3. Das Sweep-Verfahren

Sweep-Verfahren (Scan-Verfahren, Scanlinien- / Scanebenen-Verfahren):
wichtige algorithmische Technik / Paradigma

analog zu "divide and conquer"

- Prinzip: Rückführung eines *n*-dim. Problems auf ein (*n*–1)-dimensionales
- dies wird erkauft durch Einführung einer Zeitdimension

erste Beispiele:
Sweep im Eindimensionalen (nach Klein 1997)

Beispiel 1:
Maximum einer total geordneten Menge von Objekten

gegeben: *n* Objekte q_1, ..., q_n aus einer Menge *Q* mit totaler Ordnungsrelation ">", gesucht ist das Maximum dieser Elemente.
Optimales Verfahren (Laufzeit $\Theta(n)$): Alle Objekte durchlaufen, jedes testen ob größer als das bisher größte.

```
MaxSoFar := q[1];
for j := 2 to n do
    if MaxSoFar < q[j]
    then MaxSoFar := q[j];
write("Das Maximum ist ", MaxSoFar)
```

dieses ist "Sweep": die Objekte werden der Reihe nach besucht, dabei wird eine bestimmte Information über die schon besuchten Objekte mitgeführt und jeweils aktualisiert.

Zeitdimension: for-Schleife
Aus 1-dim. statischen Problem wird 0-dim. dynamisches Problem.

Beispiel 2:
Dichtestes Paar von n reellen Zahlen $x_1, ..., x_n$ (*closest pair*).

1. Schritt: Sortieren der n Zahlen (Zeitaufwand $O(n \log n)$).
Sortierte Folge: $x_1' \leq ... \leq x_n'$.
2. Schritt: Zahlen in dieser Reihenfolge durchlaufen, Abstand jeweiliger Nachbarn prüfen, vergleichen mit bisherigem Minimalabstand (*MinDistSoFar*).

```
MinDistSoFar := x'[2] − x'[1];
ClosPos := 2;
for j := 3 to n do
    if MinDistSoFar > x'[j] − x'[j − 1]
    then
        MinDistSoFar := x'[j] − x'[j − 1];
        ClosPos := j;
write("Ein dichtestes Paar bilden ",
                    x'[ClosPos − 1], x'[ClosPos])
```

(*ClosPos* merkt die Position, an der das aktuell dichteste Paar auftritt.)

$\Rightarrow$ Zeitaufwand des 2. Schritts: linear in n
$\Rightarrow$ Gesamtlaufzeit dieser Lösung ist $O(n \log n)$.
(das ist optimal, siehe letztes Korollar in Kapitel 2.6.)

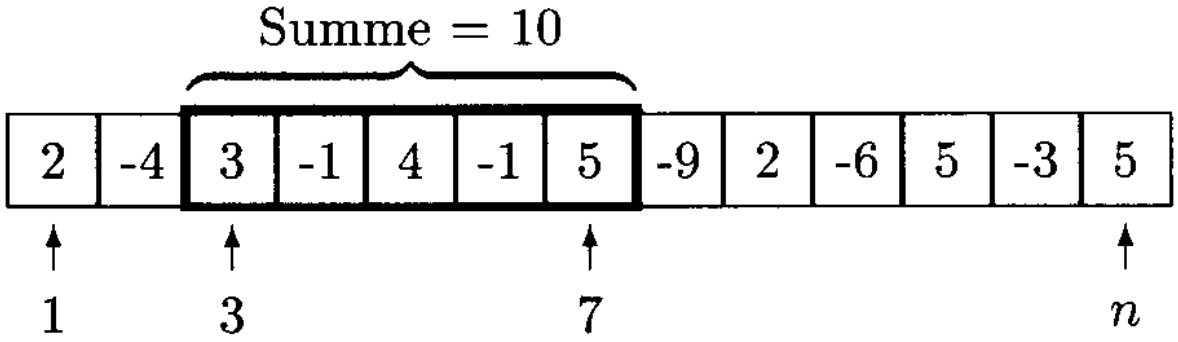
Beispiel 3 (schon nichttrivial):
Bestimmung der **maximalen Teilsumme** aus einer gegebenen Zahlenfolge (*maximum subvector problem*)

Motivation: Aktienkurse
gegeben: Folge der Kursdifferenzen für die Tage 1, 2, ..., n.
Wieviel hätte man maximal verdienen können, wenn man die Aktie am richtigen Tag gekauft und am richtigen Tag wieder verkauft hätte?

Variation[*i*] sei die Kursdifferenz am *i*-ten Tag.

Max. Gewinn ist: *Variation*[*i*] + *Variation*[*i*+1] + ... + *Variation*[*j*] für die "richtige" Kombination (*i*, *j*), für die also diese Summe maximal wird.

Beispiel:



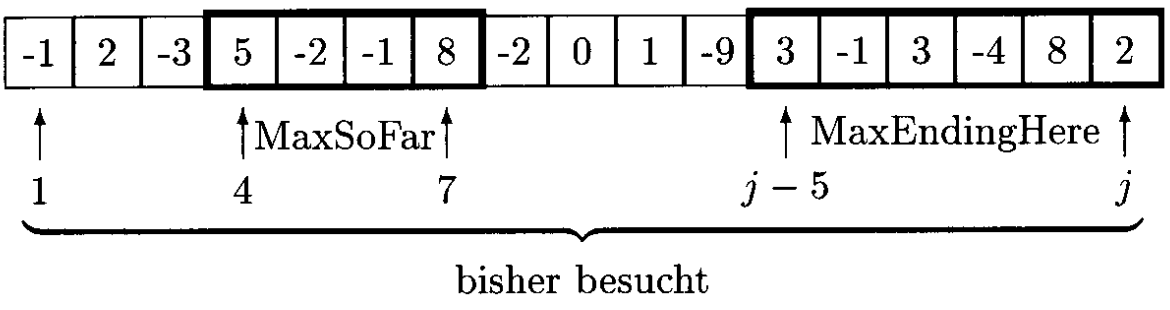
Max. Teilsumme (10) hier vom 3. bis 7. Tag.

Naiver Algorithmus: 3 for-Schleifen für *i*, *j* und Summationsindex *k*, Laufzeit $\Theta(n^3)$.

Einfache Verbesserung: nach jeder Erhöhung von *j* nicht die ganze Summe neu berechnen, sondern nur zur alten Summe den neuen Summanden *Variation*[*j*+1] addieren
$\Rightarrow$ Laufzeit $\Theta(n^2)$.

Behauptung: mit Sweep-Technik geht es in optimaler Zeit $\Theta(n)$!

Objekte *Variation*[1], *Variation*[2], ... *Variation*[*n*]
werden in dieser Reihenfolge besucht (linearer Aufwand).
Max. Teilsumme der bisher besuchten *j*–1 Zahlen:
$MaxSoFar_{j-1}$.
Wenn der nächste (*j*-te) Eintrag besucht wird: dieses bisherige Max. kann nur übertroffen werden durch Teilsummen, die den *j*-ten Eintrag einschließen.
Beispiel: bisherige max. Teilsumme 10, wird ersetzt durch 11:



Als weitere Information wird nun beim Sweep mitgeführt:
max. Teilsumme der schon betrachteten Einträge, die beim zuletzt neu hinzugekommenen Eintrag endet:

MaxEndingHere$_j$ sei das Max. aller Summen
Variation[*h*] + *Variation*[*h*+1] + ... + *Variation*[*j*]
mit *h* zwischen 1 und *j*, falls wenigstens eine dieser Summen nichtnegativ, sonst 0.
Das *h*, für das dieses Max. angenommen wird, ist entweder *h* = *j* (dann ex. keine pos. Teilsumme, die in j–1 endet, und *MaxEndingHere*$_{j-1}$ war 0), oder dasselbe *h* wie im Fall *j*–1.
$\Rightarrow$
Aktualisierungsvorschrift für *MaxEndingHere*$_j$:

MaxEndingHere$_j$ = max(0, *MaxEndingHere*$_{j-1}$ + Variation[*j*]).

Damit schneller + eleganter Algorithmus für die max. Teilsumme:

```
MaxSoFar := 0;
MaxEndingHere := 0;
for j := 1 to n do
    MaxEndingHere :=
                max(0, MaxEndingHere + Variation[j]);
    MaxSoFar := max(MaxSoFar, MaxEndingHere);
write("Die maximale Teilsumme beträgt ", MaxSoFar)
```

Beachte:
Implementation auch als nicht-terminierendes Verfahren (online-Verfahren) möglich, da jeder neue Eintrag nur einmal gebraucht wird.
Speicherplatzbedarf und tägl. Antwortzeit im online-Betrieb sind dann konstant.

Sweep in der Ebene (plane sweep) (nach Hinrichs 2001)

wandernde, meist senkrechte Gerade (*sweep line*) "fegt die Ebene von links nach rechts aus".

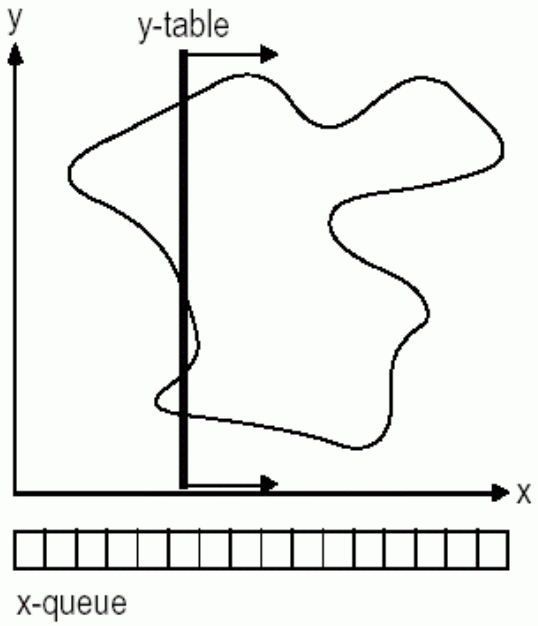
Vorteile gegenüber "divide and conquer"-Verfahren:

- Plane-sweep: Einfaches algorithmisches Paradigma zur Lösung geometrischer Probleme in der Ebene
- Plane-sweep Algorithmen sind iterativ und vermeiden somit den Aufwand an Zeit und Speicherplatz, um den bei Divide & Conquer Algorithmen auf Grund der Rekursion notwendigen Stack zu verwalten.
- Plane-sweep Algorithmen arbeiten inkrementell und haben daher den Vorteil, daß eine Aktualisierung des gegenwärtigen Zustandes einfacher ist, wenn man nur einen neuen Punkt hinzufügt, als wenn man zwei Mengen von jeweils n/2 Punkten miteinander verknüpfen muß.

Prinzip des Sweep-Verfahrens in der Ebene:

- Plane-sweep bewegt vertikale Linie von links nach rechts über die Ebene und überstreicht dabei eine gegebene geometrische Konfiguration, für die ein Problem gelöst werden soll.
- Die Linie stoppt in jedem Transitionspunkt, auch Ereignis genannt, in dem etwas Relevantes bzgl. der Problemstellung geschieht. Die gesamte Verarbeitung vollzieht sich an dieser Front.
- *x-queue* speichert die Transitionspunkte, an denen der Algorithmus anhalten muß.
- *y-table* speichert den Status des Sweeps, d.h. die angesammelten Informationen, die für die Lösung des Problems relevant sind.
- In dem Streifen zwischen zwei Ereignissen ändern sich die für die Problemstellung relevanten Eigenschaften der Konfiguration nicht, und daher muß die y-table nicht aktualisiert werden.

Die *y*-table heißt auch "Sweep-Status-Struktur" (SSS).



- Plane-Sweep Gerüst

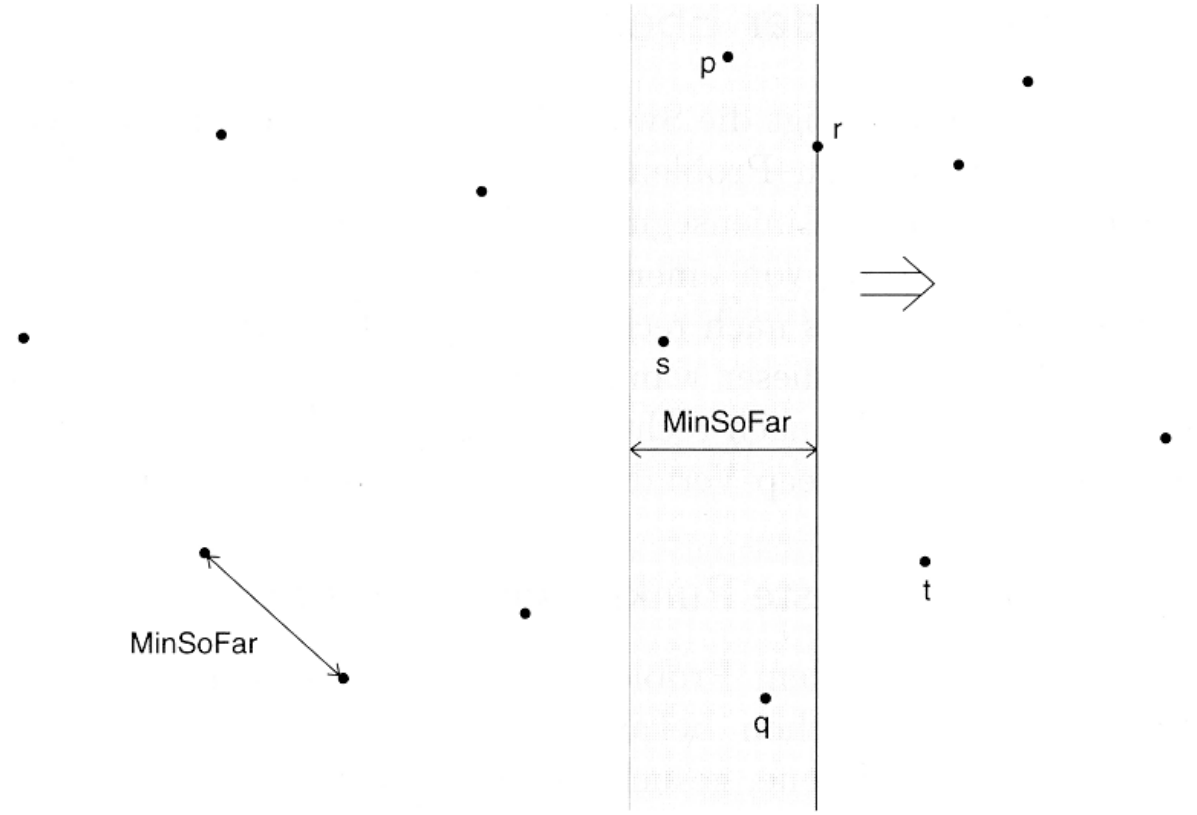
```
initXqueue;
initYtable;
while not emptyX do
      transition(nextX);
```

- *nextX* gibt das nächste zu verarbeitende Ereignis aus der x-queue zurück.
- *emptyX* testet, ob die x-queue leer ist.
- *transition* umfaßt die gesamte Arbeit, die verrichtet werden muß, wenn ein neues Ereignis angetroffen wird; sie bewegt die Front von dem Streifen zur Linken eines Ereignisses zu dem Streifen unmittelbar rechts von diesem Ereignis.

Beispielproblem: dichtestes Punktepaar in der Ebene (*closest pair*)

gegeben: n Punkte p_1, ..., p_n in der Ebene
gesucht: Paar mit minimalem Abstand (bzgl. Metrik d_k).

Im Unterschied zum 1-dim. Problem kann der Abstand eines mit der *sweep line* neu erreichten Punktes (*r* = *nextX*) zu seinem direkten Vorgänger in der Xqueue (*q* in der Abb.) größer sein als der Abstand zu noch weiter links liegenden Punkten (*p* in der Abb.):



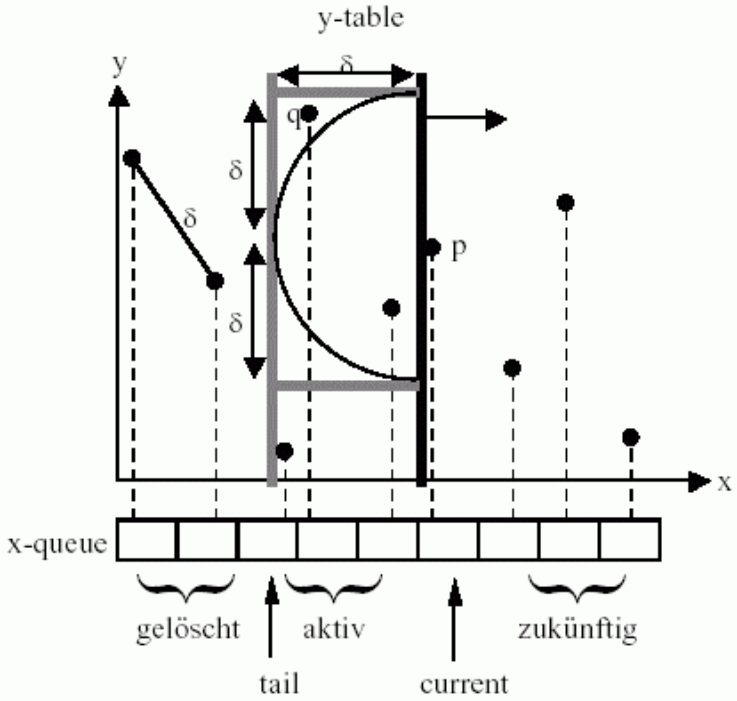
(aus Klein 1997).

Punkte mit Abstand > MinSoFar hinter der *sweep line* sind aber uninteressant.

⇒ Verzeichnis muss die Punkte im gekennzeichneten Streifen speichern (Sweep-Status-Struktur, *y*-table)

Die folgenden Ereignisse erfordern eine Aktualisierung der Sweep-Status-Struktur:

1. Der linke Streifenrand wandert über einen Punkt hinweg.
2. Der rechte Streifenrand (*sweep line*) stößt auf neuen Punkt.



- *x-queue* speichert die Punkte der Menge S, geordnet nach ihren x-Koordinaten, als Ereignisse, die während des Ebenendurchlaufs bearbeitet werden müssen.
- Zwei Zeiger in die x-queue, nämlich *tail* und *current*, partitionieren S in vier disjunkte Teilmengen:
  1. Auf die *gelöschten* Punkte zur Linken von tail wird nicht mehr zugegriffen.
  2. Die *aktiven* Punkte zwischen tail (einschließlich) und current (ausschließlich) werden für die weitere Verarbeitung noch benötigt.
  3. Der Transitionspunkt p wird gegenwärtig bearbeitet.
  4. Die *zukünftigen*, rechts von der Front liegenden Punkte sind noch nicht betrachtet worden, sie werden irgendwann in der Zukunft bearbeitet werden.
- y-table speichert nur die aktiven Punkte gemäß ihrer y-Koordinaten.

- Initialisiere die y-table mit den beiden am weitesten links liegenden Punkten, d.h. den ersten beiden Punkten in der x-queue, die somit aktiv sind. δ wird mit ihrem Abstand initialisiert.
- Der Ebenendurchlauf beginnt mit dem dritten Punkt in der x-queue.
- Motivation für Unterscheidung zwischen gelöschten und aktiven Punkten: Beim Antreffen eines neuen Punktes p will man wissen, ob dieser Punkt mit irgendeinem Punkt zu seiner Linken ein neues Paar mit kleinstem Abstand bildet. Merke ein solches bisher gefundenes Paar mit kleinstem Abstand zusammen mit der minimalen Distanz δ.
- Alle Kandidaten, die mit dem Punkt p ein neues Paar mit minimalem Abstand bilden können, liegen in dem linken Halbkreis um p mit Radius δ.
- Halbkreisabfrage komplex $\Rightarrow$ ersetze Halbkreisabfrage durch effizientere Rechtecksabfrage, d.h. Halbkreis $\rightarrow$ Container
- Implementierung der Rechtecksabfrage in zwei Schritten:
  1. Deaktiviere alle Punkte zur Linken der Front, die einen Abstand $\geq \delta$ von der Front haben. Diese Punkte liegen zwischen 'tail' und 'current' in der x-queue und können leicht entfernt werden, indem man 'tail' nach rechts bewegt und die Punkte in der y-table löscht.
  2. Betrachte nur die Punkte q im δ-Streifen, deren vertikaler Abstand von p höchstens δ beträgt: $|q_y - p_y| < \delta$. Diese Punkte findet man in der y-table, indem man an der durch die y-Koordinate von p festgelegten Position startet und die y-table nach oben und unten absucht.

- Über eine Transition hinweg müssen die folgenden Invarianten aufrechterhalten werden:
  1. δ ist die minimale Distanz, die zwischen einem Paar von bisher bearbeiteten Punkten gefunden wurde ("gelöscht" oder "aktiv").
  2. Die aktiven Punkte (in der x-queue zwischen 'tail' und 'current' und in der y-table gemäß der y-Koordinate gespeichert) sind genau die, die im Innern eines δ–Streifens zur Linken der Front liegen.

- Die Bearbeitung eines Transitionspunktes umfaßt drei Schritte:
  1. Entferne alle Punkte q mit $q_x \leq p_x - \delta$ aus der y-table. Man findet sie, indem 'tail' nach rechts bewegt wird.
  2. Füge p in die y-table ein.
  3. Finde unter den Vorgängern und Nachfolgern von p in der y-table alle Punkte q mit $|q_y - p_y| < \delta$. Falls solch ein Punkt gefunden wird und seine Distanz von p kleiner als δ ist, aktualisiere δ und das bisher gefundene Paar mit minimaler Distanz.

Implementierung: (nach Hinrichs 2001)

- *x-queue* enthält alle Punkte gemäß $\leq_x$ sortiert:

  xQueue: array[1 .. maxN] of point;

- $closest_1$ und $closest_2$ beschreiben das bisher gefundene Paar von Punkten mit minimalem Abstand δ:

  $closest_1$, $closest_2$: point;

- n ist die Gesamtanzahl der zu verarbeitenden Punkte, t und c bestimmen die Position von *tail* und *current* in der x-queue:

  t, c, n: 1 .. maxN;

- *initX* speichert alle Punkte in der *x-queue* sortiert bezüglich $\leq_x$:

  procedure initX;

- *initY* erzeugt leere *y-table*:

  procedure initY;

- Ein neuer Punkt wird in die y-table eingefügt durch:

  procedure insertY(p: point);

- Ein neuer Punkt wird in der y-table gelöscht durch:

  procedure deleteY(p: point);

- Der Nachfolger eines Punktes p in der y-table bezüglich $\leq_y$ wird zurückgegeben durch:

  function succY(p: point): point;

- Der Vorgänger eines Punktes p in der y-table bezüglich $\leq_y$ wird zurückgegeben durch:

  function predY(p: point): point;

- *checkDelta* überprüft, ob zwei Punkte p_1 und p_2 einen Abstand $< \delta$ haben. Ist dies der Fall, so werden $closest_1$ und $closest_2$ sowie δ aktualisiert:

```
procedure checkDelta(p1, p2: point);
  begin
    newDelta := distance(p1, p2);
    if  newDelta < delta  then  begin
      delta := newDelta;
      closest1 := p1;  closest2 := p2;
    end;
  end;
```

- Der Plane-sweep Algorithmus wird folgendermaßen initialisiert:
  initX; initY;

```
closest1 := xQueue[1];  closest2 := xQueue[2];
delta := distance(closest1, closest2);
insertY(closest1);  insertY(closest2);
c := 3;  t := 1;
```

- Abarbeitung der Ereignisse durch die folgende Schleife:

```
while  c ≤ n do
  begin  transition;  c := c + 1;  { nächstes Ereignis }  end;
```

- *transition* umfaßt die gesamte Bearbeitung für einen neuen Punkt:

```
procedure transition;
  begin
```

1. *Entferne alle Punkte q mit* $q_x \leq p_x - \delta$ *aus der y-table:*

```
current := xQueue[c];
while  current.x – xQueue[t].x ≥ delta  do  begin
  deleteY(xQueue[t]);  t := t + 1
end;
```

2. *Füge p in die y-table ein:*

```
insertY(current);
```

3a. Überprüfe die Nachfolger des neuen Punktes in der y-table:

```
check := current;
repeat
  check := succY(check);
  checkDelta(current, check)
until  check.y – current.y > delta;
```

3b. Überprüfe die Vorgänger des neuen Punktes in der y-table:

```
check := current;
repeat
  check := predY(check);
  checkDelta(current, check)
until  current.y – check.y > delta;
end;  { transition }
```

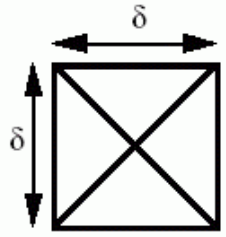
Hilfssatz für die Analyse des Laufzeitverhaltens:

Ein Rechteck mit den Kantenlängen $s \cdot \delta$ und $t \cdot \delta$ kann höchstens $(s+1) \cdot (t+1) + s \cdot t$ Punkte enthalten, deren Abstand bezüglich einer beliebigen d_k-Metrik mindestens δ beträgt.

Beweis:

Betrachte zunächst die Manhattan-Metrik d_1:

Partitioniere Rechteck in $s \cdot t$ Quadrate der Kantenlänge δ, zerlege jedes Quadrat in 4 paarweise disjunkte Dreiecke:



Dreiecke sind Halbkreise mit Durchmesser δ bzgl d_1.

Kein Halbkreis kann mehr als einen Punkt in seinem Inneren enthalten.

Quadrat höchstens 5 Punkte enthalten, wenn diese in den Eckpunkten der Dreiecke, d.h. auf dem Rand der Halbkreise, positioniert sind.

In dem Rechteck gibt es insgesamt $(s+1) \cdot (t+1) + s \cdot t$ verschiedene Dreieckseckpunkte $\Rightarrow$ Behauptung des Lemmas für d_1.

$d_1(p,q) \geq d_k(p,q) \geq d_\infty(p,q) \Rightarrow$ Rechteck kann für eine beliebige d_k-Metrik nicht mehr als $(s+1) \cdot (t+1) + s \cdot t$ Punkte mit paarweiser Distanz $\geq \delta$ enthalten

Satz:
Das Sweep-Verfahren löst das *closest pair*-Problem in der Ebene in optimaler Zeit $\Theta(n \log n)$ und Speicherplatz $\Theta(n)$.

Beweis:
Dass *mindestens n* log *n* erforderlich ist, ergibt sich durch Transformation aufs ε-Closeness-Problem (vgl. Kap. 2).
Noch zu zeigen: Das Sweep-Verfahren braucht $O(n \log n)$ Zeit und $O(n)$ Speicherplatz.

- Implementierung der *y-table* durch balancierten Baum, z.B. AVL-Baum oder 2-3-Baum, so daß die Operationen *insertY*, *deleteY*, *succY*, und *predY* in O(log n) Zeit durchgeführt werden können.
- Benötigter Speicherplatz für y-table: O(n)
- AVL-Baum: zusätzliche Zeiger, die auf den Vorgänger und Nachfolger des in einem Knoten gespeicherten Elementes e verweisen $\Rightarrow$ succY und predY in O(1) Zeit, wenn die Position des e speichernden Knotens bekannt ist. Diese zusätzliche Verzeigerung kann während Einfüge- und Löschoperationen aufrecht erhalten werden, ohne daß dadurch die Zeitkomplexität für 'insertY' und 'deleteY' verschlechtert wird.
- *initX* generiert die sortierte *x-queue* in O(n·log n) Zeit und benötigt O(n) Speicherplatz.
- *deleteY* wird höchstens einmal für jeden Punkt aufgerufen, alle Aufrufe von *deleteY* kosten daher O(n·log n) Zeit.
- Jeder Punkt wird einmal in die y-table eingefügt, somit kosten auch alle Aufrufe von *insertY* O(n·log n) Zeit.

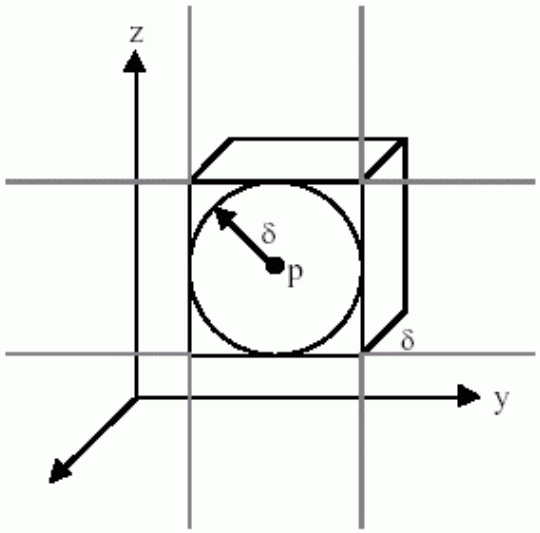
- Zeitaufwand für Schritt 3:

Anzahl der Aufrufe von *succY* in der Schleife in Schritt 3a ist um 1 größer als Anzahl Punkte in der oberen Hälfte des achsenparallelen Containers.
Analog: Anz. d. Aufrufe von *predY* in 3b um 1 größer als Anz. Punkte in der unteren Hälfte desselben.
Alle Punkte links von der *scan line* haben Abstand $\geq \delta$.
Mit dem Hilfssatz folgt: dieser Container ist "dünn besiedelt".

Für jede d_k-Metrik enthält der Container nicht mehr als 8 Punkte
$\Rightarrow$ *succY* und *predY* werden höchstens 10 mal aufgerufen
$\Rightarrow$ Schritt 3 kostet $O(\log n)$ Zeit
$\Rightarrow$ für alle Punkte summiert: $O(n \log n)$ Zeit.

Sweep in höheren Dimensionen:

- Läßt sich das Plane-sweep Prinzip auch auf höherdimensionale Räume übertragen?
- Idee im 3d:
  - Punkte sortiert bezüglich x-Koordinate in der x-queue.
  - Überstreiche Raum mit einer y-z-Ebene.
  - Paar von Punkten mit minimalem Abstand δ unter den links von p liegenden Punkten sei bei Bearbeitung von p bekannt.
  - Betrachte alle Punkte, die in der zur Linken von p sich erstreckenden Halbkugel um p mit Radius δ liegen, um zu bestimmen, ob p mit einem bereits vorher angetroffenen Punkt ein neues Paar mit minimalem Abstand bildet.
  - Schließe die Halbkugel in ihren achsenparallelen Container mit Kantenlängen $2\cdot\delta$ in der y- und z-Dimension und δ in der x-Dimension ein. Die Anzahl Punkte, die sich innerhalb dieses Containers befinden können, ist wiederum durch eine kleine Konstante c_k beschränkt, die von der zugrundeliegenden Metrik d_k abhängt.
  - Implementiere Containerabfrage in zwei Schritten:
    1. Entferne alle Punkte q zur Linken von p, die bezüglich der x-Koordinaten einen Abstand größer als δ von p haben, d.h. $q_x \leq p_x - \delta$.
    2. Suche unter den Punkten in der y-z-table alle die Punkte, die innerhalb eines Quadrats mit Seitenlänge $2\cdot\delta$ um p liegen:



- Problem: 2-dimensionale Bereichsabfrage kann im allgemeinen nicht in O(log n) Zeit pro Punkt mit einer der bekannten Datenstrukturen beantwortet werden.
- Plane-sweep reduziert die Dimensionalität eines Problems, indem es eine Raumdimension durch eine *Zeitdimension* ersetzt.

Sweep-Verfahren zur Bestimmung der Schnittpunkte von Liniensegmenten

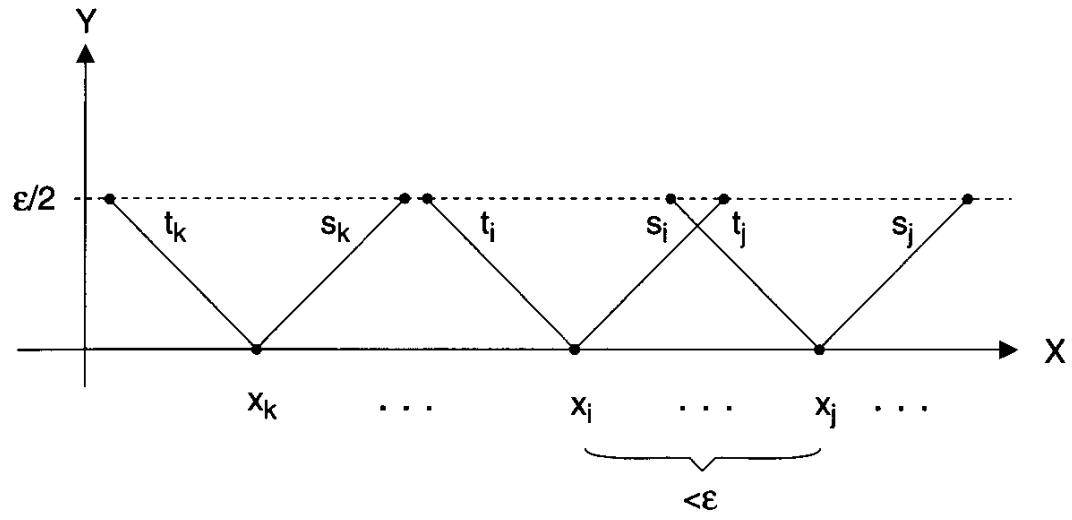
(Bentley & Ottmann 1979)

Gegeben: n Liniensegmente (Strecken) $s_i = l_i\, r_i$ (i = 1, ..., n) in der Ebene.
Existenzproblem: Gibt es unter den n Segmenten zwei, die einen [echten] Schnittpunkt haben?
Aufzählungsproblem: Alle [echten] Schnittpunkte sollen aufgezählt werden.
(*Echte* Schnittpunkte: solche, die im Inneren beider beteiligten Strecken liegen.)
Es werden zunächst nur echte Schnittpunkte betrachtet.

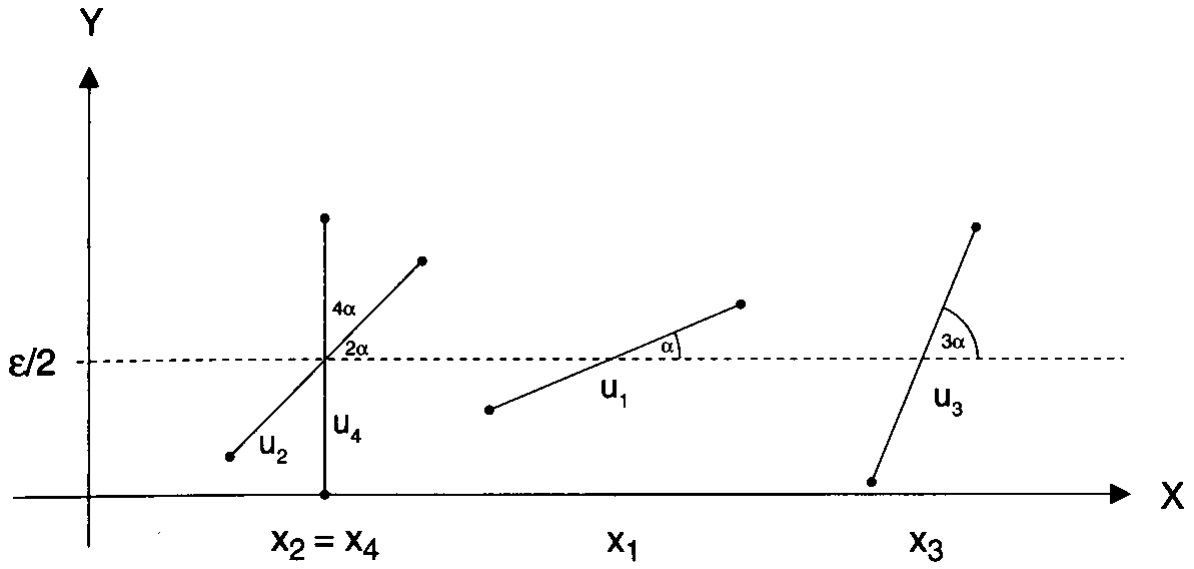
Untere Schranken für beide Probleme:

Satz:
Herauszufinden, ob zwischen n Strecken in der Ebene ein echter Schnittpunkt existiert, hat die Zeitkomplexität $\Omega(n \log n)$.
Alle k Schnittpunkte aufzuzählen, hat die Zeitkomplexität $\Omega(n \log n + k)$.

Beweis: durch Transformation des Problems "ε-Closeness" auf das Existenzproblem in linearer Zeit:
Seien n reelle Zahlen x_1, ..., x_n und ein $\varepsilon > 0$ gegeben.
Bilde die Strecken s_i : von $(x_i, 0)$ nach $(x_i + \varepsilon/2, \varepsilon/2)$
und t_i : von $(x_i, 0)$ nach $(x_i - \varepsilon/2, \varepsilon/2)$.



Diese Konstruktion ist in linearer Zeit möglich.
s_i und t_j haben genau dann einen echten Schnittpunkt, wenn gilt: $0 < | x_i - x_j | < \varepsilon$.
Um ε-Closeness zu lösen: teste, ob bei diesen Segmenten ein echter Schnittpunkt vorliegt. Falls ja: ε-Closeness wird positiv beantwortet, fertig.
Falls nein: die voneinander verschiedenen x_k haben alle mindestens den Abstand ε. Aber: Es kann immer noch gleiche Zahlen geben.
Deshalb zweiter Schnittpunkttest:
Zu jedem x_i konstruiere Strecke u_i der Länge ε mit Mittelpunkt $(x_i , \varepsilon/2)$. Jede dieser Strecken erhält einen anderen Winkel zur x-Achse, der nur vom Index i abhängt:



Da verschiedene Punkte schon mindestens den Abstand ε haben, können sich ihre zugehörigen Strecken nicht schneiden.

Zwei Strecken zu Punkten x_i, x_j mit verschiedenem Index $i \neq j$ haben genau dann einen echten Schnittpunkt, wenn $x_i = x_j$ gilt.

$\Rightarrow$ die Antwort auf den zweiten Schnittpunkttest entscheidet das ε-Closeness-Problem

$\Rightarrow$ Existenzproblem für Schnittpunkte hat Zeitkompl. $\Omega(n \log n)$.

Hat man Algorithmus für das Aufzählungsproblem $\Rightarrow$ Existenz-problem ist auch gelöst; außerdem müssen alle k Schnittpunkte aufgezählt werden $\Rightarrow$ unt. Schranke $\Omega(n \log n + k)$.

Naiver Algorithmus für Schnittpunkt-Test (alle Paare von Strecken testen): Zeitaufwand $O(n^2)$.
Da Anzahl der Schnittpunkte im ungünstigsten Fall $\in O(n^2)$, geht es nicht besser, wenn nur n als Parameter betrachtet wird.

Nachteil: dieser Aufwand würde beim naiven Alg. immer anfallen, auch wenn es nur wenige (oder keine) Schnittpunkte gibt.

Man wünscht Algorithmus, dessen Laufzeit nur dann groß wird, wenn auch die Anzahl k der Lösungen groß ist:
Output-Sensitivität

Sweep-Verfahren

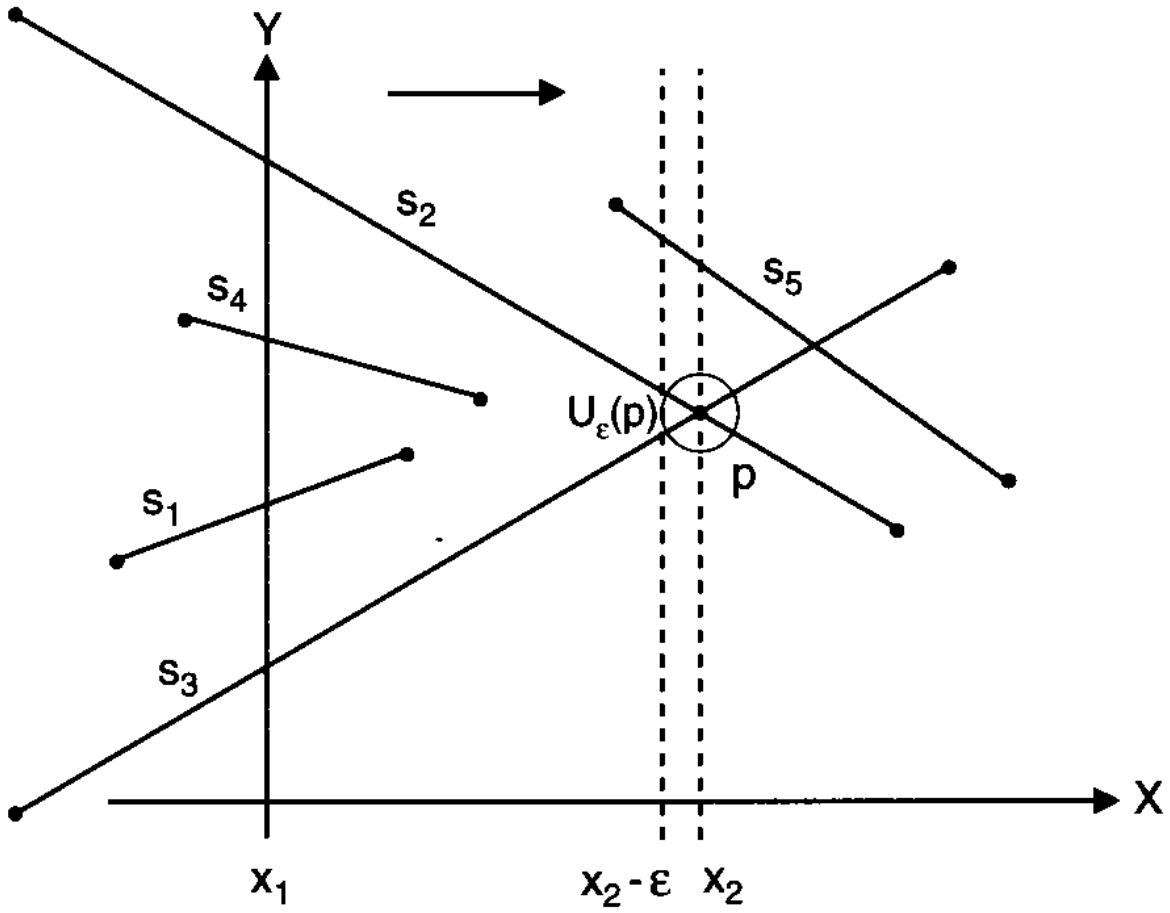
zunächst vereinfachende Annahmen:

- keine der Strecken vertikal
- Schnitt ist leer oder einzelner Punkt
- in 1 Punkt schneiden sich höchstens 2 Segmente
- alle End- und Schnittpunkte haben unterschiedliche x-Koordinaten.

Sweep line bewegt sich von links nach rechts

aktuelle x-Koordinate der *sweep line* = "Zeitpunkt" x_0

Zu jedem x_0: y-Koordinaten der Schnittpunkte der *sweep line* mit den Strecken def. totale Ordnung auf Menge der von der *sweep line* geschnittenen Strecken (Ordnung wird mitgeführt in der Sweep-Status-Struktur).



hier zur Zeit x_1 Ordnung $s_3 < s_1 < s_4 < s_2$.

Vor Beginn des Sweep werden alle Endpunkte von Strecken nach ihren *x*-Koordinaten sortiert.

Welche Ereignisse verändern die Ordnung der Strecken?

(1) Sweep line stößt auf linken Endpkt. einer Strecke
(2) Sweep line erreicht rechten Endpkt. einer Strecke
(3) Sweep line erreicht einen echten Schnittpunkt zweier Strecken.

Wann werden diese Ereignisse bearbeitet?
(1) und (2) klar (*x*-Koord. der Endpkte sind bekannt).
zu (3): Schnittpunkte sind vorher nicht bekannt
aber:
aus den Voraussetzungen u. aus Stetigkeitsüberlegungen folgt: wenn 2 Strecken einen echten Schnittpunkt haben, sind sie unmittelbar vorher direkte Nachbarn in der Ordnung längs der *sweep line*.

Also: teste 2 Strecken *sofort* auf möglichen Schnitt, wenn sie in der Ordnung längs der *sweep line* zu direkten Nachbarn werden!

Das heißt:
Bei Behandlung von (1):

- neue Strecke *s* an richtiger Stelle in aktuelle Ordnung längs der *sweep line* einfügen

sei danach *p* der direkte Vorgänger und *q* der direkte Nachfolger von *s* längs der *sweep line*:

- teste Schnitt: *s* mit *p* und *s* mit *q* (siehe Kapitel 2.6).

Bei Behandlung von (2):

- entferne Strecke *s*, deren rechter Endpunkt erreicht wurde, aus der SSS
- früherer direkter Vorgänger und Nachfolger sind nun selbst benachbart $\Rightarrow$ teste diese auf Schnitt

somit:
Sweep-Verfahren zur Lösung des Existenzproblems:

- sortiere die 2*n* Endpunkte der Strecken nach aufsteigenden x-Werten, speichere sie in Array
- Sweep-Vorgang: bearbeite auftretende Ereignisse (1) und (2) wie oben beschrieben
- sobald erster Schnitt-Test positives Ergebnis liefert $\Rightarrow$ fertig!

Ereignisse vom Typ (3) brauchen nicht behandelt zu werden.

notwendige Operationen in der Sweep-Status-Struktur:

FügeEin(*SSS*, *s*, *x*) (*s* Strecke, *x* Zeitpunkt)
Entferne(*SSS*, *s*, *x*)
Vorgänger(*SSS*, *s*, *x*)
Nachfolger(*SSS*, *s*, *x*)

verwende zur Implementierung der SSS (Liniensegmente entspr. Ordnung längs der *sweep line*) AVL-Baum mit verketteten Blättern; darin höchstens *n* Strecken gespeichert

$\Rightarrow$ jede der Operationen in Zeit $O(\log n)$ ausführbar

Bearbeitung von jedem Ereignis (davon höchstens 2*n*) erfordert max. 3 solche Operationen $\Rightarrow$

Satz:
Man kann in optimaler Zeit $O(n \log n)$ und mit Speicherplatz $O(n)$ herausfinden, ob von n Strecken in der Ebene mindestens zwei einen echten Schnittpunkt haben.

Lösung des Aufzählungsproblems:

- merke gefundene echte Schnittpunkte als zukünftige Ereignisse vom Typ (3) vor ($\Rightarrow$ dynamische Datenstruktur für die Ereignisse notwendig)
- Bei Ereignis v. Typ (3) müssen die beiden sich schneidenden Strecken in der SSS vertauscht werden:
  Vertausche(SSS, *s*1, *s*2, *x*)

danach hat jede der Strecken *s*1 und *s*2 neuen direkten Nachbarn, mit dem auf Schnitt zu testen ist.

Erforderliche Operationen für Ereignisstruktur (*ES*):
Initialisierung von *ES*
Test, ob Warteschlange *ES* leer
FügeEin(*ES*, Ereignis)
NächstesEreignis(*ES*)

mögliche Objektstruktur für Ereignisse:

```
type tEreignis = record
    Zeit: real;
    case Typ: (LinkerEndpunkt,
                RechterEndpunkt, Schnittpunkt) of
        LinkerEndpunkt, RechterEndpunkt:
            Seg: tSegment;
        Schnittpunkt:
            USeg, OSeg: tSegment;
```

Sweep-Algorithmus zum Aufzählen aller echten Schnittpunkte von n Strecken (aus Klein 1997):

```
(* Initialisierung *)
initialisiere die Strukturen SSS und ES;
sortiere die 2n Endpunkte nach aufsteigenden X-Koordinaten;
erzeuge daraus Ereignisse;
füge diese Ereignisse in die ES ein;

(* sweep und Ausgabe *)
while ES ≠ ∅ do
    Ereignis := NächstesEreignis(ES);
    with Ereignis do
        case Typ of
            LinkerEndpunkt:
                FügeEin(SSS, Seg, Zeit);
                VSeg := Vorg(SSS, Seg, Zeit);
                TesteSchnittErzeugeEreignis(VSeg, Seg);
                NSeg := Nachf(SSS, Seg, Zeit);
                TesteSchnittErzeugeEreignis(Seg, NSeg);
            RechterEndpunkt:
                VSeg := Vorg(SSS, Seg, Zeit);
                NSeg := Nachf(SSS, Seg, Zeit);
                Entferne(SSS, Seg, Zeit);
                TesteSchnittErzeugeEreignis(VSeg, NSeg);
            Schnittpunkt:
                Berichte (USeg, OSeg) als Paar mit Schnitt;
                Vertausche(SSS, USeg, OSeg, Zeit);
                VSeg := Vorg(SSS, OSeg, Zeit);
                TesteSchnittErzeugeEreignis(VSeg, OSeg);
                NSeg := Nachf(SSS, USeg, Zeit);
                TesteSchnittErzeugeEreignis(USeg, NSeg)
```

Prozedur *TesteSchnittErzeugeEreignis*(*S*, *T*):
testet, ob die Strecken *S* und *T* echten Schnittpkt. haben
falls ja: neues Ereignis vom Typ *Schnittpunkt* wird erzeugt, mit *USeg* = *S*, *OSeg* = *T* und *x*-Koord. des Schnittpunkts als Zeit;
dieses wird in *ES* eingefügt.

Implementierung der Ereignisstruktur *ES*:
Warteschlange, auf der jede Operation *FügeEin* und *NächstesEreignis* in Zeit $O(\log(\#\text{ gespeicherte Ereignisse}))$ ausführbar ist
(z.B. auch AVL-Baum)

Satz:
Mit dem Sweep-Verfahren kann man die k echten Schnittpunkte von n Strecken in der Ebene in der Zeit $O((n+k)\log n)$ und mit Speicherplatz $O(n+k)$ bestimmen.

Beweis: Anzahl aller Ereignisse = $2n+k$. Keines wird mehrfach in die ES aufgenommen (Versuch, schon vorhandenes Objekt erneut einzufügen, wird abgewiesen)
$\Rightarrow$ wegen $k \leq n^2$ sind nicht mehr als $O(n^2)$ Ereignisse gleichzeitig in der ES gespeichert
$\Rightarrow$ jeder Zugriff auf die ES ist in Zeit $O(\log n^2) = O(\log n)$ ausführbar.
Die *while*-Schleife wird $(2n+k)$ mal durchlaufen, bei jedem Durchlauf wird eine beschränkte Zahl (≤ 7) von Operationen auf der SSS oder der ES ausgeführt $\Rightarrow$ es folgt die Behauptung.

Man kann den Speicherplatzbedarf für die ES bei diesem Algorithmus schärfer durch $O(n(\log n)^2)$ abschätzen (hier ohne Beweis; Pach & Sharir 1991).

Verbesserung des Verfahrens, zur Reduktion des Speicherplatzbedarfs: Man merke sich nur die Schnittpunkte zwischen solchen Strecken, die aktuell in der SSS benachbart sind (davon kann es zu jedem Zeitpunkt höchstens $n-1$ geben).

$\Rightarrow$ Satz: Die k echten Schnittpunkte von n Strecken in der Ebene lassen sich in der Zeit $O((n+k)\log n)$ und mit Speicherplatz $O(n)$ bestimmen.

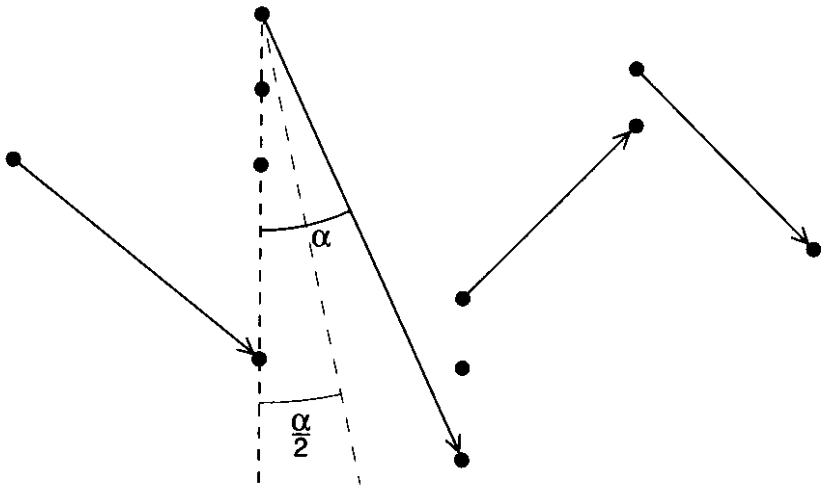
Beachte:
Diskrepanz zwischen (allg.) unterer Schranke $\Omega(n \log n + k)$ und oberer Schranke $O((n+k) \log n)$ (spezieller Algorithmus) für den Zeitbedarf des Aufzählungsproblems der Schnittpunkte.

Lösung der genauen Komplexität des Problems:

- Chazelle & Edelsbrunner 1992: Algorithmus mit optimaler Zeit $O(n \log n + k)$, aber mit Speicherbedarf $O(n+k)$
- Balaban 1995: Speicherbedarf auf $O(n)$ gedrückt.

Wie beseitigen wir die vereinfachenden Annahmen?

- Endpunkte mit gleicher x-Koordinate:
  - Sortiere diese nach ihren y-Koordinaten (d.h. lexikografisch)
  - verbinde jeweils den obersten Punkt einer Gruppe mit gleicher x-Koordinate mit dem untersten Punkt der rechten Nachbargruppe
  - bestimme für jede dieser Verbindungsstrecken den Winkel mit der negativen y-Achse



  - bestimme das Minimum α dieser Winkel
  - rotiere das alte Koordinatensystem im Uhrzeigersinn um $\alpha/2$ ($\rightarrow$ neues Koordinatensystem x', y')

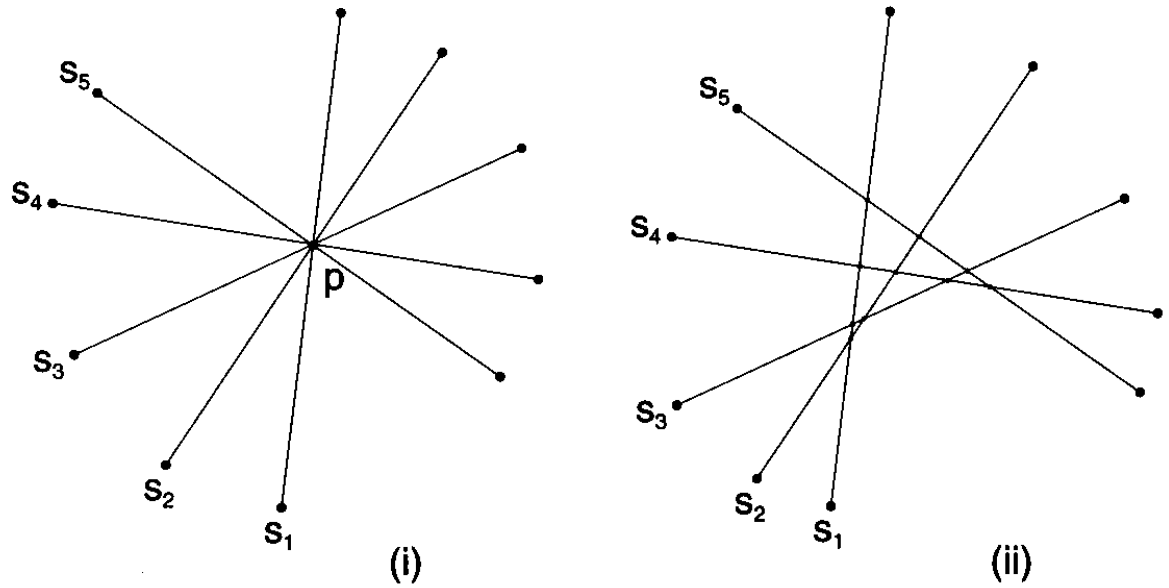
$\Rightarrow$ im neuen Koordinatensystem haben alle Endpunkte verschiedene x'-Koordinaten.

es kann trotzdem noch zum Auftreten von Ereignissen mit gleichen *x*-Koordinaten kommen (durch Schnittpunkte)

$\rightarrow$ modifiziere Algorithmus so, dass beim Auftreten von Ereignissen mit gleicher *x*-Koord. in der ES erst die linken Endpunkte, dann die Schnittpunkte, dann die rechten Endpunkte bearbeitet werden
- innerhalb dieser Gruppen werden die Ereignisse nach aufsteigenden *y*-Koordinaten sortiert
- wenn mehrere linke Endpunkte zusammenfallen: verwende die Ordnung der anhängenden Strecken
- analog für mehrfache rechte Endpunkte

- vielfache Schnittpunkte:

bearbeite diese so, als würde es sich um mehrere verschiedene Schnittpunkte handeln

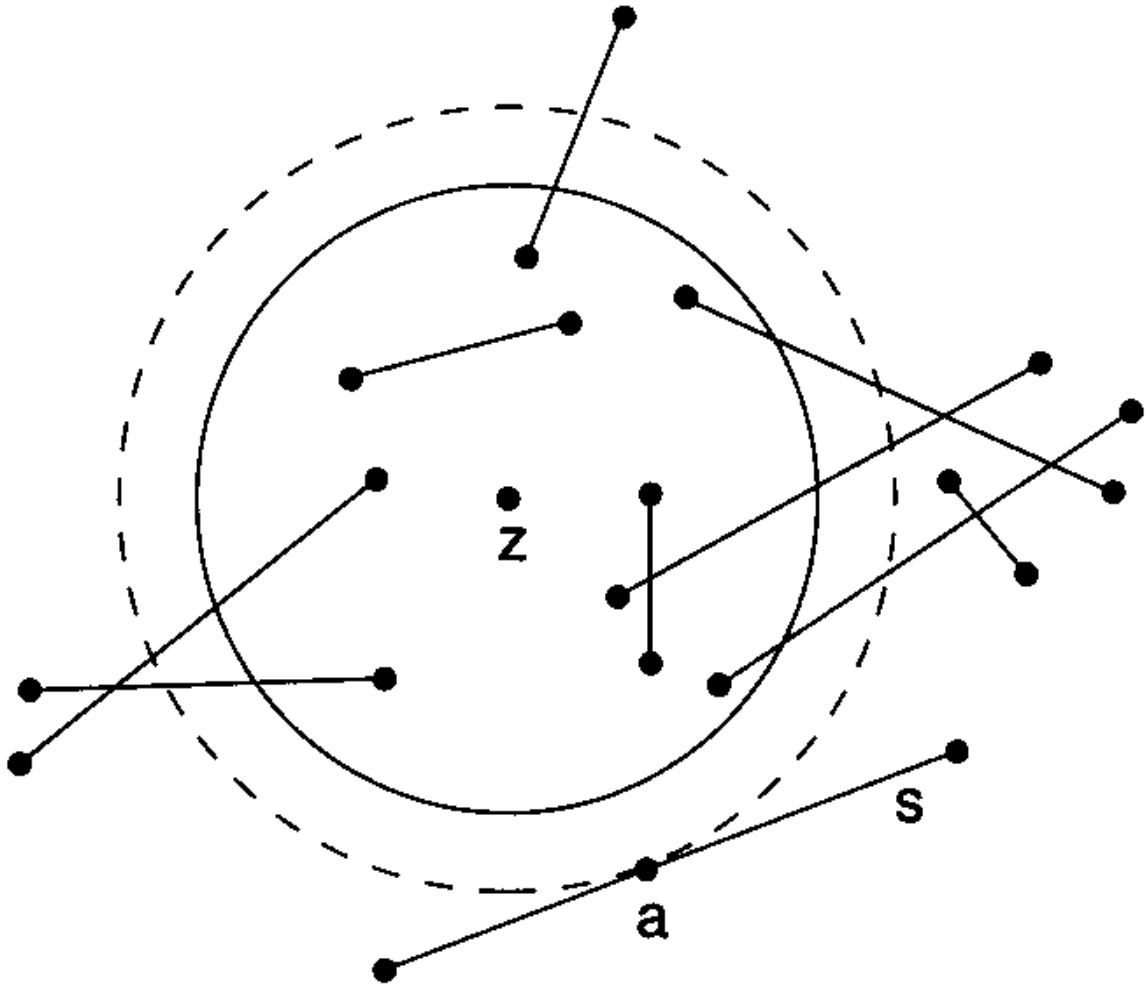


d.h. bearbeite (i) wie (ii) (ohne das Koordinatensystem oder die Strecken zu ändern!)

Variante des Sweep-Verfahrens:

sweep circle

- betrachte statt einer Geraden einen expandierenden Kreis
- Nachteil: eine Strecke kann "aktiv" werden, bevor der Kreis eine ihrer Endpunkte erreicht hat (z.B. für Strecke *s* in der Abb.): für jede Strecke ist zuerst der erste Auftreffpunkt *a* des *sweep circle* zu berechnen – wenn dieser kein Endpunkt ist, gibt er zu einem Ereignis Anlass.



Vorteil des *sweep circle* in der Praxis, wenn die Zahl der Segmente sehr groß ist und die Schnittpunkte nur in einem kleinen Bildausschnitt ermittelt werden sollen.