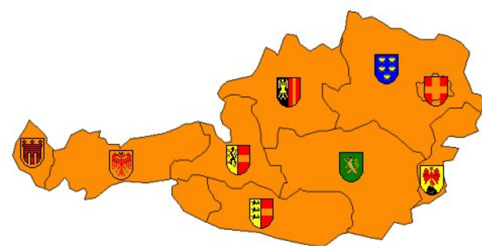
- R. Sedgewick, Algorithmen, Addison-Wesley, 2002, Seiten 457 - 465
- Ottmann, Wiedemayer, *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996, Seiten 435 - 444

Inhalt

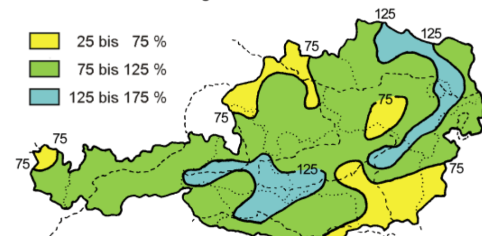
- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- **Gebietsräpresentation und Lokalisierung**
- Distanzprobleme und Voronoi-Diagramme

Gebietsrepästenation: Motivation

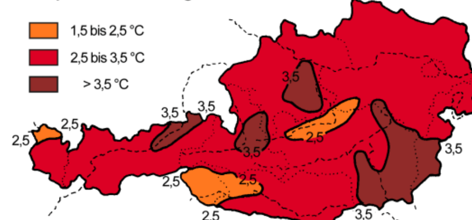
- Bei der digitalen Kartographie spielt die Repräsentation von Gebietsinformationen eine zentrale Rolle
  - Aufteilung der Landkarte in Staaten, Länder, politische Bezirke, etc.
  - Einteilung eines Gebiets nach der Landnutzung (Stadt, Felder, Wald, Wasser)
  - Klassifikation eines Gebiets nach bestimmten Merkmalen (z.B. Niederschlag, Vorkommen von Tieren, Bevölkerungsdichte, wirtschaftliche Entwicklung, ...)
  - etc.



Prozent des Niederschlagsnormalwertes Mai 2003

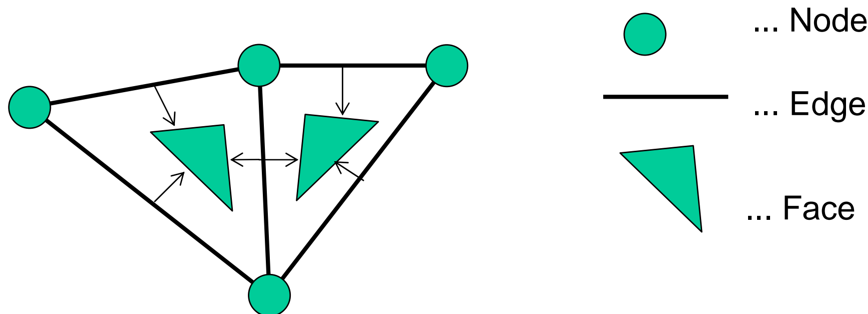


Temperaturabweichungen Mai 2003



Gebietsinformation: Einfache Realisierung mit erweiterten Graphen

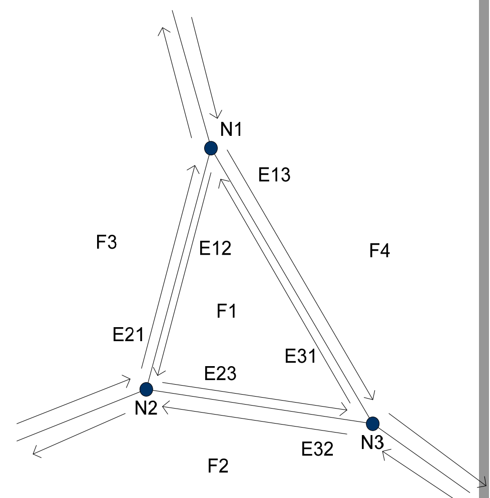
- Gebiete werden üblicherweise durch Polygone dargestellt
- einfache Realisierung als Graph + Gebietsobjekte
  - Knoten speichern Koordinaten
  - Kanten stellen Gebietsgrenzen dar und speichern zusätzlich die Verweise auf die 2 Gebietsobjekte, die sie trennen
  - Gebiete (faces) sind neue Datenobjekte und speichern
    - Liste von Kanten, die die Grenze des Gebiets darstellen
    - Informationen über die Gebiete



Doubly-Connected Edge List: Graphen mit 2 gerichteten Kanten

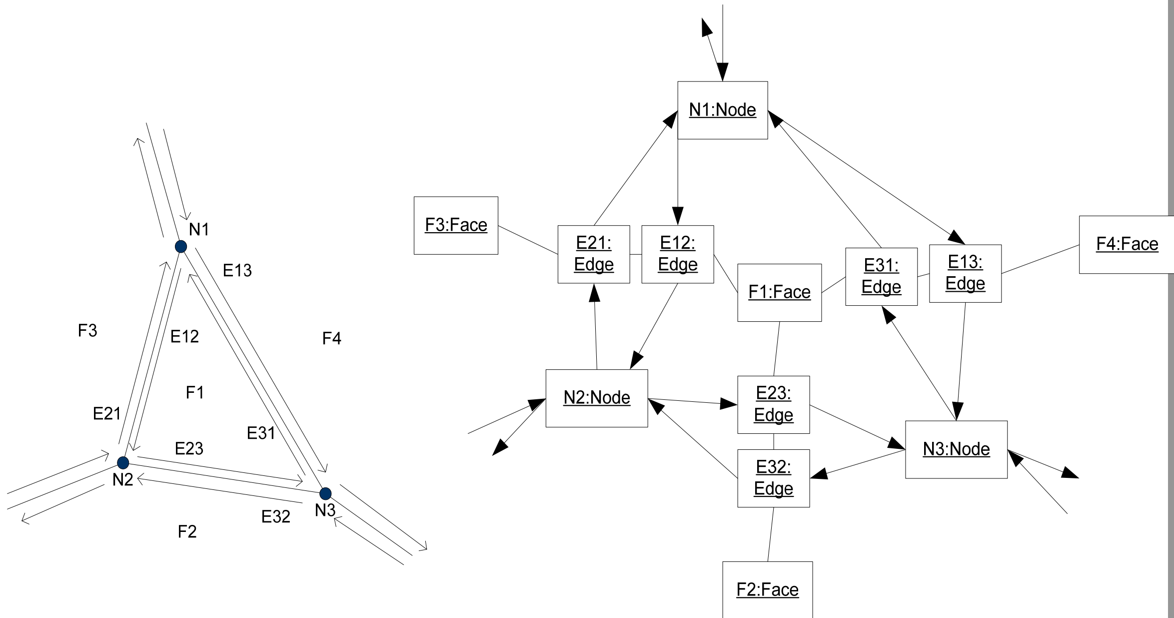
Doubly-Connected Edge List

- ist eine Erweiterung von Graphen für die Darstellung von Gebieten.
- Kanten trennen Gebiete und haben damit 2 Seiten.
- Deshalb trennt man eine Kante in 2 gerichtete Kanten für beide Seiten
- Man nennt die beiden zusammengehörigen Kanten ***Twins***
- und die *Twin*(*e*) einer Kante *e* verweist auf das benachbarte Gebiet.
- Damit ist jeder gerichteten Kante genau ein Gebiet zugeordnet und die Folge der Kanten bildet eine **geschlossene Grenze** für das Gebiet.
- Dabei sind die Kanten eines Gebiets so gerichtet, dass sie einen Kreis bilden und das Gebiet immer **zur linken Seite** der Kante liegt.



Geometrische Algorithmen

Doubly-Connected Edge List: Verkettete Objektstruktur



Geometrische Algorithmen
Gebietsrepräsentation

ADTs für Doubly-Connected Edge List

- ADT Edge unterstützt folgende Operationen:

  Node from() – Zugriff auf den Ausgangsknoten

  Node to() – Zugriff auf den Endknoten

  Face face() – Zugriff auf das Gebiet

  Edge twin() – Zugriff auf die parallele Kante des benachbarten Gebiets

  Edge pred() – Zugriff auf die Vorgängerkante der Grenze des Gebiets

  Edge succ() – Zugriff auf die Nachfolgekante der Grenze des Gebiets

ADTs für Doubly-Connected Edge List

- ADT Node unterstützt folgende Operationen

  Point getPosition() – Zugriff auf die Position des Knotens

  Edge[] getOutEdges() – Zugriff auf alle wegführenden Kanten

  Edge[] getInEdges() – Zugriff auf alle ankommenden Kanten

  Face[] getFaces() – Zugriff auf alle angrenzenden Gebiete

  Edge getOutEdgeOfFace(Face face) – Zugriff auf die wegführende Kante für ein bestimmtes Gebiet

  Edge getInEdgeOfFace(Face face) – Zugriff auf die ankommenden Kante für ein bestimmtes Gebiet

Geometrische Algorithmen
Gebietsrepräsentation

ADTs für Doubly-Connected Edge List

- ADT Face unterstützt folgende Operationen:

  Edge getFirstEdge() – Zugriff auf die erste Kante der Grenze

  Edge[] getBorder() – Zugriff auf die sortierte Menge der Kanten, welche die Grenze des Gebiets darstellt

  Face[] getNeighbors() – Zugriff auf alle Nachbargebiete

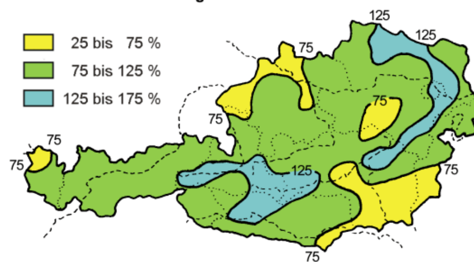
  Face getNeighbor(Edge e) – Zugriff auf das Nachbargebiet für eine Kante

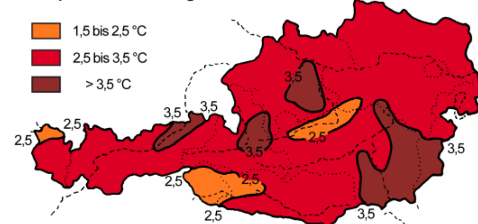
  Object getInfo() – Zugriff auf die spezielle Gebietsinformation

Achtung: Hier ist nur *eine Aussengrenze* angenommen. Sollen auch Innengrenzen (= Grenzen zu eingeschlossenen Nachbargebieten) berücksichtigt werden, gibt es nicht nur eine sondern eine *Menge* von Grenzen!!

Überlagerung von Gebietsinformationen

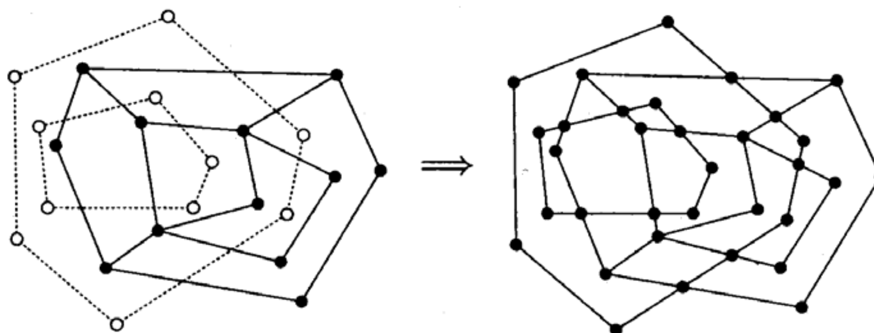
- Die Überlagerung von mehreren Gebietsinformationen ist eine wesentliche Operation in der Kartographie
- In dieser Weise lassen sich unterschiedliche geographische Informationen zusammenführen
- Beispiele:
  - Verteilung von Niederschlag und Vorkommen bestimmter Vegetation
  - Landnutzung und Vorkommen einer bestimmten Tierpopulation
  - Bevölkerungsdichte und wirtschaftliche Entwicklung
  - ...





Überlagerung von Gebietsinformationen: Vorgehen

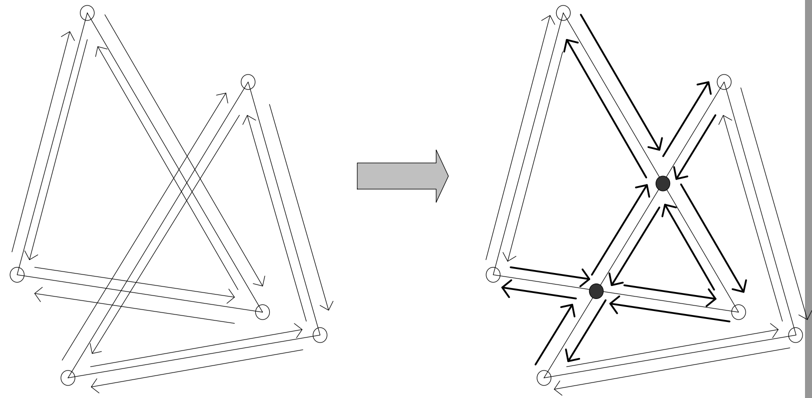
- Bei der Überlagerung von Gebieten stellt der Schnitt von Liniensegmenten (wie besprochen) die wesentliche Operation dar.
- Vorgehen:
  1. Berechnung alle Schnitte der Grenzlinien (Kanten) der beiden Gebietsstrukturen (wie beim allgemeinen Linienschnittproblem besprochen)
  2. Bei Vorkommen eines Schnittes von 2 Grenzlinien müssen die Kanten der Doubly-Connected Edge List entsprechend angepasst werden (siehe nächste Folie).
  3. Im Nachhinein wird mit einem Scan-Line-Verfahren nach neuen geschlossenen Grenzlinien gesucht, die neue Gebiete (Faces) darstellen, und diese als neue Faces in die Doubly-Connected Edge List eingefügt.



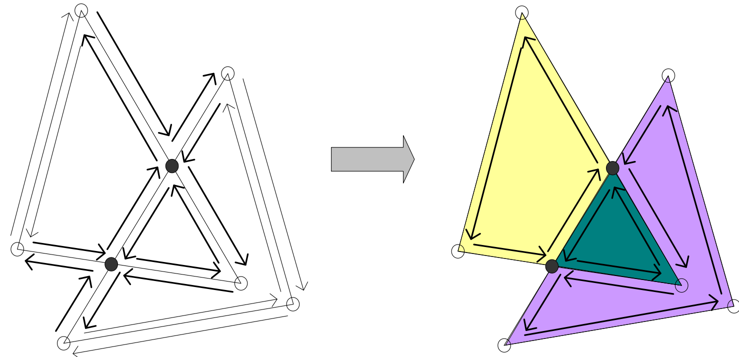
Überlagerung: Update der Doubly-Connected Edge List

Geometrische Algorithmen
Überlagerung von Gebietsinformationen

- Beispiel Update bei Schnitt der Kanten



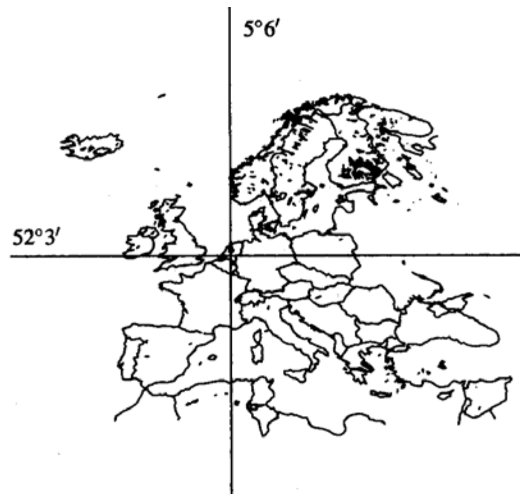
- Finden neuer geschlossener Kantenzüge und Einfügen neuer Gebiete (Faces)



Lokalisierung eines Punktes

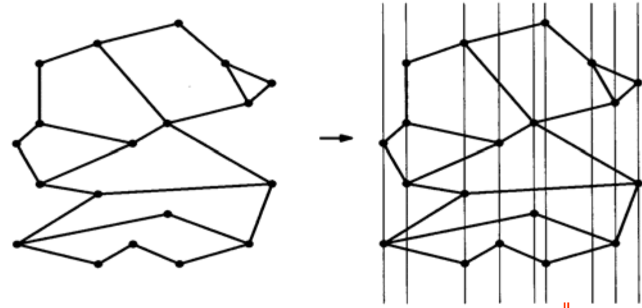
Geometrische Algorithmen
Punkt-Lokalisierung

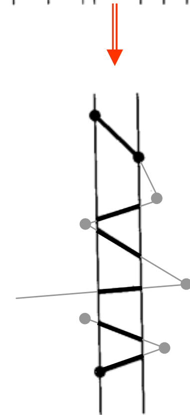
- Eine weitere wichtige Operation auf Gebieten ist das Finden des Gebietes, in welchem ein bestimmter Punkt liegt (*Point Location*)
  - Gegeben: ein Ort in der Form von Koordinaten
  - Gesucht: das Land (Gebiet), in dem dieser Punkt liegt.
- Die algorithmische Herausforderung ist, aus einer großen Anzahl von komplexen Gebieten das gesuchte Gebiet zu finden
- Ansatz ist der Aufbau von Suchstrukturen zur binären Suche bzw. Aufbau von speziellen Suchbäumen



Lokalisierung eines Punktes: 2-dimensionale binäre Suche

- Bei der 2-dimensionalen binären Suche wird der Suchraum zuerst nach den Eckpunkten in vertikale Streifen eingeteilt





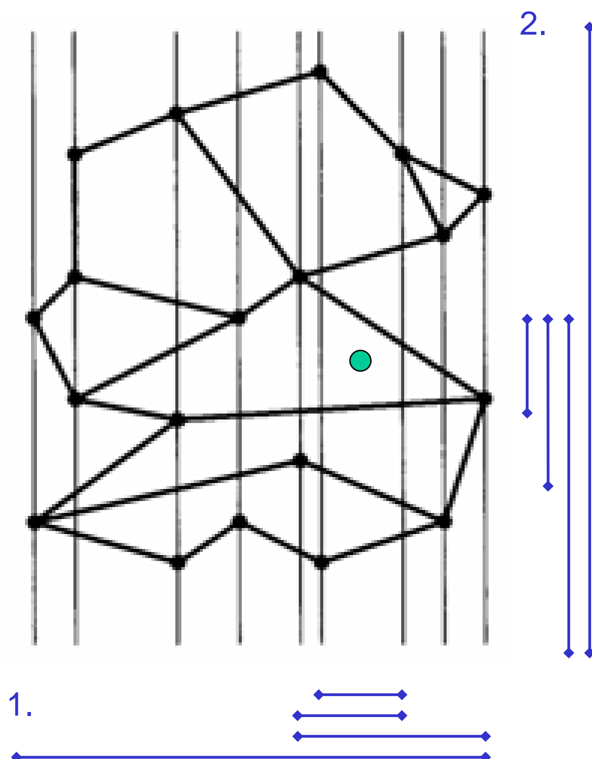
- Dadurch entstehen Streifen mit trapezförmigen Ausschnitten aus Gebieten

Lokalisierung eines Punktes: 2-dimensionale binäre Suche

Geometrische Algorithmen
Punkt-Lokalisierung

- Man baut nun eine 2-dimensionale Suchstruktur auf
  - sortierte Liste von Streifen
  - jeder Streifen sortierte Liste von Trennlinien von Gebieten
- In dieser Struktur kann man zweimal binär Suchen
  1. zuerst binäre Suche nach dem Streifen
  2. dann binäre Suche nach dem Gebiet im Streifen (oberhalb oder unterhalb des Segments)

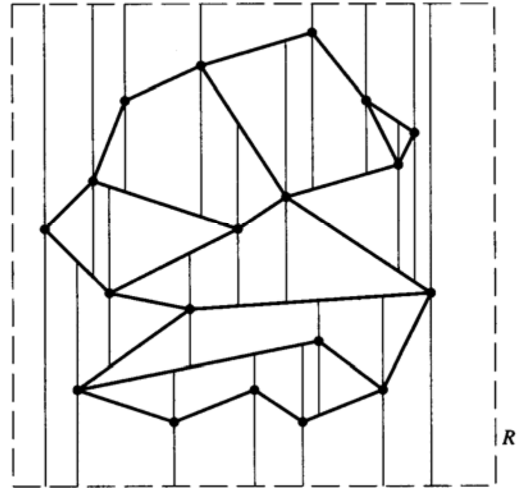
kann mit ccw gelöst werden



<u>Nachteil des Verfahrens</u>: Speicheraufwand kann gewaltig sein !!!

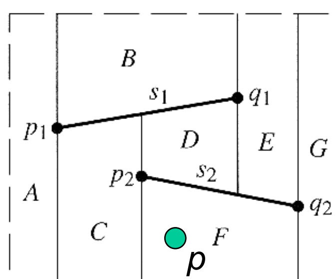
Lokalisierung eines Punktes: Trapezoidal Maps

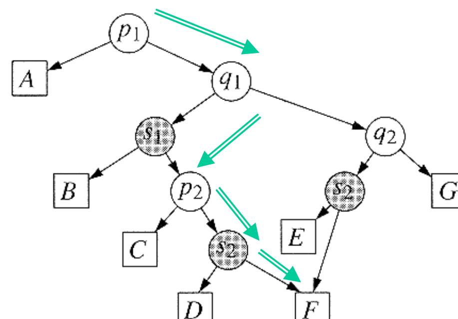
- Ein weiteres Verfahren zur Punkt-Lokalisierung ist
  - die Aufteilung der Gebiete nach *Trapezoidal Maps*
  - und der Aufbau eines *hierarchischen binären Suchbaums* nach dieser Einteilung
- Bei einer Aufteilung in Trapezoidal Maps geht man folgend vor:
  - jeder Eckpunkt wird nach oben und nach unten bis zum nächsten Segment verlängert



Lokalisierung eines Punktes: Suchbaum

- die dadurch entstehenden Teilgebiete werden für den Aufbau des Suchbaums verwendet
  - *Blattknoten* im Suchbaum stellen die Teilgebiete dar
  - Innere Knoten entweder Punkte und ihre vertikalen Trennlinien (*Punktknoten*)
  - oder Liniensegmente (*Segmentknoten*)
- Suche nach einem Punkt *p* erfolgt, dass man
  - bei Punktknoten unterscheidet, ob der Punkt *p* links oder rechts vom Punktknoten liegt
  - bei einem Segmentknoten unterscheidet, ob der Punkt *p* oberhalb oder unterhalb des Segments liegt
  - bis man bei Blattknoten angelangt ist





Literatur

- M. de Berg et al., *Computational Geometry*, Springer, 2000.

Inhalt

- Geometrische Figuren
- Schnitt zweier Strecken
- Enthaltensein in einem Polygon
- Konvexe Hülle
- Geometrische Datenstrukturen und Bereichssuche
- Scan-Line-Methode und geometrische Schnitte
- Geometrisches Divide-and-Conquer
- Gebietsräpresentation und Lokalisierung
- Distanzprobleme und Voronoi-Diagramme

Motivation

Geometrische Algorithmen
Nächster Punkt und Voronoi-Diagramme

- Ein oft auftretendes Problem ist in einer gegebenen, festen Punktmenge (Basisstationen) für einen neuen Punkt (aktuelle Position) den nächsten Punkt zu finden
- Das Voronoi-Diagramm ist eine geometrische Struktur, die die effiziente Berechnung des nächsten Punktes ermöglicht

Definition Voronoi-Polygon: Die Menge aller Punkte, die von einem gegebenen Punkt in einer Punktmenge einen geringeren Abstand haben als von allen anderen Punkten in der Punktmenge.

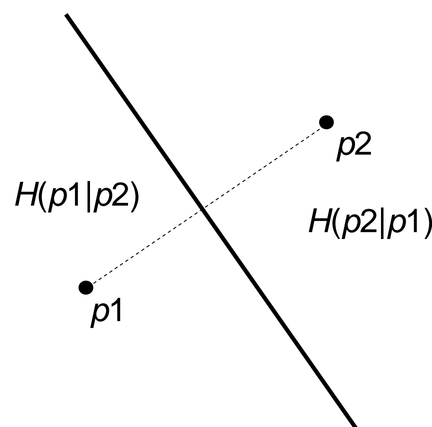
Definition Voronoi-Diagramm: Vereinigung aller Voronoi-Polygone einer Punktmenge.

Voronoi-Polygon bei 2 Punkten

Geometrische Algorithmen
Nächster Punkt und Voronoi-Diagramme

Veranschaulichung für Punktmenge mit 2 Punkten *p*1 und *p*2:

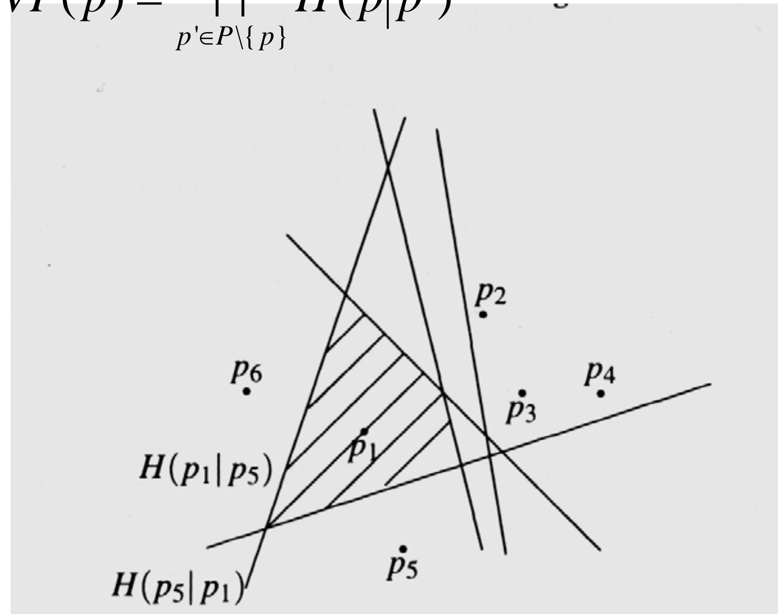
- Mittelsenkrechte auf die Verbindungsstrecke von *p*1 und *p*2 teilt die Ebene in 2 Teile:
  - *H*(*p*1|*p*2) ist die Halbebene der Punkte, die näher zu *p*1 als zu *p*2 liegen
  - *H*(*p*2|*p*1) ist die Halbebene der Punkte, die näher zu *p*2 als zu *p*1 liegen



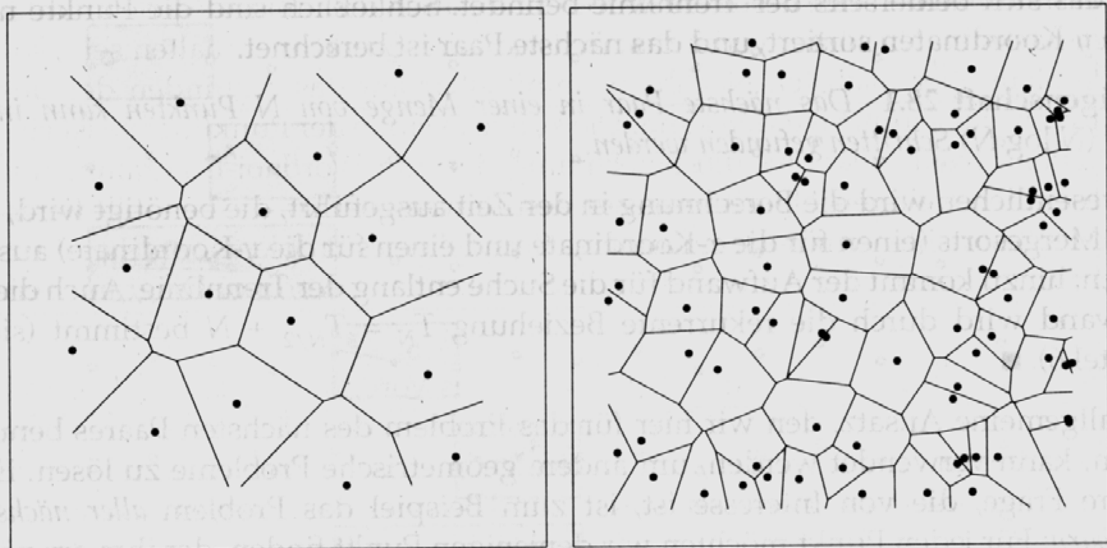
Voronoi-Polygon für beliebige Punktmengen

- Geht man von einer beliebigen Punktmenge *P* aus, so ergibt sich das Voronoi-Polygon *VP*(*p*) als der Durchschnitt aller Halbebenen *H*(*p*|*p*')

$$VP(p) = \bigcap_{p' \in P \setminus \{p\}} H(p|p')$$

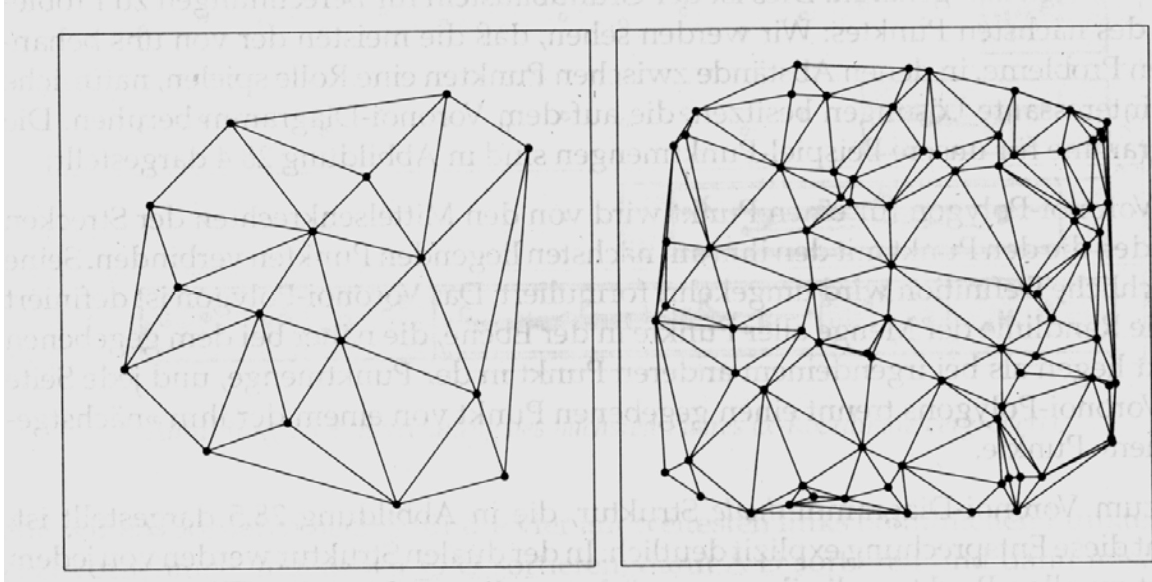


Voronoi-Diagramm: Beispiele



Delaunay-Triangulation

- Die Delaunay-Triangulation ist die zum Voronoi-Diagramm duale Struktur
  - verbindet nächste Punkte miteinander, wenn sie eine Kante im Voronoi-Diagramm teilen



Problemlösungsverfahren

- Sind Voronoi-Diagramm und Delaunay-Triangulation für eine gegebene Punktmenge einmal berechnet, lassen sich eine Menge von Problemen effizient lösen, z. B.:
  - Konvexe Hülle
  - Nächster Punkt

- Nächster Punkt:
  - *gegeben*: Punktmenge P und neuer Punkt q
  - *gesucht*: $p \in P$ welcher q am nächsten ist
  - *Lösung*: Suche $VP(p)$ mit $q \in VP(p)$

Näheres zu Voronoi-Diagrammen siehe: Ottmann, Wiedemayer, *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996, Seiten 501-533

Zusammenfassung

Geometrische Algorithmen

- Geometrische Problemstellungen treten in vielen Bereichen auf
  - CAD
  - Geographie
  - Schaltungsentwurf
  - Graphik
  - ...
- Für Menschen einfache Probleme, müssen durch komplexe Algorithmen gelöst werden
- Viele Verfahren wurden in jüngster Zeit entwickelt
  - allgemeine Problemlösungsstrategien
  - spezielle Algorithmen
  - effiziente Datenstrukturen
- noch immer aktuelles Forschungsgebiet der Informatik/Mathematik (*Computational Geometry*)

Literatur

Geometrische Algorithmen

- R. Sedgewick, Algorithmen, Addison-Wesley, 2002, Seiten 465 - 468
- Ottmann, Wiedemayer, *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996, Seiten 501-533