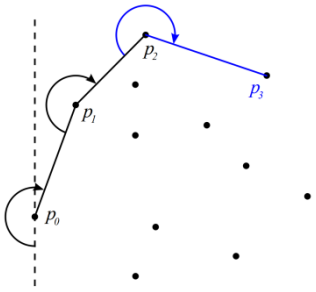
Geometrische Algorithmen

1 Konvexe Hülle

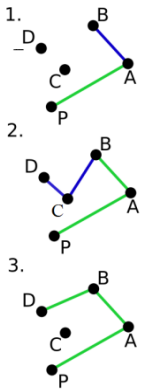
1.1 Jarvis March

Man beginnt bei einem Punkt p_0, von welchem man weiss, dass dieser in der konvexen Hülle liegen wird (z.B. der Punkt mit der kleinsten x-Koordinaten). Dann wird iterativ vorgegangen, und ein Punkt p_{i+1} ausgewählt, sodass alle Punkte links von der Geraden p_i-p_{i+1} liegen. Das geht in linearer Zeit, indem man die Winkel aller Geraden von p_0 durch einen andere Punkt betrachtet. Besteht die konvexe Hülle also aus h Punkten, so ergibt dies eine Laufzeit von O(n h), was in der Praxis ziemlich schlecht ist.



1.2 Graham scan

Wiederum wird mit einem Punkt p_0, welcher sicher in der konvexen Hülle enthalten ist, begonnen. Nun berechnet man den Winkel zwischen der Gerade p_0-p_i und der Gerade x = 0 für alle i aus und sortiert diese aufsteigend. Dies kostet O(n log n) Zeit.



Dann geht man durch die sortierte Folge der Punkte, beginnend bei p_1. Dabei wird für jeden Punkt entschieden, ob man einen Links- oder Rechtsknick durchgeführt hat. Bei einem Linksknick ist alles ok, wurde jedoch ein Rechtsknick ausgeführt beim Weg von p_{i-2} über p_{i-1} zu p_i, so wurde p_{i-1} fälschlicherweise zur konvexen Hülle hinzugenommen. Folglich wird dieser Punkt gestrichen und man betrachtet p_i erneut (jetzt ist es wieder möglich, dass ein Linksknick gemacht wurde).

Mit diesem Verfahren kann natürlich auch die konvexe Hülle eines Polygons gefunden werden, sogar in linearer Zeit, da dort kein Sortieren notwendig ist, und man direkt dem Polygon folgen kann.

2 Schnitt von Liniensegmenten

2.1 Orthogonale Lage

Um n orthogonale Liniensegmente auf Schnitte zu überprüfen, kann man das Scanline-Prinzip verwenden. Dazu werden alle Segmente aufsteigend bezüglich ihrer x-Koordinate sortiert, um die Haltepunkte der Scanline zu bestimmen. Um den aktuellen Status der Scanline zu speichern, wird ein binärer Suchbaum (idealerweise ein Blattsuchbaum). Dann wird links begonnen, und die Sweepline bewegt sich fortlaufend nach rechts, wobei an jedem Haltepunkt einer von drei Fällen eintrifft:

- Der Haltepunkt ist linkes Ende eines horizontalen Segmentes: Das Segment wird in die Scanline Datenstruktur aufgenommen, und zwar wird die y-Koordinate des Segmentes in den Baum eingetragen.
- Der Haltepunkt ist rechtes Ende eines horizontales Segmentes: Das Segment wird aus der Scanline-Datenstruktur entfernt.
- Der Haltepunkt ist ein vertikales Segment: Dieses Segment wird nicht in die Datenstruktur aufgenommen, dafür werden aber alle aktiven Segmente (also alle, die sich in der Scanline-Datenstruktur befinden) auf einen Schnitt mit diesem Segment überprüft. Dazu kann ein range-query mit der oberen und unteren Y-Koordinate als Grenzen verwendet werden.

Will man nun nur wissen, ob ein Schnitt vorhanden ist, so geht das mit einem beliebigen balancierten Suchbaum, denn sobald ein Schnitt gefunden wurde, kann abgebrochen werden.

Sollen jedoch alle Schnitte berichtet werden, benötigt man einen Blattsuchbaum, mit verketteter Blattliste. Dann kann man bei der range-query nach dem einen Ende suchen, und dann einfach so lange der verketteten Liste folgen, bis das andere Ende der query überschritten wurde.

Ist man nur an der Anzahl der Schnitte interessiert, so ist eine weitere Modifikation der Scanline-Datenstruktur nötig. An jedem Knoten werden nun zusätzlich die Anzahl Elemente in diesem Teilbaum gespeichert. Das lässt nun zu, dass für eine range-query in logarithmischer Zeit die Anzahl antworten ermittelt werden können. Dazu sucht man nach beiden Enden, wobei sich diese beiden Pfade an einem eindeutigen Knoten v teilen. Von dort aus folgt man nun zuerst dem linken Pfad, und wenn immer dieser Pfad nach rechts verzweigt, hat der Knoten an der Verzweigung einen linken Sohn, dessen Kinder alle im Bereich liegen. Die Anzahl Elemente, die dort steht, wird also mitgezählt. Dann wird auf dem rechten Pfad symmetrisch dasselbe ausgeführt. Für die Endpunkte selbst ist dann eine Fallunterscheidung nötig, insgesamt geht alles aber in logarithmischer Zeit.

Zusammenfassend gelten also folgende Laufzeiten:

```
detect: O( n log(n) )
report: O( n log(n) + k )        bei k Schnitten, mit k = O( n^2 )
count:  O( n log(n) )
```

Für orthogonale Segmente waren also primär eine Datenstruktur nötig, nämlich ein binärer Suchbaum (balanciert, möglicherweise Blattsuchbaum). Dazu wurde noch die sortierte Liste der Haltepunkte benötigt, diese konnte jedoch mit einem einfachen Array (oder mit einer sonstigen beliebigen linearen Datenstruktur) realisiert werden.

2.2 Beliebige Lage (Verfahren von Bentley und Ottman)

Befinden sich die Segmente in beliebiger Lage, so scheint die Situation schwieriger. Dennoch lässt sich das Sweepline-Prinzip anwenden: Wenn zu jedem Zeitpunkt alle „benachbarten Liniensegment“ auf der Scanline betrachtet werden, wird mit Sicherheit kein Schnitt verpasst. *Benachbart* in diesem Zusammenhang heisst, dass der Schnitt aller Segmente mit der Scanline betrachtet wird, und die Scanline-Datenstruktur wieder eine Menge von Punkten darstellt. Die Haltepunkte sind nun wie zuvor die sortierten End- und Anfangspunkte der Segmente, zusätzlich werden jedoch weitere Punkte, nämlich die Schnitte selbst hinzukommen. Bei jedem Haltepunkt gibt es nun eine von den folgenden Situationen:

- Linkes Ende eines Segmentes: Das Segment wird in die Scanline-Datenstruktur eingefügt, und es werden zwei Schnitttests durchgeführt (mit beiden Nachbarn). Gibt es einen Schnitt, wird dieser in die Haltepunkt-Struktur eingefügt.
- Rechtes Ende eines Segmentes: Das Segment wird aus der Scanline-Datenstruktur entfernt und die beiden Nachbarn, die nun selbst benachbart sind, werden auf Schnitt geprüft. Gibt es einen, wird dieser wiederum in die Haltepunkt-Struktur eingefügt.
- Schnittpunkt zwischen A und B: A und B müssen vertauscht werden in der Reihenfolge in der Scanline-Datenstruktur, und zusätzlich werden zwei Schnitttest, mit den neuen Nachbarn von A und B durchgeführt. Gegebenenfalls werden diese Schnitte in die Haltepunkte-Struktur eingefügt. Zusätzlich wird nun der Schnitt berichtet. Dies geschieht erst hier, da sonst gewisse Schnitte mehrmals berichtet werden könnten.

Dieser Algorithmus benötigt also zwei Datenstrukturen:

1. **Scanline-Datenstruktur**: Hier werden die gerade aktiven Liniensegmente gespeichert, wobei diese eingefügt, gelöscht und mittels *get-neighbor* abgefragt werden können müssen. Es bietet sich also an, einen balancierten Blattsuchbaum (dessen Blätter verketten sind) zu verwenden.
2. **Haltepunkt-Datenstruktur**: In dieser Struktur werden alle Haltepunkte gespeichert, also alle Anfangs- und Endpunkte der Segmente (zu Beginn bekannt) und alle Schnittpunkte (werden laufend berechnet). Da ein Schnittpunkt mehrere Male eingefügt werden kann, genügt eine einfache Event Queue wie etwa ein binärer Heap nicht; stattdessen muss ebenfalls ein balancierter Baum verwendet werden.

Um die Laufzeit zu analysieren, macht man folgende Beobachtungen: Der Algorithmus passiert im allgemeinen 2n+k Haltepunkte. An jedem Haltepunkt wird eine konstante Zahl von Operationen ausgeführt, wobei jeweils die Grösse der jeweiligen Binärbäume über die Zeitkomplexität entscheiden. Da sich höchstens $O(n+k) = O(n^2)$ Elemente in den Bäumen befinden, sind alle Operationen in logarithmischer Zeit durchführbar. Damit ergeben sich folgende Gesamtlaufzeiten:

```
detect: O( n log(n) )            (beim 1. Schnittpunkt abbrechen)
report: O( (n + k) log(n) )      bei k Schnitten, mit k = O( n^2 )
count:  -                        (nicht behandelt)
```

3 Rechteckschnittproblem

Um eine Menge von n iso-orientierter Rechtecke in der Ebene auf Schnitte zu prüfen, kann folgende Überlegung gemacht werden: Mithilfe eines Scanline-Verfahrens kann das statische Problem in der Ebene in eine Abfolge von dynamischen Intervallschnitt-Problemen überführen. Wir brauchen also eine Datenstruktur, in welche wir Intervalle einfügen und entfernen können. Zusätzlich soll effizient bestimmt werden können, welche Intervalle in der Struktur für ein gegebenes Intervall gemeinsame Punkte besitzen.

Um das Problem weiter zu vereinfachen, kann folgende Überlegung gemacht werden:

Sei $[a, b]$ ein gegebenes Intervall, welches wir auf Schnitt prüfen wollen. Dann sind $[c, d]$ alle Intervalle, welche wir berichten müssen:

$$\{[c, d] \mid [a, b] \cap [c, d] \neq \emptyset\}$$

Dieser Ausdruck lässt sich nun folgendermassen umformen:

$$\{[c, d] \mid a \text{ spiesst } [c, d] \text{ auf}\} \cup \{[c, d] \mid c \text{ liegt im Bereich } [a, b]\}$$

Folglich genügt es, eine Datenstruktur zu finden, welche die für die gespeicherte Menge an Intervallen und ein Intervall $[a, b]$ folgende zwei Fragen beantworten kann:

1. Berichte alle Intervalle $[c, d]$, die vom linken Randpunkt *a* aufgespiesst werden.
2. Berichte alle Intervalle $[c, d]$, deren linker Randpunkt *c* im Bereich $[a, b]$ liegt.

Die zweite Frage lässt sich natürlich leicht mit einem range-tree (Bereichs-Suchbaum) beantworten, die erste Frage hingegen ist neu.

3.1 Algorithmus

```
// liefert zu einer Menge von N iso-orientierten Rechtecken in der
// Ebene die Menge aller k Paare von sich schneidenden Rechtecken
Q: Folge der 2N ob./unteren Rechteckseiten in abnem. y-Reihenfolge
L: 0 // Menge der Schnitte der aktiven Rechtecke mit der Scan-line

while Q ist nicht leer do
  begin
    q : nächster Haltepunkt von Q;
    if q ist oberer Rand eines Rechtecks R, q = [xl(R),xr(R)] then
     begin
     bestimme alle Rechtecke R' derart, dass das Intervall
     [xl(R'),xr(R')] in L ist und sich R und R' schneiden.
     gebe (R,R') aus;

     füge [xl(R),xr(R)] in L ein
     end
    else // q ist unterer Rand eines Rechtecks R
      entferne [xl(R),xr(R)] aus L
end
```

3.2 Datenstruktur für L

Um die aktiven Rechtecke zu verwalten, benötigt man eine Datenstruktur, die Intervalle speichern und entfernen kann, sowie folgende Fragen beantworten:

1. Berichte alle Intervalle $[c, d]$, die vom linken Randpunkt a aufgespiesst werden.
2. Berichte alle Intervalle $[c, d]$, deren linker Randpunkt c im Bereich $[a, b]$ liegt.

Wie erwähnt lässt sich die zweite Frage einfach mit einem range-tree beantworten. Für die erste Frage lässt sich ein Intervall- oder Segmentbaum einsetzen.

Insgesamt lässt sich das Problem also in `O(n logn + k)` Zeit lösen. Je nachdem für welchen Baum man sich entscheidet, gibt dies einen unterschiedlichen Platzverbrauch: Die komplette Lösung mittels Intervalltree braucht `O(n)` Speicherplatz, während man mit einem Segmentbaum bei `O(n logn)` Speicher liegt.