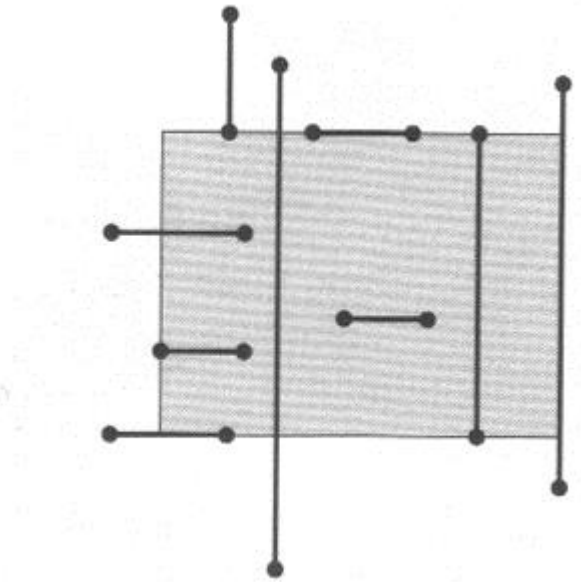
5.1. Intervallbäume (Intervall Trees)

Oft steht man vor der Aufgabe aus einer großen Menge von Liniensegmenten diejenigen auszuwählen, die zumindest teilweise innerhalb eines Rechteckes liegen. Ein typisches Beispiel ist die Anzeige eines Ausschnittes einer Straßenkarte oder eines integrierten Schaltkreises.

Bei integrierten Schaltkreisen treten vor allem horizontale und vertikale Linien auf. Wir wollen uns in diesem Abschnitt daher auf diesen (deutlich einfacheren) Fall beschränken. Der allgemeine Fall wird in 5.3 behandelt.



5.1. Intervallbäume (Intervall Trees)

Sei S eine Menge von n achsenparallelen Liniensegmenten in der Ebene. Sei $W = [x, x'] \times [y, y']$ das Fenster für unsere Suche, d. h., wir suchen alle Elemente in S, die mit W einen nicht leeren Schnitt haben.

Dazu gibt es vier Fälle:

1) Ein Segment liegt vollständig in W.
2) Ein Segment schneidet den Rand einfach.
3) Ein Segment schneidet den Rand zweifach.
4) Ein Segment überlappt teilweise den Rand von W.

5.1. Intervallbäume (Intervall Trees)

In den Fällen 1), 2) und 4) liegt ein Endpunkt in W. Dies können wir mit dem Range Tree aus §3.2 bezogen auf die $m = 2n$ Endpunkte mit $O(m \log m)$ Speicher und $O\big((\log m)^2 + k\big)$ Suchzeit bewältigen, wobei k die Anzahl der Endpunkte in W ist.

Mit Erweiterungen (fractional cascading) ist sogar $O((\log m) + k)$ möglich.

Doppelt ausgegebene Liniensegmente (beide Endpunkte in *W*, also 1) und in 4) möglich) lassen sich durch einen Marker vermeiden.

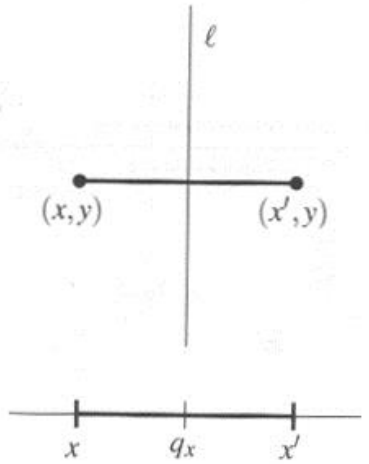
Es ergibt sich das folgende Lemma.

Lemma 5.1: Sei S eine Menge von n achsenparallelen Liniensegmenten in der Ebene. Die Segmente mit mindestens einem Endpunkt in einem achsenparallelen Rechteck (Fenster) können in $O((\log n) + k)$ Zeit mit Hilfe einer Datenstruktur mit $O(n \log n)$ Speicherbedarf ermittelt werden, wobei k die Anzahl der gesuchten Liniensegmente ist.

5.1. Intervallbäume (Intervall Trees)

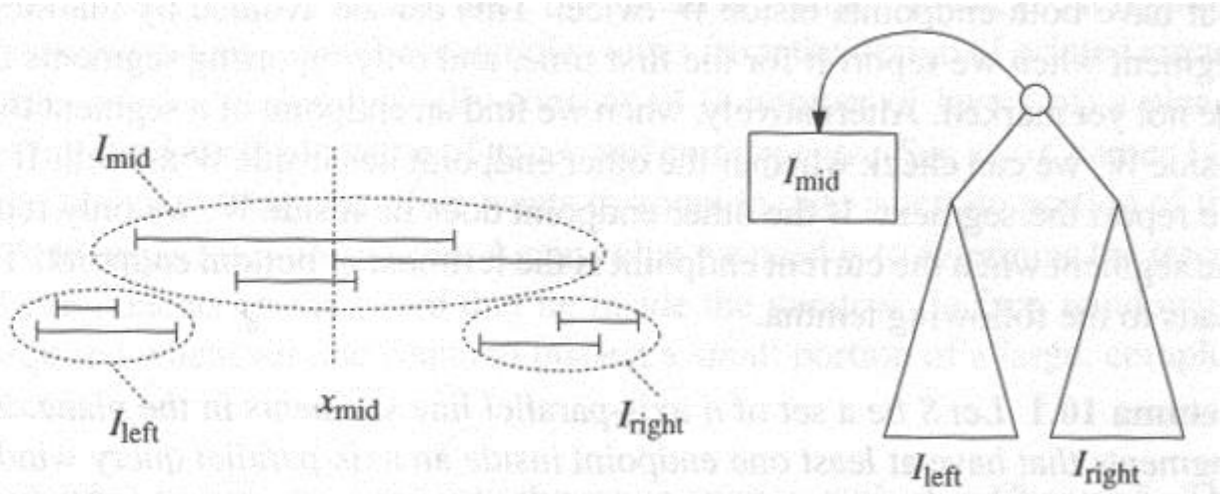
Es bleibt die Aufgabe alle Segmente zu finden, die W schneiden, aber keinen Endpunkt in W haben. Dabei gibt es zwei Fälle, da die horizontalen Segmente den linken und rechten Rand, die vertikalen Segmente dagegen den oberen und unteren Rand, schneiden. Wir beschränken uns auf die horizontalen Segmente und suchen diejenigen, die den linken Rand schneiden, wobei wir wieder die bereits zuvor gefundenen Segmente durch eine Markierung erkennen und weglassen können.

Wir haben nun ein eindimensionales Problem vor uns, da wir zu jedem Segment wissen wollen, ob ein bestimmter x -Wert q_x zwischen den Randwerten x und x' liegt.



5.1. Intervallbäume (Intervall Trees)

Sei also $I := \{[x_1, x_1'], [x_2, x_2'], \ldots, [x_n, x_n']\}$ eine Menge abschlossener Intervalle. Ferner sei x_{mid} der Median der $2n$ Intervallendpunkte. Wenn nun der Abfragewert q_x rechts von x_{mid} liegt, scheiden alle Intervalle aus, deren rechter Endpunkt links von x_{mid} liegt. Entsprechend können für q_x links von x_{mid} die Intervalle mit linkem Endpunkt rechts von x_{mid} aussortiert werden. Für die Intervalle, die x_{mid} enthalten, bauen wir eine weitere Struktur.



Classification of the segments with respect to x_{mid}

5.1. Intervallbäume (Intervall Trees)

Für die Intervalle, die x_{mid} enthalten, erzeugen wir zwei Listen. Die eine sortiert die Intervalle nach aufsteigendem linken Endpunkt. Die andere sortiert die Intervalle nach absteigendem rechten Endpunkt. Wenn nun unsere Suchanfrage q_x links von x_{mid} liegt, nutzen wir die erste Liste und geben solange Elemente aus, bis der linke Endpunkt rechts von q_x liegt. Analog nutzen wir die andere Liste, wenn q_x rechts von x_{mid} liegt.

5.1. Intervallbäume (Intervall Trees)

Als Datenstruktur erhalten wir den sogenannten Intervallbaum (intervall tree).

- Wenn $I = \emptyset$ ist, dann ist der Intervallbaum ein Blatt.
- Für $I \neq \emptyset$ sei x_{mid} der Median der Endpunkte aller Intervalle.

Ferner seien

$$\begin{aligned} I_{left} &:= \{[x_j, x_j'] \in I | x_j' < x_{mid}\} \\ I_{mid} &:= \{[x_j, x_j'] \in I | x_j < x_{mid} \leq x_j'\} \\ I_{right} &:= \{[x_j, x_j'] \in I | x_{mid} < x_j\} \end{aligned}$$

Der Baum besteht aus einer Wurzel v mit Wert x_{mid}, einer Liste $L_{left}(v)$ mit den Elementen von I_{mid} aufsteigend nach x sortiert, einer Liste $L_{right}(v)$ mit den Elementen von I_{mid} absteigend nach x' sortiert, einem linken Teilbaum von v und einem rechten Teilbaum von v. Der linke Teilbaum ist ein Intervallbaum für I_{left} und der rechte Teilbaum ist ein Intervallbaum für I_{right}.

5.1. Intervallbäume (Intervall Trees)

Algorithm CONSTRUCTINTERVALTREE(I)
Input. A set I of intervals on the real line.
Output. The root of an interval tree for I.
1. **if** $I = \emptyset$
2. **then return** an empty leaf
3. **else** Create a node v. Compute x_{mid}, the median of the set of interval endpoints, and store x_{mid} with v.
4. Compute I_{mid} and construct two sorted lists for I_{mid}: a list $\mathcal{L}_{\text{left}}(v)$ sorted on left endpoint and a list $\mathcal{L}_{\text{right}}(v)$ sorted on right endpoint. Store these two lists at v.
5. $lc(v) \leftarrow$ CONSTRUCTINTERVALTREE(I_{left})
6. $rc(v) \leftarrow$ CONSTRUCTINTERVALTREE(I_{right})
7. **return** v

5.1. Intervallbäume (Intervall Trees)

Lemma 5.2: Ein Intervallbaum für n Intervalle nutzt $O(n)$ Speicher und hat die Tiefe $O(\log n)$.

Lemma 5.3: Ein Intervallbaum für n Intervalle kann in $O(n \log n)$ Zeit erstellt werden.

Beweis: Die Intervalle können nach linkem und rechtem Endpunkt vorsortiert werden.

QED

5.1. Intervallbäume (Intervall Trees)

Algorithm QUERYINTERVALTREE(v, q_x)
Input. The root v of an interval tree and a query point q_x.
Output. All intervals that contain q_x.

1. **if** v is not a leaf
2. **then if** $q_x < x_{\text{mid}}(v)$
3. **then** Walk along the list $\mathcal{L}_{\text{left}}(v)$, starting at the interval with the leftmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
4. QUERYINTERVALTREE($lc(v), q_x$)
5. **else** Walk along the list $\mathcal{L}_{\text{right}}(v)$, starting at the interval with the rightmost endpoint, reporting all the intervals that contain q_x. Stop as soon as an interval does not contain q_x.
6. QUERYINTERVALTREE($rc(v), q_x$)

5.1. Intervallbäume (Intervall Trees)

Für die Suchzeit ergibt sich offensichtlich $O((\log n) + k)$, wobei k die Anzahl auszugebender Intervalle ist.

Theorem 5.4: Ein Intervallbaum für eine Menge I von n Intervallen benötigt $O(n)$ Speicher und kann in $O(n \log n)$ Zeit gebaut werden. Alle Intervalle, die einen Suchpunkt enthalten, können in $O((\log n) + k)$ gefunden werden, wobei k die Anzahl der gesuchten Intervalle ist

5.1. Intervallbäume (Intervall Trees)

Leider löst der Intervallbaum unsere Problem nicht ganz, da wir nur herausfinden, welche Liniensegmente von der Geraden durch den linken Rand unseres Fensters geschnitten werden. Um auch noch festzustellen, ob das Linienstück $q_x \times [q_y, q_y']$ schneidet, müssen wir statt der Listen eine andere Struktur verwenden. Wir müssen von I_{mid} wissen, welche linken Endpunkte in $(-\infty, q_x] \times [q_y, q_y']$ liegen. Dies ist durch einen Range Tree zu lösen! Dies ergibt $O(n_{mid} \log n_{mid})$ Speicher und $O((\log n_{mid}) + k_{mid})$ Suchzeit. Dadurch ergibt sich $O(n \log n)$ Speicher für die gesamte Struktur und insgesamt $O((\log n)^2 + k)$ Suchzeit.

5.1. Intervallbäume (Intervall Trees)

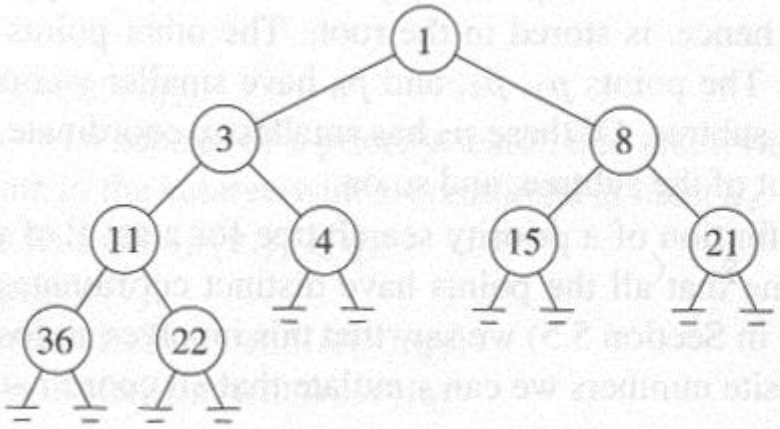
Theorem 5.5: Sei S eine Menge von n horizontalen Segmenten in der Ebene. Die ein vertikales Segment schneidenden Segmente können in $O\left((\log n)^2 + k\right)$ Zeit mit Hilfe von $O(n \log n)$ Speicher gefunden werden, wobei k die Anzahl gesuchter Segmente ist. Die Suchstruktur kann in $O(n \log n)$ aufgebaut werden.

Korollar 5.6: Sei S eine Menge von n achsenparallelen Liniensegmenten in der Ebene. Die Segmente, die ein achsenparalleles Suchfenster schneiden, können in $O\left((\log n)^2 + k\right)$ Zeit mittels einer Suchstruktur mit $O(n \log n)$ Speicher gefunden werden, wobei k die Anzahl der gesuchten Elemente ist. Die Suchstruktur kann in $O(n \log n)$ Zeit aufgebaut werden.

5.2. Suchbäume mit Priorität (Priority Search Trees)

Wir wollen nun eine zweite, speichereffizientere Datenstruktur für Fenstersuchen der Form $(-\infty, q_x] \times [q_y, q_y']$ auf Punktmengen kennenlernen. Dazu nutzen wir einen Haufen (heap).

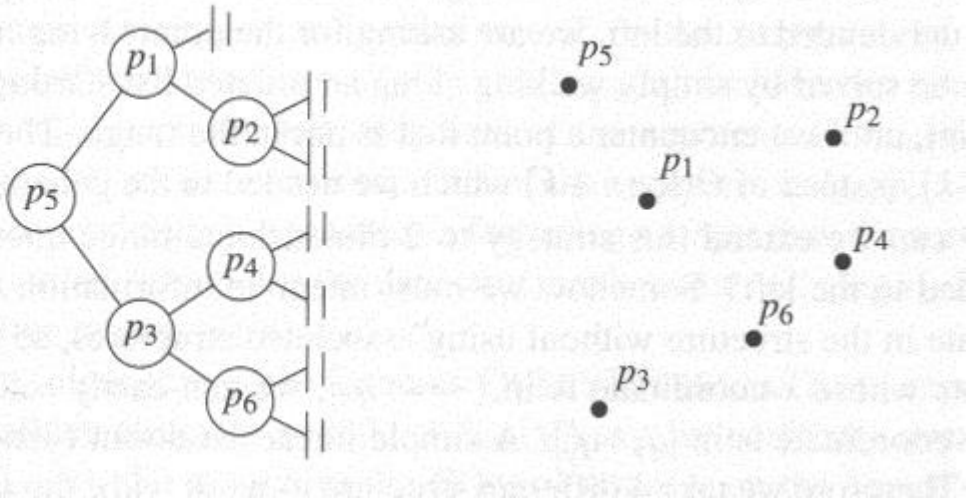
Ein Haufen ist ein binärer Baum, in dessen Wurzel das Objekt mit dem kleinsten bzw. größten Schlüssel abgelegt wird. Die übrigen Objekte werden in zwei etwa gleich große Mengen zerlegt, die rekursiv als Haufen organisiert werden.



A heap for the set $\{1, 3, 4, 8, 11, 15, 21, 22, 36\}$

5.2. Suchbäume mit Priorität (Priority Search Trees)

Die Ordnung von links nach rechts ist in dem Haufen nicht festgelegt. Dies nutzen wir für eine Sortierung in y-Richtung. Dies erlaubt dann eine schnelle Abfrage für unsere Fenstersuche.



A set of points and the corresponding priority search tree

5.2. Suchbäume mit Priorität (Priority Search Trees)

Formal können wir unseren Suchbaum mit Priorität für eine Menge $P = \{p_1, \dots, p_n\}$ von Punkten in der Ebene nun definieren:

- Für $n = 0$ ist der Suchbaum mit Priorität ein leeres Blatt.
- Für $n > 0$ sei p_{min} der Punkt in P mit kleinster x-Koordinate.

Ferner sei y_{mid} der Median der y-Koordinaten der übrigen Punkte.

Es sei

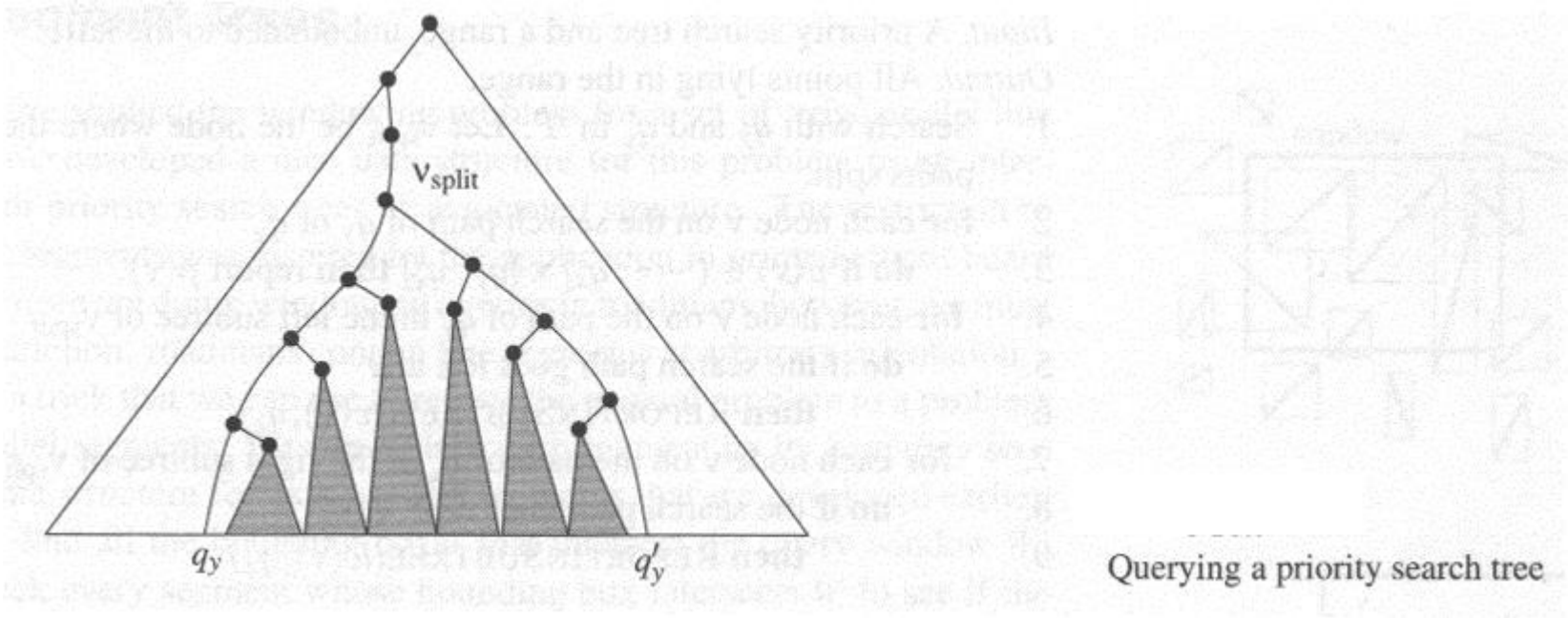
$$P_{below} := \{p \in P \backslash \{p_{min}\} | p_y \leq y_{mid}\}$$
$$P_{above} := \{p \in P \backslash \{p_{min}\} | p_y > y_{mid}\}$$

Der Suchbaum mit Priorität besteht aus einer Wurzel v mit dem Punkt $p(v) = p_{min}$ und dem Wert $y(v) := y_{mid}$ und zwei Teilbäumen.

- Der linke Teilbaum ist ein Suchbaum mit Priorität für P_{below}.
- Der rechte Teilbaum ist ein Suchbaum mit Priorität für P_{above}.

5.2. Suchbäume mit Priorität (Priority Search Trees)

Die Suche im Fenster $(-\infty, q_x] \times [q_y, q_y']$ erfolgt nun durch eine Suche nach q_y und q_y'. Zwischen den beiden Suchpfaden befinden sich Teilbäume mit dem richtigen y-Wertebereich, die wir nur anhand der x-Koordinate untersuchen.



Querying a priority search tree

5.2. Suchbäume mit Priorität (Priority Search Trees)

Zunächst betrachten wir die Suche nach der x-Koordinate in einem Teilbaum.

REPORTINSUBTREE(ν, q_x)
Input. The root ν of a subtree of a priority search tree and a value q_x.
Output. All points in the subtree with x-coordinate at most q_x.
1. **if** ν is not a leaf and $(p(\nu))_x \leqslant q_x$
2. **then** Report $p(\nu)$.
3. REPORTINSUBTREE($lc(\nu), q_x$)
4. REPORTINSUBTREE($rc(\nu), q_x$)

5.2. Suchbäume mit Priorität (Priority Search Trees)

Lemma 5.7: REPORTINSUBTREE(v, q_x) berichtet in $O(1 + k_v)$ Zeit alle Punkte im Teilbaum mit Wurzel v, deren x-Koordinate maximal q_x ist, wobei k_v die Anzahl der gesuchten Punkte ist.

Beweis: Ist p ein Punkt mit $p_x \leq q_x$, der am Knoten v in der Datenstruktur gespeichert ist, so bildet der Pfad von p zur Wurzel v eine Folge von Punkten mit absteigender x-Koordinate, so dass die Suche nicht abgebrochen wird und p berichtet wird. Da alle Punkte entlang des Pfades zu berichten sind, verursacht p nur $O(1)$ zusätzliche Zeit und es folgt $O(1 + k_v)$.

QED

5.2. Suchbäume mit Priorität (Priority Search Trees)

Neben diesen Teilbäumen sind noch die Punkte entlang der Suchpfade für q_y und q_y' bzgl. der x-Koordinate zu untersuchen.

Algorithm QUERYPRIOSEARCHTREE($\mathcal{T}, (-\infty : q_x] \times [q_y : q_y']$)
Input. A priority search tree and a range, unbounded to the left.
Output. All points lying in the range.

1. Search with q_y and q_y' in $\mathcal{T}$. Let v_{split} be the node where the two search paths split.
2. **for** each node v on the search path of q_y or q_y'
3. **do if** $p(v) \in (-\infty : q_x] \times [q_y : q_y']$ **then** report $p(v)$.
4. **for** each node v on the path of q_y in the left subtree of v_{split}
5. **do if** the search path goes left at v
6. **then** REPORTINSUBTREE($rc(v), q_x$)
7. **for** each node v on the path of q_y' in the right subtree of v_{split}
8. **do if** the search path goes right at v
9. **then** REPORTINSUBTREE($lc(v), q_x$)

5.2. Suchbäume mit Priorität (Priority Search Trees)

Lemma 5.8: QUERYPRIOSEARCHTREE berichtet die Punkte im Fenster $(-\infty, q_x] \times [q_y, q_y']$ in $O((\log n) + k)$ Zeit, wobei k die Anzahl gesuchter Punkte ist.

Beweis: Für die Punkte entlang des Suchpfades wird dies explizit getestet. Für die Punkte p in den Teilbäumen gilt wegen der Suchpfade $p_y < q_y'$ und $p_y \geq q_y$. Ferner folgt $p_x \leq q_x$ aus Lemma 5.7. Also liegen alle berichteten Punkte im Fenster.

Sei nun P ein Punkt im Fenster. Dann muss P auf oder rechts des Suchpfades für q_y und auf oder links des Suchpfades von q_y' sein. Damit wird er auch berichtet.

Die Suchzeit ist linear in den Elementen entlang des Suchpfades für q_y und q_y' zuzüglich der Zeit in REPORTSUBTREE. Die Tiefe des Baumes ist $O(\log n)$, also $O((\log n) + k)$.

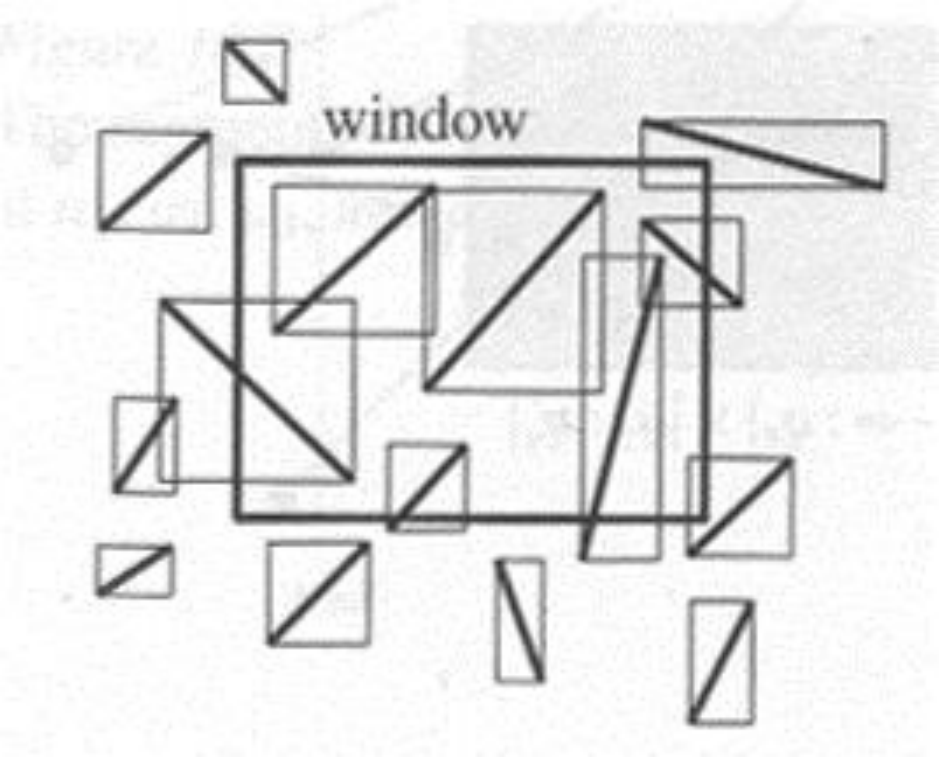
QED

5.2. Suchbäume mit Priorität (Priority Search Trees)

Theorem 5.9: Ein Suchbaum mit Priorität für eine Menge P von n Punkten in der Ebene nutzt $O(n)$ Speicher und kann in $O(n \log n)$ Zeit erstellt werden. Alle Punkte in einem Suchfenster der Form $(-\infty, q_x] \times [q_y, q_y']$ können in $O((\log n) + k)$ Zeit berichtet werden, wobei k die Anzahl der berichteten Punkte ist.

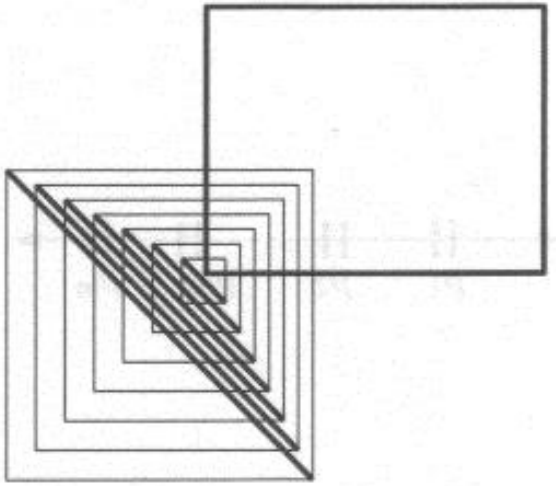
5.3. Segmentbäume (Segment Trees)

Wir wollen uns dem Problem beliebiger Liniensegmente zuwenden, wie es für Landkarten und Straßenkarten typisch ist. Wenn wir statt der Liniensegmente minimale, umfassende achsenparallele Rechtecke betrachten, können wir dies wieder auf achsenparallele Liniensegmente zurückführen.



5.3. Segmentbäume (Segment Trees)

Leider hat dieser Zugang Nachteile, so dass wir uns etwas anderes überlegen müssen, wenn wir gute obere Schranken für die Suchzeit auch bei ungünstiger Lage der Segmente garantieren wollen.



5.3. Segmentbäume (Segment Trees)

Die Segmente mit einem Eckpunkt im Suchfenster lassen sich wieder über einen Range Tree finden.

Daher konzentrieren wir uns auf die Segmente, die den Rand unseres Suchfensters schneiden. Da horizontale und vertikale Segmente in analoger Weise behandelt werden können, konzentrieren wir uns auf ein vertikales Segment $q_x \times [q_y, q_y']$ und suchen ein Verfahren zur effizienten Intersection Query für sich untereinander nicht schneidende Segmente.

(Wenn die Segmente sich schneiden dürfen, wird es komplizierter, siehe [de Berg, van Kreveld, Overmars, Schwarzkopf, Computational Geometry, Springer, Berlin, 2000, Kapitel 16].)

5.3. Segmentbäume (Segment Trees)

Zunächst suchen wir nach einer Datenstruktur, um Intervalle, die einen Punkt enthalten, schnell zu finden. Damit sollen aus allen Segmenten später diejenigen gefunden werden, welche die Gerade $\{x = q_x\}$ enthalten.

Sei also $I = \{[x_1, x_1'], \dots, [x_n, x_n']\}$ eine Menge von n Intervallen. Sei $p_1, \dots, p_m$ die sortierte Liste der voneinander verschiedenen Endpunkte. Dann zerlegen wir die reelle Achse in elementare Intervalle

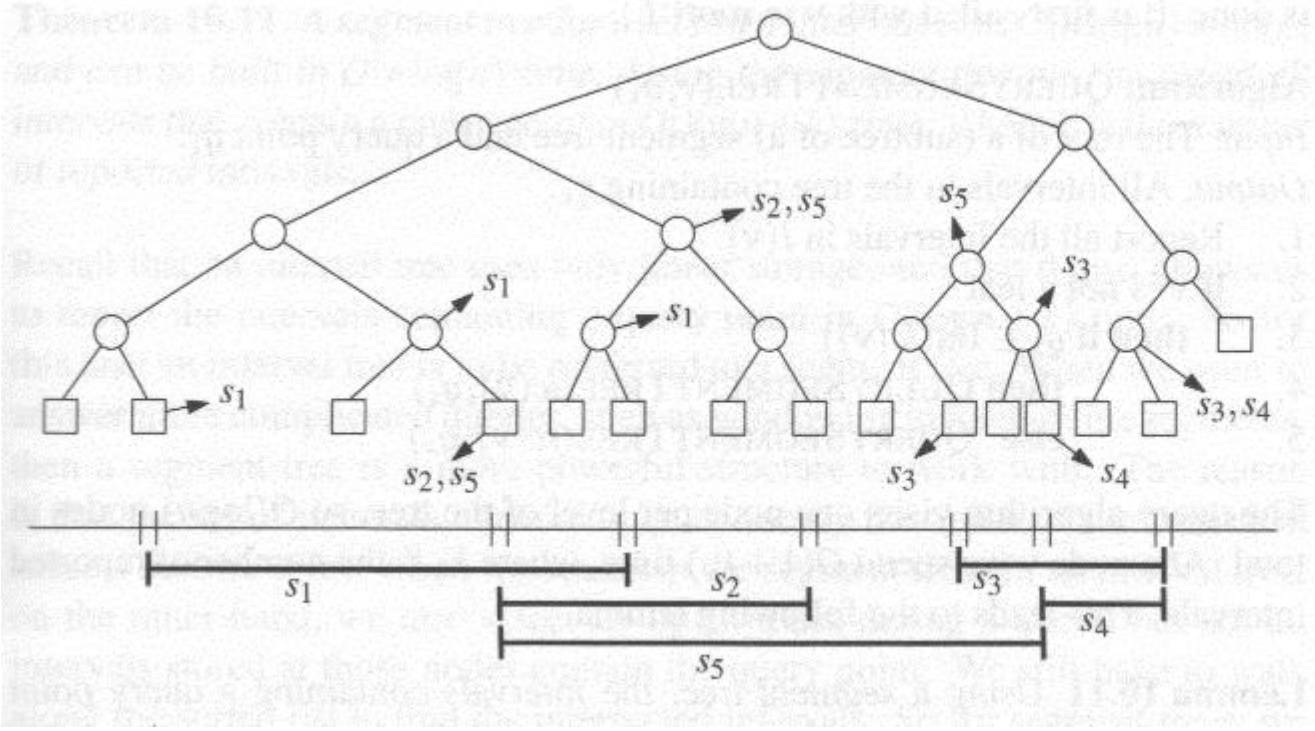
$$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \dots, [p_m, p_m], (p_m, \infty)$$

um so besser alle Intervalle zu finden, die einen bestimmten Punkt enthalten.

Wenn wir nun alle Intervalle, die ein elementares Intervall enthalten, in einem Blatt ablegen würden, erhielten wie eine $O\big((\log n) + k\big)$ Suchzeit, aber eventuell $O(n^2)$ Speicherbedarf!

5.3. Segmentbäume (Segment Trees)

Um dies besser zu organisieren, speichern wir die Intervalle möglichst weit oben in einem binären Suchbaum für die elementaren Intervalle.



A segment tree: the arrows from the nodes point to their canonical subsets.

5.3. Segmentbäume (Segment Trees)

Um den **Segmentbaum** zu definieren, legen wir fest:

Das **Skelett** eines Segmentbaumes ist ein balancierter binärer Baum T. Die **Blätter** von T korrespondieren zu den elementaren Intervallen zur Menge I und sind entsprechend von links nach rechts im Baum geordnet. Das elementare Intervall zum Blatt b heißt $Int(b)$.

Die **inneren Knoten** v von T korrespondieren zur Vereinigung der elementaren Intervalle, die im Baum unterhalb angeordnet sind. $Int(v)$ ist die Vereinigung aller $Int(b)$ im Teilbaum mit Wurzel v.

Jeder Knoten oder jedes Blatt v speichert das Intervall $Int(v)$ und eine Menge $I(v) \subseteq I$ von Intervallen (etwa in einer verknüpften Liste). Diese **kanonische Teilmenge** des Knotens v enthält die Intervalle $[x, x'] \in I$, so dass

$$Int(v) \subseteq [x, x'] \quad \wedge \quad Int\big(parent(v)\big) \not\subseteq [x, x']$$

5.3. Segmentbäume (Segment Trees)

Lemma 5.10: Ein Segmentbaum benötigt $O(n \log n)$ Speicher.

Beweis: Da T ein balancierter binärer Suchbaum mit maximal $4n + 1$ Blättern ist, hat er die Höhe $O(\log n)$. Wir behaupten, dass jedes Intervall $[x, x'] \in I$ in der Menge $I(v)$ von maximal zwei Knoten der gleichen Tiefe im Baum liegt. Seien v_1, v_2, v_3 drei Knoten (in dieser Reihenfolge) gleicher Tiefe im Baum. Wenn $[x_1, x_2]$ bei v_1 und v_3 gespeichert ist, liegt das Intervall vom linken Endpunkt von $Int(v_1)$ bis zum rechten Endpunkt von $Int(v_3)$ in $[x_1, x_2]$. Da $Int(v_2)$ dazwischen liegt, sind $Int(v_2)$ und $Int\big(parent(v_2)\big)$ Teilmengen von $[x_1, x_2]$. Also ist $[x_1, x_2]$ nicht in v_2 abgelegt. Für feste Tiefe gibt es also nur zwei Zeiger auf $[x_1, x_2]$. Insgesamt reichen $O(n \log n)$ Speicherplätze.

QED

5.3. Segmentbäume (Segment Trees)

Die Suche nach den Intervallen erfolgt nun rekursiv mit den Startwerten $(root(T), q_x)$.

Algorithm QUERYSEGMENTTREE(v, q_x)
Input. The root of a (subtree of a) segment tree and a query point q_x.
Output. All intervals in the tree containing q_x.
1. Report all the intervals in $I(v)$.
2. **if** v is not a leaf
3. **then if** $q_x \in \text{Int}(lc(v))$
4. **then** QUERYSEGMENTTREE($lc(v), q_x$)
5. **else** QUERYSEGMENTTREE($rc(v), q_x$)

Ohne Probleme erkennen wir:

Lemma 5.11: Mit einem Segmentbaum können die Intervalle, die einen Suchpunkt q_x enthalten, in $O\big((\log n) + k\big)$ berichtet werden, wobei k die Anzahl der gesuchten Intervalle ist.

5.3. Segmentbäume (Segment Trees)

Der Aufbau eines Segmentbaumes erfolgt durch Sortieren der Endpunkte in $O(n \log n)$ Zeit und anschließendem Aufbau des binären Suchbaumes T in $O(n)$ Zeit. Das Einfügen der Intervalle erfolgt jeweils durch einen Aufruf der folgenden rekursiven Funktion mit $(root(t), [x, x'])$.

Algorithm INSERTSEGMENTTREE(v, $[x : x']$)
Input. The root of a (subtree of a) segment tree and an interval.
Output. The interval will be stored in the subtree.

```
1.  if Int(v) ⊆ [x : x']
2.     then store [x : x'] at v
3.     else if Int(lc(v)) ∩ [x : x'] ≠ ∅
4.             then INSERTSEGMENTTREE(lc(v), [x : x'])
5.          if Int(rc(v)) ∩ [x : x'] ≠ ∅
6.             then INSERTSEGMENTTREE(rc(v), [x : x'])
```

5.3. Segmentbäume (Segment Trees)

An jedem Knoten v speichern wir das Intervall oder $Int(v)$ enthält einen Endpunkt. Nach Lemma 5.10. gibt es nur zwei Knoten für jede Tiefe, in denen $[x, x']$ abgelegt wird, also besuchen wir bis zu 4 Knoten pro Tiefe, insgesamt also $O(n \log n)$.

Theorem 5.12: Ein Segmentbaum für eine Menge I von n Intervallen nutzt $O(n \log n)$ Speicher und kann in $O(n \log n)$ Zeit gebaut werden. Mit diesem Segmentbaum können wir alle Intervalle, die einen Punkt q_x enthalten, in $O\big((\log n) + k\big)$ Zeit ermitteln, wobei k die Anzahl gesuchter Intervalle ist.

Für eine solche Suchanfrage ist der Intervallbaum wegen seines $O(n)$ Speicherbedarfs in aller Regel besser. Der Vorteil des Segmentbaums ist, dass wir in den Knoten ausschließlich die gesuchten Intervalle haben, so dass wir darauf weitere assoziierte Strukturen aufbauen können.

5.3. Segmentbäume (Segment Trees)

Wir kehren nun wieder zu unserem Fensterproblem zurück. Wir haben eine Menge S beliebig orientierter, disjunkter Segmente in der Ebene. Wir wollen alle Segmente finden, die ein vertikales Zielsegment $q := q_x \times [q_y, q_y']$ schneiden. Wir bauen einen Segmentbaum T mit den x-Intervallen der Segmente in S. Ein Knoten $v \in T$ gehört dann zu einem vertikalen Streifen der Form $Int(I) \times (-\infty, \infty)$ und die Intervalle in v zu Segmenten, die den Streifen komplett durchlaufen, nicht aber den Streifen zu $parent(v)$. Diese Segmente bilden die Menge $S(v)$. Wenn wir mit q_x durch T laufen, finden wir $O(\log n)$ kanonische Teilmengen. Da die Segmente in einem $S(v)$ sich nicht schneiden und den ganzen Streifen überspannen, können wir sie nach der y-Koordinate sortieren, also in einem entsprechenden Suchbaum organisieren.

5.3. Segmentbäume (Segment Trees)

Für S nutzen wir also als Datenstruktur

einen Segmentbaum T der x-Intervalle der Segmente,

wobei die kanonische Teilmengen an jedem Knoten v als binärer Suchbaum $T(v)$ basierend auf der vertikalen Ordnung der Segmente im Streifen abgelegt werden.

5.3. Segmentbäume (Segment Trees)

Der Speicheraufwand der assoziierten Suchbäume ist linear in $m = \#S(v)$, also reicht insgesamt $O(n \log n)$ Speicher. Der Aufbau der assoziierten Strukturen erfordert $O(m \log m)$ Zeit, so dass $O\big(n(\log n)^2\big)$ Zeit benötigt wird. Behält man eine entsprechend aktualisierte Sortierung der Segmente während des Aufbaus des Segmentbaumes bei, sinkt dies auf $O(n \log n)$. Als Suchzeit ergibt sich $O\big((\log m) + k_v\big)$ in den assoziierten Bäumen, also insgesamt $O\big((\log n)^2 + k\big)$.

Theorem 5.13: Sei S eine Menge von n disjunkten Segmenten in der Ebene. Die Segmente, die ein gegebenes vertikales Segment schneiden, können in $O\big((\log n)^2 + k\big)$ Zeit gefunden werden, wobei $O(n \log n)$ Speicher für eine Suchstruktur benötigt wird, die in $O(n \log n)$ Zeit erstellt werden kann. k ist die Anzahl gesuchter Elemente.

5.3. Segmentbäume (Segment Trees)

Korollar 5.14: Sei S eine Menge von n Segmenten in der Ebene mit disjunktem Inneren. Die ein gegebenes vertikales Segment schneidenden Segmente können in $O\big((\log n)^2 + k\big)$ Zeit mit Hilfe einer Suchstruktur gefunden werden, deren Aufbau $O(n \log n)$ Zeit und $O(n \log n)$ Speicher erfordert. k ist die Anzahl gesuchter Segmente.

5.3. Segmentbäume (Segment Trees)

Literatur

Der Intervallbaum für Streifenanfragen (stabbing queries) stammt von Edelsbrunner [H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Aurtria, 1980] und McCreight [E. M. McCreight. Efficient algorithms for enumerating intersection intervals and rectangles. Report CSL-80-9, Xerox Palo Alto Res. Center, Palo Alto, CA, 1980].

Der Suchbaum mit Priorität stammt ebenfalls von McCreight [E, M. McCreight. Priority search trees. SIAM J. Comput., 14:257-276, 1985]

5.3. Segmentbäume (Segment Trees)

Literatur

Der Segmentbaum wurde von Bentley [J. L. Bentley. Solutions to Klee's rectangle problmes. Technical report, Carnegie Mellon Univ., Pittsburgh, PA, 1977] erfunden.

Es gibt viele Erweiterungen auf höhere Dimensionen [B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Algorithmica, 11:116-132, 1994, H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980, J. Matousek, M. Sharir, and E. Welzl. A subexponentional bound for linear programming. Algrithmica, 16:498-516, 1996, M. H. Overmars. Geometric data structures for computer graphics: an overview. In R. A. Earnshaw, editor, Theoretical Foundations of Computer Graphics and CAD. NATO ASI Series F, vol. 40, pages 21-49. Springer-Verlag. 1988, V. K. Vaishnavi and D. Wood. Rectilinear line segment intersection,layered segment trees and dynamization. J. Algorithms, 3:160-176,1982].

5.3. Segmentbäume (Segment Trees)

Intervallbäume und Segmentbäume können auch dynamisiert werden, indem red-black-trees statt der binären Bäume verwendet werden [L. J. Guibas and Sedgewick. A dichromatic framework for balanced trees. In Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci., pages 8-21, 1978, T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. The MIT Press, Cambridge, MA, 1990].

Bei Overmars [M. H. Overmars. The Design of Dynamic Data Structures. Lecture Notes in Computer Science, vol. 156. Springer-Verlag, 1983] finden sich Ideen zur Zerlegung von Suchproblemen.

5.3. Segmentbäume (Segment Trees)

Es sei erwähnt, dass man die Streifensuche verallgemeinern kann. Im d-dimensionalen Raum lassen sich alle Punkte in einem achsenparallelen Rechteck mit $O\left(n(\log n)^{d-1}\right)$ Speicher finden, wobei $O\left((\log n)^{d}\right)$ Suchzeit anfällt. Fractional Cascading reduziert die Suchzeit um einen Faktor und ein Intervallbaum auf unterste Ebene reduziert den Speicherbedarf um einen weiteren $\log n$–Faktor.

Intervallbäume und Suchbäume mit Priorität lassen sich, soweit bekannt, nicht erweitern. Dies geht nur für Segmentbäume und Bereichsbäume (Range Trees). Aber Intervallbäume und Suchbäume mit Priorität sind geeignete assoziierte Strukturen.

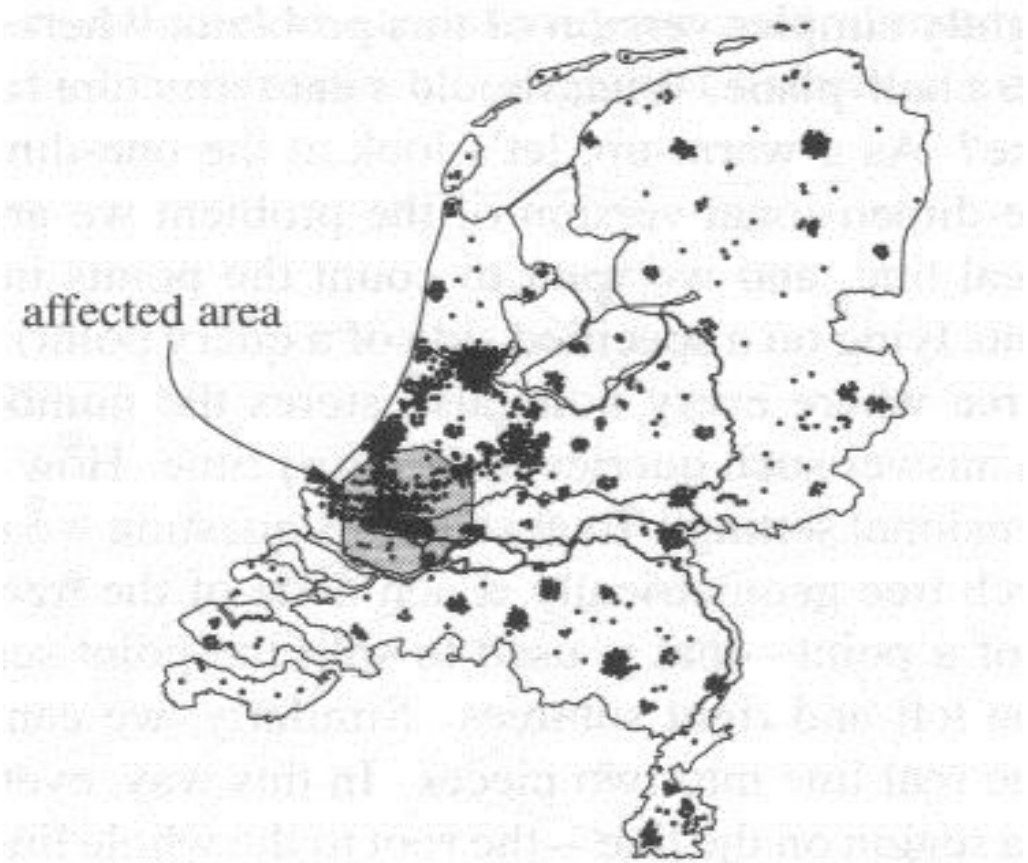
5.4. Unterteilungsbäume (Partition Trees)

Wir haben bisher Punkte und Linienstücke in Fenstern gesucht und zu einem Punkt die umgebende Facette einer planaren Unterteilung. Unsere Suchanfragen waren bisher also aus Punkten oder achsenparallelen Quadern aufgebaut. Wir wollen diese Beschränkung nun aufgeben und beliebige polygonale Suchbereiche zulassen, wobei wir uns durch Triangulierung auf Dreiecke beschränken können.

Als primäres Ziel unseres Algorithmus werden wir zunächst einfach die Punkte im Suchgebiet zählen, aber jede andere weitere Verarbeitung der gefundenen Punkte ist ebenfalls möglich.

5.4. Unterteilungsbäume (Partition Trees)

Population density of the Netherlands

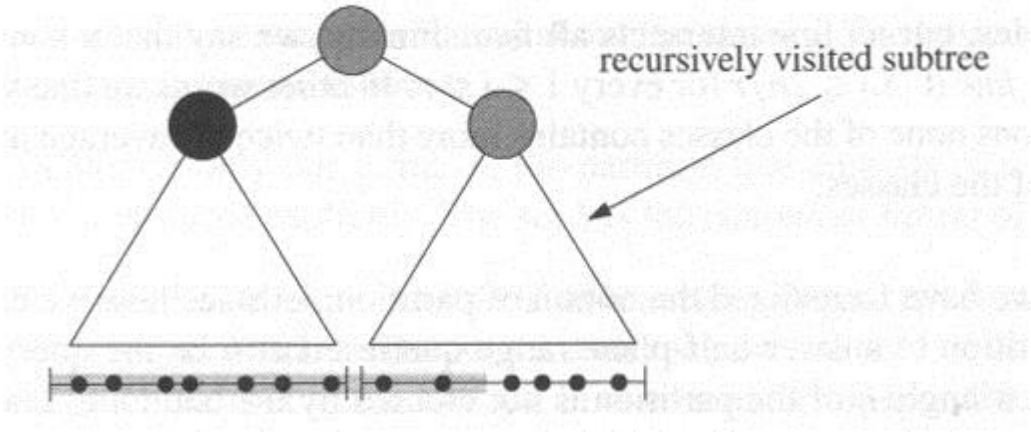


5.4. Unterteilungsbäume (Partition Trees)

Unser Problem kann in der folgenden Weise formalisiert werden:

Problem 5.15: Es sei eine Menge S von n Punkten in der Ebene und ein Dreieck T in der Ebene gegeben. Zähle die Punkte von S in T!

In 1D können wir die Punkte innerhalb einer Halbgeraden sicher in $O(\log n)$ mit Hilfe eines binären Suchbaumes zählen.



Answering a half-line query with a binary tree

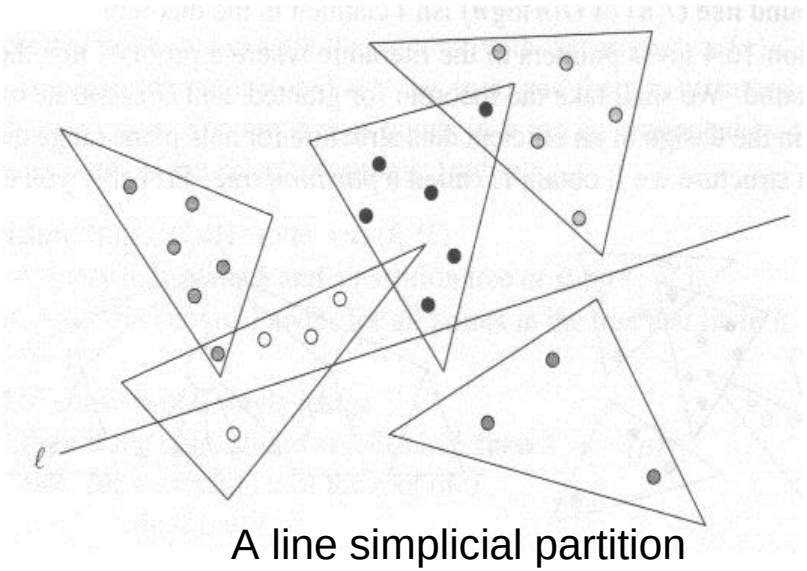
Leider kann man keine effektive Datenstruktur angeben, die für jede mögliche Halbebene einfach eine analoge Betrachtung zulässt, also S in Punkte innerhalb und in Punkte außerhalb zu zerlegen.

5.4. Unterteilungsbäume (Partition Trees)

Die Lösung liegt darin, mehr als nur zwei Regionen zu benutzen und dafür zu sorgen, dass man für eine beliebige Anfrage nur wenige Regionen genauer ansehen muss. Dazu nutzen wir simpliziale Zerlegungen.

Definitions 5.16: Eine **simpliziale Zerlegung** für eine Menge S von n Punkten in der Ebene ist eine Menge $\Psi(S) := \{(S_1, T_1), \ldots, (S_r, T_r)\}$, wobei die S_i disjunkte Teilmengen von S sind, deren Vereinigung ganz S ist. Jedes T_i sei ein Dreieck, das S_i enthält. Die S_i heißen **Klassen**. r heißt **Größe** von $\Psi(S)$.

Die Dreiecke müssen nicht disjunkt sein, aber ein Punkt darf nur zu einem Dreieck gehören.



A line simplicial partition

5.4. Unterteilungsbäume (Partition Trees)

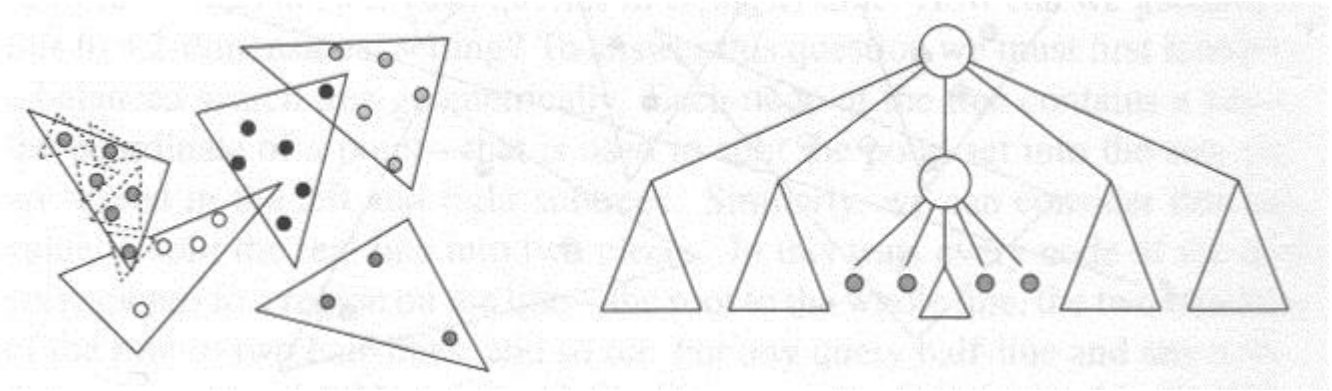
Eine Gerade l schneidet T_i, wenn sie das Innere von T_i trifft. Wenn die Punkte in S nicht in allgemeiner Lage sind, sind auch Liniensegmente an Stelle der Dreiecke zugelassen. Eine Gerade schneidet ein solches Segment, wenn sie sein Inneres trifft, aber nicht enthält.

Die **Schnittzahl von I bzgl. $\Psi(S)$** ist die Anzahl durch l geschnittener Dreiecke in $\Psi(S)$. Die **Schnittzahl von $\Psi(S)$** ist die maximale Schnittzahl aller Geraden bzgl. $\Psi(S)$.

Eine simpliziale Zerlegung heißt **fein**, falls $|S_i| \leq \frac{2n}{r}$ für $i = 1, \ldots, r$ gilt.

5.4. Unterteilungsbäume (Partition Trees)

Die Idee bei der Beantwortung der Suchanfrage ähnelt den Intervallbäumen. Bei einer Anfrage mit einer Halbebene gibt es Dreiecke, die ganz drin oder draußen liegen und damit leicht gezählt oder ausgegeben werden können. Die geschnittenen Dreiecke werden dann rekursiv weiter behandelt, wenn sie ebenfalls simplizial zerlegt sind.



Eine simpliziale Zerlegung und der zugehörige Unterteilungsbaum

5.4. Unterteilungsbäume (Partition Trees)

Offensichtlich beeinflusst die Schnittzahl die Komplexität der Suchanfrage.

Theorem 5.17: Für jede Menge S mit n Punkten in der Ebene und jede Zahl r, $1 \leq r \leq n$, gibt es eine feine simpliziale Zerlegung der Größe r mit Schnittzahl $O(\sqrt{r})$. Ferner kann man für jedes $\varepsilon > 0$ diese Zerlegung in $O(n^{1+\varepsilon})$ konstruieren.

[J. Matousek, Efficient Partition Trees. Discrete Computational Geometry 8:315-3134, 1992]

5.4. Unterteilungsbäume (Partition Trees)

Ein Unterteilungsbaum (partition tree) zu einer Menge S hat folgende Eigenschaften:

- Wenn S nur einen Punkt P enthält, ist der Unterteilungsbaum ein Blatt, das P enthält. S ist die **kanonische Teilmenge** des Blattes.

- Für $|S| > 1$ hat der Baum T bis zu r Kinder pro Knoten. Die Kinder der Wurzel repräsentieren die Dreiecke einer feinen simplizialen Zerlegung der Größe r von S. Das Dreieck zum Kind v heißt $T(v)$ und die Teilmenge von S heißt $S(v)$. Das Kind v ist die Wurzel eines Unterteilungsbaumes T_v für $S(v)$.

Mit jedem Kind v speichern wir das Dreieck $T(v)$ und die Anzahl Punkte in $S(v)$ für die Zählanfrage. Bei anderen Problemen wird andere Information gespeichert.

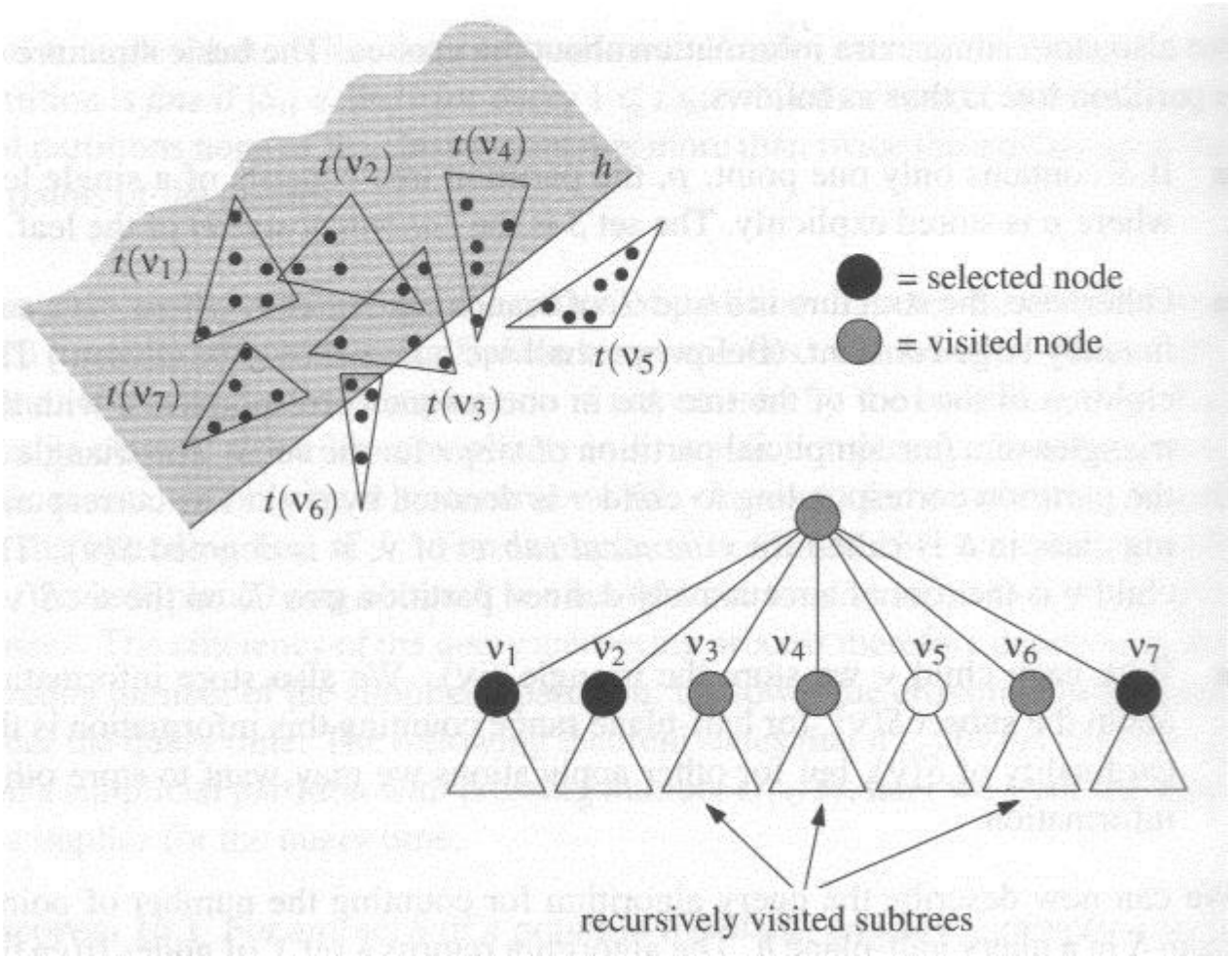
5.4. Unterteilungsbäume (Partition Trees)

Der Suchalgorithmus zu einer Halbebene h liefert nun eine Menge Y von Knoten des Unterteilungsbaumes, so dass gilt $S \cap h = \bigcup_{v \in Y} S(v)$.

Algorithm SELECTINHALFPLANE($h, \mathcal{T}$)
Input. A query half-plane h and a partition tree or subtree of it.
Output. A set of canonical nodes for all points in the tree that lie in h.

```
1.  Υ ← ∅
2.  if T consists of a single leaf μ
3.     then if the point stored at μ lies in h then Υ ← {μ}
4.     else for each child v of the root of T
5.              do if t(v) ⊂ h
6.                    then Υ ← Υ ∪ {v}
7.                    else if t(v) ∩ h ≠ ∅
8.                            then Υ ← Υ ∪ SELECTINHALFPLANE(h, T_v)
9.  return Υ
```

5.4. Unterteilungsbäume (Partition Trees)



Answering a half-plane range query using a partition tree

5.4. Unterteilungsbäume (Partition Trees)

Zunächst analysieren wir den Speicherbedarf.

Lemma 5.18: Sei S eine Menge mit n Punkten in der Ebene. Ein Unterteilungsbaum für S benötigt $O(n)$ Speicher.

Beweis: Sei $M(n)$ die maximale Anzahl Knoten eines Unterteilungsbaumes und $n_v := |S_v|$. Dann gilt

$$M(n) \leq \begin{cases} 1 & n = 1 \\ 1 + \sum_v M(n_v) & n > 1 \end{cases}$$

Es ist $\sum_v n_v = n$ und $n_v \leq \frac{2n}{r}$ für alle v. Für $r > 2$ ergibt sich dann $M(n) = O(n)$. Der Speicherbedarf eines Knotens ist $O(r)$.

QED

5.4. Unterteilungsbäume (Partition Trees)

Anschließend betrachten wir die Laufzeit der Suchanfrage.

Lemma 5.19: Sei S eine Menge von n Punkten in der Ebene. Für jedes $\varepsilon > 0$ gibt es einen Unterteilungsbaum für S, so dass wir für eine Halbebenenanfrage h $O(n^{0.5+\varepsilon})$ Knoten des Baumes auswählen können, die genau die Punkte in h enthalten. Halbebenenanfragen zum Zählen der Punkte können somit in $O(n^{0.5+\varepsilon})$ Zeit beantwortet werden.

5.4. Unterteilungsbäume (Partition Trees)

Beweis: Sei $\varepsilon > 0$ gegeben. Nach Theorem 5.17 gibt es eine Konstante c, so dass zu jedem r eine feine simpliziale Unterteilung der Größe r mit Schnittzahl höchstens $c \cdot \sqrt{r}$ existiert. Wir setzen $r := 2 \cdot \left(c \cdot \sqrt{2}\right)^{\frac{1}{\varepsilon}}$. Sei $Q(n)$ die maximale Suchzeit für eine Suche mit n Punkten, h die Halbebene und $n_v := |S(v)|$.

Dann gilt

$$Q(n) \leq \begin{cases} 1 & n = 1 \\ 1 + \sum_{v \in C(h)} Q(n_v) & n > 1 \end{cases}$$

wobei die Summe über die Menge $C(h)$ der Kinder v der Wurzel läuft, deren Dreiecke $T(v)$ von h geschnitten wird.

Es ist $|C(h)| \leq c \cdot \sqrt{r}$.

Ferner gilt $n_v \leq \frac{2n}{r}$ für alle v.

Dies liefert $Q(n) = O(n^{0.5+\varepsilon})$.

QED

5.4. Unterteilungsbäume (Partition Trees)

Man kann wegen der Schranke $O(n^{0.5})$ enttäuscht sein. Aber dies ist der Preis für $O(n)$ Speicherbedarf.

Das allgemeine Prinzip hinter allen Suchalgorithmen der Vorlesung ist, dass die Antwort aus charakteristischen (kanonischen) Antworten (characteristic subsets) zusammengesetzt wird. Je mehr Antworten in der Suchstruktur abgelegt werden, desto weniger braucht man, um eine beliebige Anfrage zu beantworten. Also erlaubt mehr Speicher eine schnellere Suche. In 5.5 werden wir $O(\log n)$ Suchzeit mit Hilfe von $O(n^2)$ Speicher erreichen.

5.4. Unterteilungsbäume (Partition Trees)

Um statt Halbebenen Dreiecke zu benutzen, bleiben Datenstrukturen und Algorithmus gleich! Man muss nur die Suchzeit analysieren. Bei einem Dreieck steigt die Schnittzahl einfach von $c \cdot \sqrt{r}$ nach $3 \cdot c \cdot \sqrt{r}$. Mit größerem r gilt jedoch die gleiche Grenze.

Theorem 5.20: Sei S eine Menge von n Punkten in der Ebene. Für jedes $\varepsilon > 0$ gibt es eine Datenstruktur für S, die $O(n)$ Speicher nutzt, so dass die Punkte von S innerhalb eines Dreieckes in $O(n^{0.5+\varepsilon})$ Zeit gezählt werden können. Die Punkte können in $O(k)$ zusätzlicher Zeit ausgegeben werden, wobei k die Anzahl der Punkte ist. Die Datenstruktur kann in $O(n^{1+\varepsilon})$ Zeit gebaut werden.

5.4. Unterteilungsbäume (Partition Trees)

Beweis: Bis auf Konstruktionszeit und Punktausgabe ist alles gezeigt. Die Konstruktion erfolgt rekursiv. Sei $T(n)$ die Zeit und $\varepsilon > 0$ gegeben. Nach Theorem 5.17 kann eine feine simpliziale Unterteilung von S der Größe r mit Schnittzahl $O(\sqrt{r})$ in $O(n^{1+\varepsilon'})$ für jedes $\varepsilon' > 0$ gebaut werden. Mit $\varepsilon' = \frac{\varepsilon}{2}$ ergibt sich

$$T(n) \leq \begin{cases} O(1) & n = 1 \\ O\left(n^{1+\frac{\varepsilon}{2}}\right) + \sum_v T(n_v) & n > 1 \end{cases}$$

wobei die Summe über die Kinder des Baumes läuft. Es gilt $\sum_v n_v = n$ und es folgt $T(n) = O(n^{1+\varepsilon})$ für die Rekurrenzrelation.

Die Punktausgabe erfordert das Durchlaufen der ausgewählten Teilbäume. Wenn die inneren Knoten eines Baumes alle mindestens 2 Kinder haben, hat der Baum *O*(#Blätter) Knoten, also folgt $O(k)$.

QED

5.4. Unterteilungsbäume (Partition Trees)

Unterteilungsbäume mit mehreren Ebenen

Unterteilungsbäume sind mächtige Werkzeuge, insbesondere wenn man an den Knoten assoziierte Strukturen für die charakteristischen Teilmengen aufhängt. Dies nutzen wir nun, um alle eine beliebige Gerade schneidenden Linienstücke zu berechnen.

Problem 5.21: Sei S eine Menge von n Liniensegmenten in der Ebene. Zähle die Segmente, die eine Gerade l schneiden!

Sind $P_{right}(s)$ und $P_{left}(s)$ die rechten und linken Endpunkte des Liniensegments s, so schneidet l das Liniensegment s, wenn die Endpunkte auf verschiedenen Seiten liegen oder ein Endpunkt auf l liegt. Wir zeigen, wie man die Segmente mit $P_{right}(s)$ oberhalb und $P_{left}(s)$ unterhalb von l zählt. Der umgekehrte Fall ist natürlich analog lösbar. Für eine vertikale Gerade l sei dazu der linke Teil unterhalb l.

5.4. Unterteilungsbäume (Partition Trees)

Die Strategie ist einfach. Wir nutzen einen Unterteilungsbaum, um im ersten Schritt alle Segmente s mit $P_{right}(s)$ oberhalb l zu finden. Zu den charakteristischen Mengen bilden wir jedoch einen assoziierten Unterteilungsbaum mit den linken Endpunkten, so dass wir aus der Menge der Segmente mit $P_{right}(s)$ oberhalb l diejenigen mit $P_{left}(s)$ unterhalb l herausfinden können.

Formal sei für eine Teilmenge S' von S die Menge der rechten Endpunkte $P_{right}(S') := \{P_{right}(s) | s \in S'\}$ und die Menge der linken Endpunkte $P_{left}(S') := \{P_{left}(s) | s \in S'\}$ definiert. Die Datenstruktur ergibt sich als:

- Die Menge $P_{right}(S)$ wird in einem Unterteilungsbaum T gespeichert.
  Die kanonische Teilmenge zu v in T sei $P_{right}(v)$.
  Ferner sei $S(v) := \{s \in S | P_{right}(s) \in P_{right}(v)\}$.
- Mit jedem Knoten v des Baumes T der ersten Ebene speichern wir $P_{left}(S(v))$ in einem Unterteilungsbaum ${T_v}^{assoc}$ für das Halbebenenzählen. Dieser Unterteilungsbaum heißt assoziierte Struktur für v.

5.4. Unterteilungsbäume (Partition Trees)

Algorithm SELECTINTSEGMENTS($\ell, \mathcal{T}$)
Input. A query line ℓ and a partition tree or subtree of it.
Output. A set of canonical nodes for all segments in the tree that are intersected by ℓ.

```
1.  Υ ← ∅
2.  if T consists of a single leaf μ
3.     then if the segment stored at μ intersects ℓ then Υ ← {μ}
4.     else for each child v of the root of T
5.             do if t(v) ⊂ ℓ+
6.                   then Υ ← Υ ∪ SELECTINHALFPLANE(ℓ−, T_v^assoc)
7.                   else if t(v) ∩ ℓ ≠ ∅
8.                           then Υ ← Υ ∪ SELECTINTSEGMENTS(ℓ, T_v)
9.  return Υ
```

Um die Segmente mit linkem Endpunkt oberhalb und rechtem Endpunkt unterhalb zu finden, genügt der Tausch l^+ mit l^-!

5.4. Unterteilungsbäume (Partition Trees)

Lemma 5.22: Sei S eine Menge von n Liniensegmenten in der Ebene. Ein Unterteilungsbaum mit 2 Ebenen für Segmentschnitte mit Geraden benötigt $O(n \log n)$ Speicher.

Beweis: Sei $n_v := |S(v)|$. Der Unterteilungsbaum $T^{assoc}(S(v))$ benötigt nach 5.18 $O(n_v)$ Speicher. Für den Gesamtspeicher $M(n)$ gilt

$$M(n) \leq \begin{cases} O(1) & n = 1 \\ \sum_v \left(O(n_v) + M(n_v)\right) & n > 1 \end{cases}$$

wobei die Summe über alle Kinder v läuft. Mit $\sum_v n_v = n$, $n_v \leq \frac{2n}{r}$ und $r > 2$ ergibt sich $M(n) = O(n \log n)$.

QED

5.4. Unterteilungsbäume (Partition Trees)

Lemma 5.23: Sei S eine Menge von n Liniensegmenten in der Ebene. Für jedes $\varepsilon > 0$ gibt es einen Unterteilungsbaum mit 2 Ebenen, so dass wir für eine Gerade l $O(n^{0.5+\varepsilon})$ Knoten aus dem Baum auswählen können, die genau die l schneidenden Segmente von S enthalten. Die Auswahl der Knoten erfordert $O(n^{0.5+\varepsilon})$ Zeit, so dass die Anzahl in $O(n^{0.5+\varepsilon})$ ermittelt werden kann.

Beweis: Sie $\varepsilon > 0$, $n_v := |S_V|$. Nach Lemma 5.19 ist die Suchzeit in ${T_v}^{assoc}$ $O({n_v}^{0.5+\varepsilon})$. Unseren Baum T wählen wir $r := \left\lceil 2 \cdot \sqrt{c \cdot \sqrt{2}} \right\rceil$. Dann gilt für die Suchzeit

$$Q(n) \leq \begin{cases} O(1) & n = 1 \\ O(r \cdot n^{0.5+\varepsilon}) + \sum_{i=1}^{c \cdot \sqrt{r}} Q\left(\frac{2n}{r}\right) & n > 1 \end{cases}$$

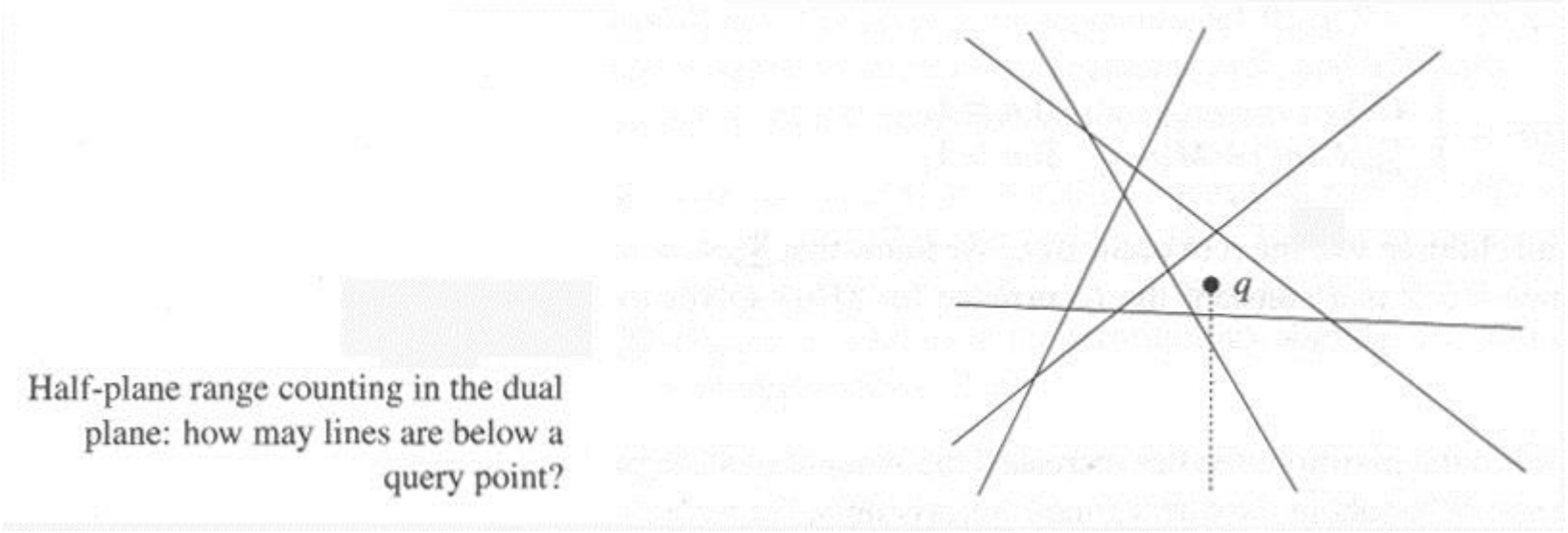
und dies ergibt $O(n^{0.5+\varepsilon})$.

QED

5.5. Cutting Trees

Da $O(\sqrt{n})$ Suchzeit lang erscheint, aber da simpliziale Unterteilungen mit weniger als $O(\sqrt{n})$ Schnittzahl nicht immer existieren, ist für einen $O(\log n)$ Suchalgorithmus für Simplexanfragen ein anderer Zugang nötig. Außerdem werden wir mehr als $O(n \log n)$ Speicher benötigen.

Um eine erste Idee zu gewinnen, betrachten wir eine Halbebenensuche im dualen Raum. Zu der Frage, wie viele Punkte in der Halbebene liegen, ist die Frage nach den unter einem Punkt liegenden Geraden äquivalent.



Half-plane range counting in the dual plane: how may lines are below a query point?

5.5. Cutting Trees

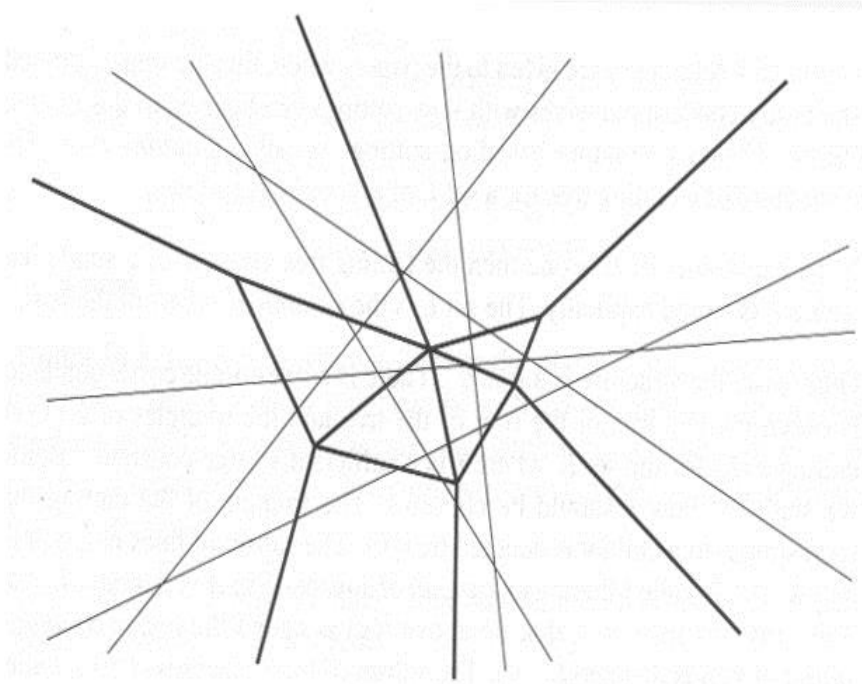
Dieses Problem lässt sich lösen, indem wir die Facette zu dem Suchpunkt ermitteln (Schnitte planarer Liniensegmente und trapezförmige Karte) und in jeder Facette die Anzahl darunter liegender Geraden zählen. Dies liefert $O(n^2)$ Speicherbedarf und $O(\log n)$ Suchzeit.

Leider ist ein analoger Zugang für Dreiecke nicht direkt möglich und wir müssen mit Unterteilungen der Ebene und ähnlichen Ideen wie bei den Unterteilungsbäumen arbeiten.

5.5. Cutting Trees

Wir bleiben bei unserer dualen Formulierung und zerlegen die Ebene in Dreiecke.

Sei also L eine Menge von n Geraden und r ein Parameter mit $1 \leq r \leq n$. Eine Gerade schneidet ein Dreieck, wenn sie das Innere trifft. Ein $1/r$**-Schnitt** für L ist eine Menge $\Xi(L) := \{t_1, \ldots, t_m\}$ von Dreiecken (möglicherweise unbeschränkt) mit disjunktem Inneren, welche die Ebene überdecken, wobei kein Dreieck mehr als n/r Geraden schneidet.



A (1/2)-cutting of size ten for a set of six lines

5.5. Cutting Trees

Theorem 5.24: Für jede Menge L von n Geraden in der Ebene und jedes r, $1 \leq r \leq n$, gibt es einen $1/r$-Schnitt mit Größe $O(r^2)$. Dieser Schnitt kann in $O(n \cdot r)$ Zeit berechnet werden.

Beweis:

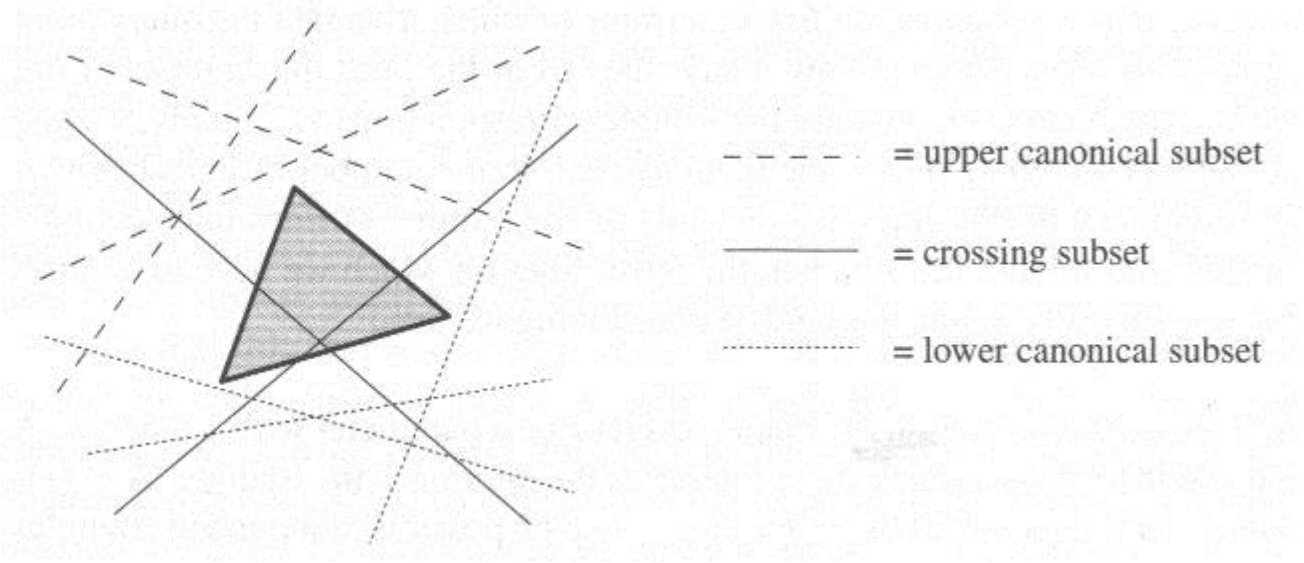
[B. Chazelle. Cutting hyperplanes for divide-and-conquer. Discrete Comput. Geom., 9:145-158, 1993.]

5.5. Cutting Trees

Ein **cutting tree** T für eine Menge L von n Geraden hat folgende Eigenschaften:

- Für $|L| = 1$ ist T ein Blatt mit der Geraden. L ist die kanonische Teilmenge des Blattes.
- Für $|L| > 1$ ist T ein Baum. Für jedes Dreieck $t(v)$ eines $1/r$-Schnittes $\Xi(L)$ gibt es ein Kind v.
  - Die Menge der Geraden unterhalb $t(v)$ heißt **untere kanonische Teilmenge** $L^-(v)$.
  - Die Menge der Geraden oberhalb $t(v)$ heißt **obere kanonische Teilmenge** $L^+(v)$.
  - Die Menge der $t(v)$ schneidenden Geraden heißt **schneidende Teilmenge** $L^c(v)$.
  - v ist Wurzel eines cutting trees für $L^c(v)$.
- Mit jedem Knoten v wird $t(v)$ gespeichert und Informationen zu $L^+(v)$ und $L^-(v)$, etwa die Anzahl der Geraden.

5.5. Cutting Trees



The canonical subsets and the crossing subset for a triangle

5.5. Cutting Trees

Algorithm SELECTBELOWPOINT($q, \mathcal{T}$)
Input. A query point q and a cutting tree or subtree of it.
Output. A set of canonical nodes for all lines in the tree that lie below q.

1. $\Upsilon \leftarrow \emptyset$
2. **if** $\mathcal{T}$ consists of a single leaf μ
3. **then if** the line stored at μ lies below q **then** $\Upsilon \leftarrow \{\mu\}$
4. **else for** each child ν of the root of $\mathcal{T}$
5. **do** Check if q lies in $t(\nu)$.
6. Let ν_q be the child such that $q \in t(\nu_q)$.
7. $\Upsilon \leftarrow \{\nu_q\} \cup$ SELECTBELOWPOINT$(q, \mathcal{T}_{\nu_q})$
8. **return** Υ

5.5. Cutting Trees

Lemma 5.25: Sei L eine Menge von n ebenen Geraden. Mit einem cutting tree können die Geraden unterhalb eines Punktes q in $O(\log n)$ Zeit in $O(\log n)$ kanonischen Teilmengen ausgewählt werden. Für jedes $\varepsilon > 0$ kann ein cutting tree mit $O(n^{2+\varepsilon})$ Speicher gebaut werden.

5.5. Cutting Trees

Beweis: Sei $Q(n)$ die Suchzeit für n Geraden. Es gilt

$$Q(n) = \begin{cases} 1 & n = 1 \\ Q(r^2) + Q\left(\frac{n}{r}\right) & n > 1 \end{cases}$$

Es folgt $Q(n) = O(\log n)$ für $r > 1$.

Sei $\varepsilon > 0$. Dann gibt es nach Theorem 5.24 einen $1/r$-Schnitt mit Größe $c \cdot r^2$.

Wir wählen $r = \lceil (2c)^{\frac{1}{\varepsilon}} \rceil$. Für den Speicherbedarf $M(n)$ gilt

$$M(n) = \begin{cases} O(1) & n = 1 \\ O(r^2) + \sum_{v=1}^{c \cdot r^2} M(n_v) & n > 1 \end{cases}$$

mit $n_v \leq \frac{n}{r}$.

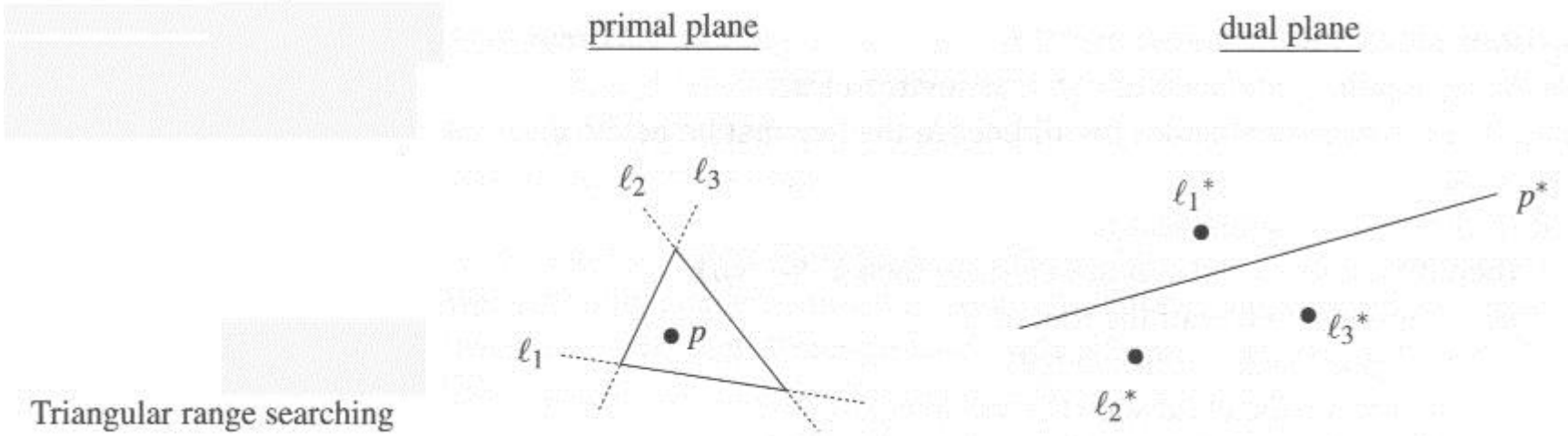
Es folgt $M(n) = O(n^{2+\varepsilon})$.

QED

5.5. Cutting Trees

Für die Simplexsuche in der Ebene betrachten wir ein Dreieck als Schnitt von drei Halbebenen.

Im Bild liegt p im Dreieck wegen $p \in {l_1}^+$, $p \in {l_2}^-$ und $p \in {l_3}^-$, also p^* unterhalb von $p \in {l_1}^*$ und oberhalb von $p \in {l_2}^*$, $p \in {l_3}^*$.



Triangular range searching

5.5. Cutting Trees

Das allgemeine Simplexsuchproblem in der Ebene lautet also dual formuliert:

Problem 5.26: Sei eine Menge L von n Geraden in der Ebene gegeben. Zu drei Punkten q_1, q_2 und q_3 mit Bezeichnern „unterhalb“ oder „oberhalb“ zähle die Geraden in L mit der durch die Bezeichner gegebenen Lage zu den Punkten.

Wir lösen das Problem zunächst nur für zwei Punkte und betrachten dann die einfache Verallgemeinerung auf das eigentliche Problem.

5.5. Cutting Trees

Ein 2-Ebenen cutting tree für eine Menge L von n Geraden zur Suche aller Geraden unterhalb von q_1, q_2 sei, wie folgt, definiert:

- Die Menge L ist in einem cutting tree T organisiert.
- Mit jedem Knoten v des cutting trees T speichern wir die untere kanonische Teilmenge als assoziierten cutting tree ${T_v}^{assoc}({L_v}^-)$.

5.5. Cutting Trees

Algorithm SELECTBELOWPAIR($q_1, q_2, \mathcal{T}$)
Input. Two query points q_1 and q_2 and a cutting tree or subtree of it.
Output. A set of canonical nodes for all lines in the tree that lie below q_1 and q_2.

1. $\Upsilon \leftarrow \emptyset$
2. **if** $\mathcal{T}$ consists of a single leaf μ
3. **then if** the line stored at μ lies below q_1 and q_2 **then** $\Upsilon \leftarrow \{\mu\}$
4. **else for** each child v of the root of $\mathcal{T}$
5. **do** Check if q_1 lies in $t(v)$.
6. Let v_{q_1} be the child such that $q_1 \in t(v_{q_1})$.
7. $\Upsilon_1 \leftarrow$ SELECTBELOWPOINT$(q_2, \mathcal{T}^{\text{assoc}}_{v_{q_1}})$
8. $\Upsilon_2 \leftarrow$ SELECTBELOWPAIR$(q_1, q_2, \mathcal{T}_{v_{q_1}})$
9. $\Upsilon \leftarrow \Upsilon_1 \cup \Upsilon_2$
10. **return** Υ

5.5. Cutting Trees

Lemma 5.27: Sei L eine Menge von n Geraden in der Ebene. Mit einem 2-Ebenen cutting tree können die Geraden L unterhalb eines Punktepaares in $O\left((\log n)^2\right)$ Zeit als $O\left((\log n)^2\right)$ kanonische Teilmengen gefunden werden. Für jedes $\varepsilon > 0$ kann der 2-Ebenen cutting tree mit $O(n^{2+\varepsilon})$ Speicher gebaut werden.

Beweis: Sei $Q(n)$ die Suchzeit. Nach Lemma 5.25 werden die assoziierten Strukturen in $O(\log n_v)$ durchsucht. Es gilt

$$Q(n) = \begin{cases} O(1) & n = 1 \\ O(r^2) + O(\log n) + Q\left(\frac{n}{4}\right) & n > 1 \end{cases}$$

und es folgt $Q(n) = O\left((\log n)^2\right)$ für $r > 1$.

5.5. Cutting Trees

Sei $\varepsilon > 0$. Nach Lemma 5.25 lassen sich die assoziierten Strukturen der Kinder der Wurzel in $O(n^{2+\varepsilon})$ aufbauen. Der Speicherbedarf $M(n)$ erfüllt

$$M(n) = \begin{cases} O(1) & n = 1 \\ \sum_v \left(O(n^{2+\varepsilon}) + M(n_v) \right) & n > 1 \end{cases}$$

mit $O(r^2)$ Kindern und $n_v \leq \frac{n}{r}$.

Es folgt $M(n) = O(n^{2+\varepsilon})$.

QED

5.5. Cutting Trees

Theorem: Sei S eine Menge von n Punkten in der Ebene. Für $\varepsilon > 0$ gibt es eine Datenstruktur namens cutting tree, die $O(n^{2+\varepsilon})$ Speicher benötigt und die Punkte in S innerhalb eines Dreieckes T in $O\big((\log n)^3\big)$ Zeit zählen kann. Die Ausgabe erfordert zusätzlich $O(k)$ Zeit, wobei k die Anzahl der Punkte von S in T ist. Die Datenstruktur kann in $O(n^{2+\varepsilon})$ Zeit gebaut werden.

5.5. Cutting Trees

Literatur:

Clarkson [K. L. Clarkson. New applications of random sampling in computational geometry. Discrete Comput. Geo., 2:195-222, 1987.] gelang die erste $O(\log n)$ Halbebenensuche mit Hilfe von Schnitten.

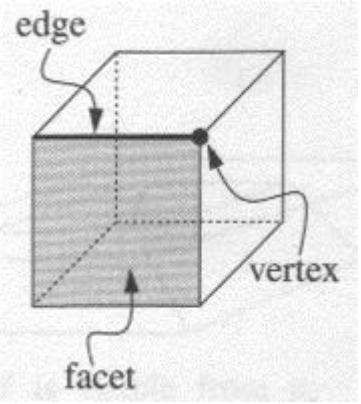
Der beste Algorithmus stammt von Chazelle [B. Chazelle. Cutting hyperplanes for divide-and-conquer. Discrete Comput. Geom., 9:145-158, 1993.], der $1/r$-Schnitte der Größe $O(r^d)$ mit einem deterministischen Algorithmus in $O(n \cdot r^{d-1})$ Zeit berechnete. Die Suchzeit lässt sich auf $O(\log n)$ reduzieren! Mit einer Kombination aus Unterteilungsbäumen und cutting trees lässt sich für m mit $n \leq m \leq n^d$ eine Datenstruktur der Größe $O(m^{1+\varepsilon})$ mit $O\left(\frac{n^{1+\varepsilon}}{m^{1/d}}\right)$ Suchzeit generieren.

5.6. Konvexe Hüllen in 3D

Der Begriff der konvexen Hülle von *n* Punkten p_1 , ... p_n lässt sich auf mehr als zwei Dimensionen erweitern. Neben der Definition als kleinster konvexer Menge, die die *n* Punkte enthält, kann man auch die Menge aller Konvexkombinationen der *n* Punkte nehmen, also

$$CH(P) = \{\sum_{i=1}^{n} \mu_i \, p_i | \mu_i \geq 0, \sum_{i=1}^{n} \mu_i = 1\}$$

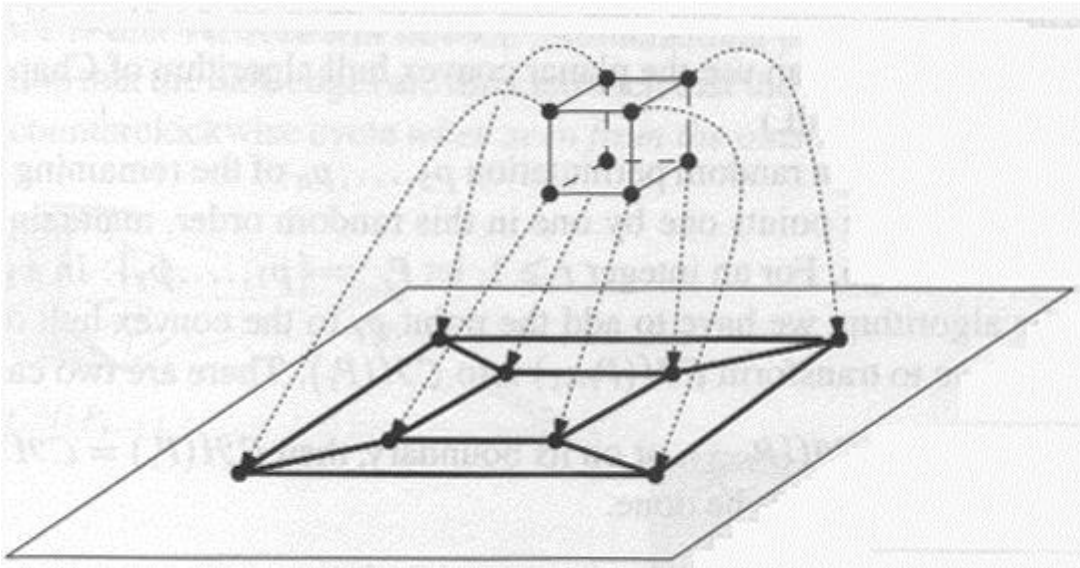
In 3D ergibt sich ein konvexer Polytop (durch ebene Polygone begrenzter Körper), dessen Ecken aus *P* sind. Man spricht von Ecken, Kanten und Facetten eine Polytops.



5.6. Konvexe Hüllen in 3D

Theorem 6.1: Sei *C* ein konvexes Polytop mit *n* Ecken. Die Anzahl der Kanten von *C* ist höchstens 3*n*-6 und die Anzahl der Facetten von *P* ist höchstens 2*n*-4.

Beweis: Man fasst den Rand des Polytops als planaren Graphen auf.



A cube interpreted as a planar graph: note that one facet maps to the unbounded face of the graph

5.6. Konvexe Hüllen in 3D

Jede Facette hat mindestens drei Kanten und jede Kante gehört zu genau 2 Facetten, also $2n_e >= 3n_f$. Aus der Eulerformel

$$n - n_e + n_f = 2$$

folgt

$$2\,n + 2\,n_f - 4 \;\geq 3\,n_f \;\rightarrow\; n_f \leq 2n \;- 4$$

Analog

$$0 = n - n_e + n_f - 2 \;\leq n - n_e + \frac{2}{3}2_e - 2 \leq n - \frac{1}{3}n_e - 2$$

und somit

$$n_e \leq 3n - 6$$

QED

5.6. Konvexe Hüllen in 3D

Korollar 6.2: Die Komplexität (Summe von Ecken, Kanten und Facetten) der konvexen Hülle von n Punkten in 3D ist $O(n)$.

5.6. Konvexe Hüllen in 3D

Algorithmus:

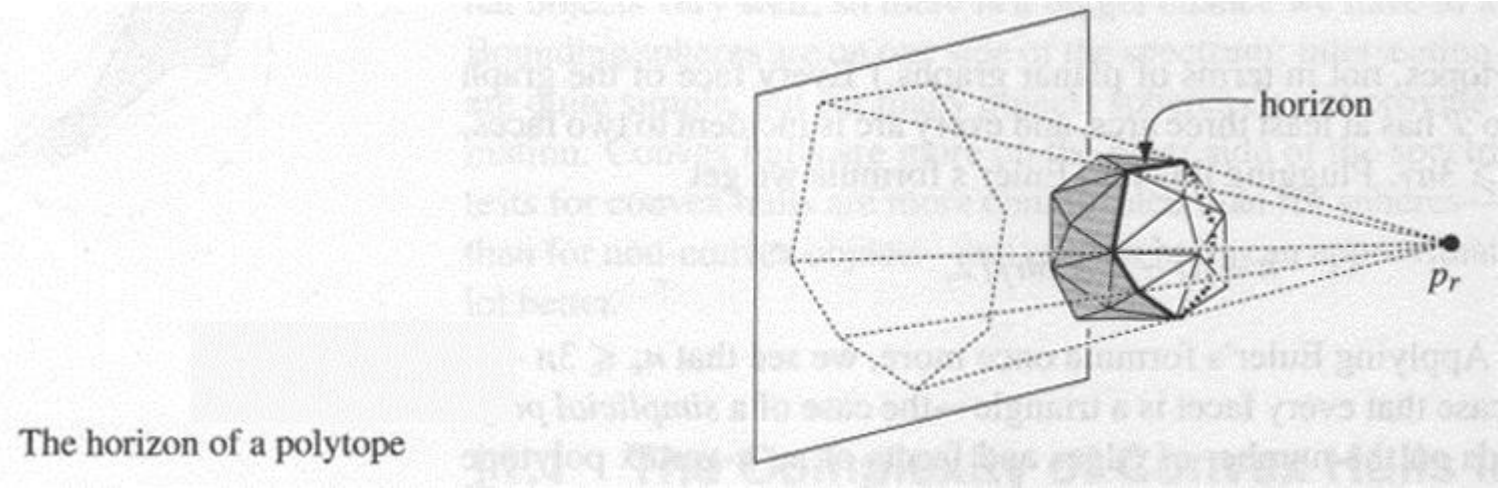
Der Algorithmus arbeitet inkrementell und randomisiert. Zunächst wählen wir vier nicht planare Punkte aus. Dann berechnen wir eine zufällige Permutation der übrigen Punkte.

Sei $P_r := \{p_1 , ... , p_r\}$. Der zentrale Schritt des Algorithmus fügt einen Punkt p_r zu $CH(P_{r-1})$ hinzu und bildet P_r. Es gibt zwei Fälle:

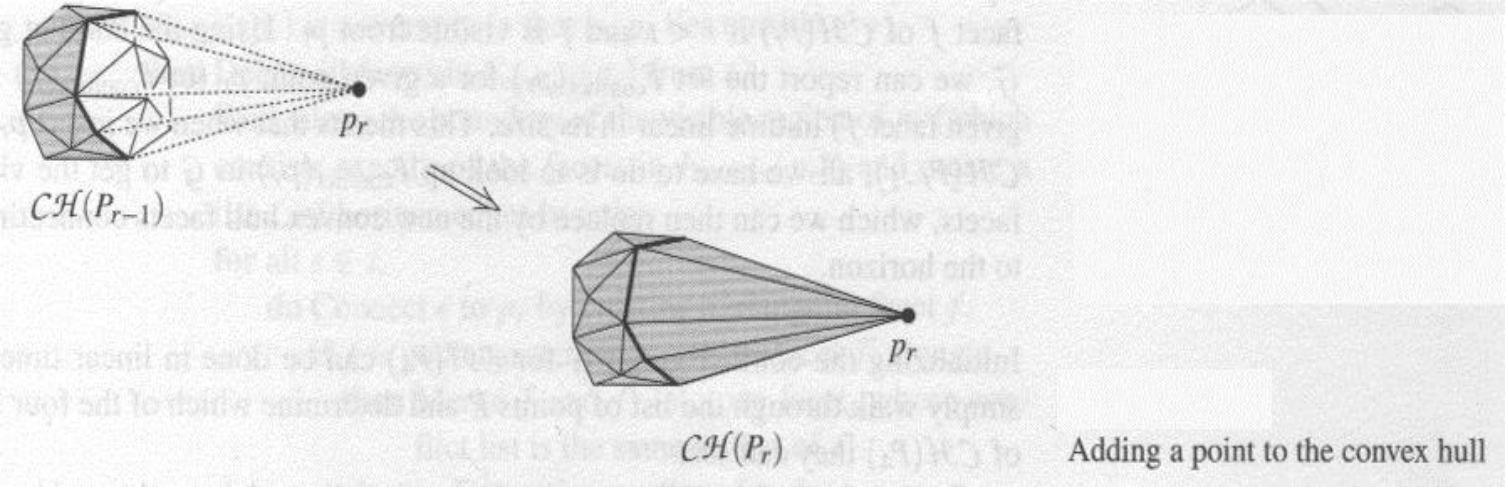
1. Wenn p_r innerhalb $CH(P_{r-1})$ oder auf dem Rand liegt, ist $CH(P_r) = CH(P_{r-1})$.

2. Liegt p_r außerhalb, so betrachte man $CH(P_{r-1})$ von p_r aus. Die sichtbaren Facetten bilden eine zusammenhängende Region auf der Oberfläche, die **von p_r sichtbare Region auf $CH(P_{r-1})$**. Deren Rand besteht aus einem geschlossenen Kantenzug, dem **Horizont L von p_r auf $CH(P_{r-1})$**. Ersetzt man die sichtbare Region von p_r auf $CH(P_{r-1})$ durch Dreiecke zwischen den Horizontkanten und p_r, so entsteht die neue konvexe Hülle.

5.6. Konvexe Hüllen in 3D



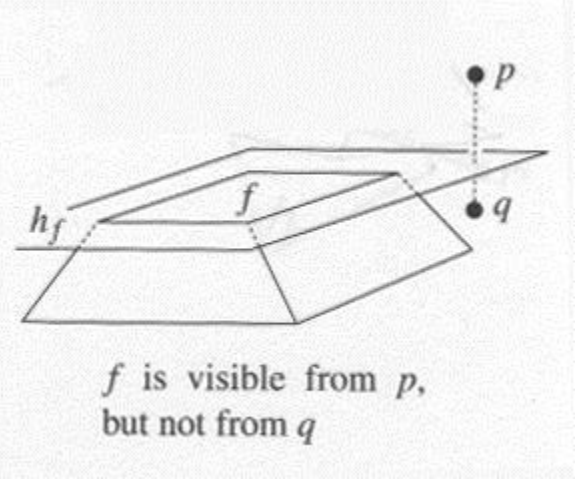
The horizon of a polytope



Adding a point to the convex hull

5.6. Konvexe Hüllen in 3D

Eine Facette f von $CH(P_{r-1})$ ist dabei sichtbar für p_r, falls p_r in dem Halbraum zu der Facette liegt, der nicht zu $CH(P_{r-1})$ gehört. (Hier wird $CH(P_{r-1})$ als Schritt der die Facetten definierenden Halbebenen aufgefasst.)



f is visible from p, but not from q

Die konvexe Hülle speichern wir als doppelt verknüpfte Kantenliste, wobei die Ecken nun 3D-Koordinaten tragen.

5.6. Konvexe Hüllen in 3D

Eine kleine Schwierigkeit gilt es zu beachten:

p_r kann in der Ebene einer Facette von $CH(P_{r-1})$ liegen. Dann erzeugen wir eine zweite Facette in der gleichen Ebene, die wir mit der ersten vereinigen müssen.

Ferner führt die naive Berechnung der sichtbaren Facetten durch einen direkten Test aller gegenwärtigen Facetten zu $O(r)$ Aufwand und wegen

$$\sum_{r=4}^{n} r = O(n^2)$$

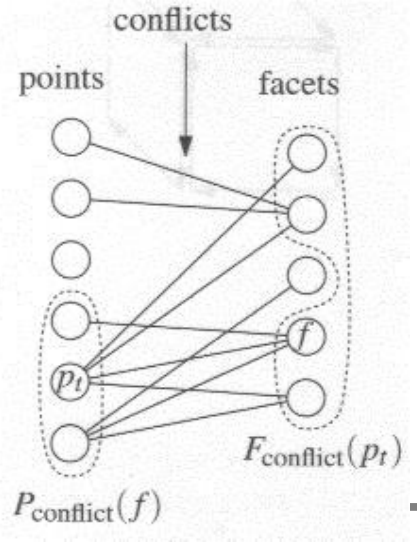
zu quadratischen Aufwand.

5.6. Konvexe Hüllen in 3D

Stattdessen führen wir Konfliktlisten ein:

- Für jede Facette f von $CH(P_r)$ sei $P_{conflict}(f) \subset \{p_{r+1}, \ldots, p_n\}$ die Menge der Punkte , die f sehen können.
- Für jeden Punkt p_t , $t > r$, sei $F_{\text{conflict}}(p_t)$ die Menge der Facetten, die p_t sehen kann.

Diese Mengen werden als Konfliktgraph G gespeichert, der bipartit ist. Es gibt eine Menge Knoten der Facetten in $CH(P_r)$ und eine Menge Knoten der Punkte $\{p_{r+1}, \ldots, p_n\}$ und die Konflikte werden als Kanten abgespeichert.



5.6. Konvexe Hüllen in 3D

Die Initialisierung des Konfliktgraphen erfordert lineare Zeit. Danach sind stets die Facetten der sichtbaren Region und *pr* zu entfernen und die Sichtbarkeit der neuen Facetten zu prüfen, also neue Kanten und Knoten einzufügen. Bei koplanaren Facetten erhält die vereinigte Facette die Kanten der alten Facette, die mit einer neuen vereinigt wird.

5.6. Konvexe Hüllen in 3D

Algorithm CONVEXHULL(P)
Input. A set P of n points in three-space.
Output. The convex hull $\mathcal{CH}(P)$ of P.

1. Find four points p_1, p_2, p_3, p_4 in P that form a tetrahedron.
2. $\mathcal{C} \leftarrow \mathcal{CH}(\{p_1, p_2, p_3, p_4\})$
3. Compute a random permutation $p_5, p_6, \ldots, p_n$ of the remaining points.
4. Initialize the conflict graph $\mathcal{G}$ with all visible pairs (p_t, f), where f is a facet of $\mathcal{C}$ and $t > 4$.
5. **for** $r \leftarrow 5$ **to** n
6. **do** (∗ Insert p_r into $\mathcal{C}$: ∗)
7. **if** $F_{\text{conflict}}(p_r)$ is not empty (∗ that is, p_r lies outside $\mathcal{C}$ ∗)
8. **then** Delete all facets in $F_{\text{conflict}}(p_r)$ from $\mathcal{C}$.
9. Walk along the boundary of the visible region of p_r (which consists exactly of the facets in $F_{\text{conflict}}(p_r)$) and create a list $\mathcal{L}$ of horizon edges in order.
10. **for** all $e \in \mathcal{L}$
11. **do** Connect e to p_r by creating a triangular facet f.
12. **if** f is coplanar with its neighbor facet f' along e
13. **then** Merge f and f' into one facet, whose conflict list is the same as that of f'.
14. **else** (∗ Determine conflicts for f: ∗)
15. Create a node for f in $\mathcal{G}$.
16. Let f_1 and f_2 be the facets incident to e in the old convex hull.
17. $P(e) \leftarrow P_{\text{conflict}}(f_1) \cup P_{\text{conflict}}(f_2)$
18. **for** all points $p \in P(e)$
19. **do** If f is visible from p, add (p, f) to $\mathcal{G}$.
20. Delete the node corresponding to p_r and the nodes corresponding to the facets in $F_{\text{conflict}}(p_r)$ from $\mathcal{G}$, together with their incident arcs.
21. **return** $\mathcal{C}$

5.6. Konvexe Hüllen in 3D

Analyse

Lemma 6.3: Die erwartete Anzahl der von CONVEXHULL erzeugten Facetten ist höchstens $6n - 20$.

Beweis: Zu Beginn erzeugen wir 4 Facetten. Die Anzahl der Facetten in Zeile 11 hängt von der Anzahl Kanten im Horizont ab. Wir nutzen wieder eine Rückwärtsanalyse: Wir betrachten $CH(P_r)$ und stellen uns das Entfernen von p_r vor. Dies betrifft genau die Facetten inzident zu p_r, kurz deg (p_r, $CH(P_r)$). Nach 6.1 gibt es maximal $3r - 6$ Kanten, also

$$\sum_{s=1}^{r} \deg(p_s, CH(P_r)) = 6\,r\, - 12$$

Da wir p_r aus $\{p_5, \ldots, p_r\}$ zufällig gewählt haben und $p_1, \ldots, p_4$ zusammen einen Grad von mindestens 12 haben, folgt:

5.6. Konvexe Hüllen in 3D

$$E[\deg(p_r, CH(P_r))] = \frac{1}{r-4}\sum_{i=5}^{r}\deg(p_i, CH(P_r))$$

$$\leq \frac{1}{r-4}\left(\left\{\sum_{i=1}^{r}\deg(p_i, CH(P_r))\right\} - 12\right)$$

$$\leq \frac{6r - 12 - 12}{r-4}$$

$$= 6$$

Gesamtzahl der erzeugten Facetten:

$$4 + \sum_{r=5}^{n} E[deg(p_r, CH(P_r))] \leq 4 + 6\,(n-4) = 6\,n - 20$$

QED

5.6. Konvexe Hüllen in 3D

Lemma 6.4: CONVEXHULL berechnet die konvexe Hülle der Menge *P* von *n* Punkten im IR^3 in O(*n* log *n*) erwarteter Zeit, wobei sich der Erwartungswert auf die zufällige Permutation der Punkte im Algorithmus bezieht.

Beweis: Die Schritte vor der Schleife erfordern max. *O*(*n* log *n*) Zeit und für p_r in $CH(P_{r-1})$, also Fconflict (*pr*) = { } ist der Aufwand konstant. Wenn $F_{conflict}(p_r)$ <>{ }, so erfordern alle Zeilen außer 17-20 gerade *O*(card ($F_{conflict}$ (p_r))) Zeit. Diese zu löschenden Facetten müssen auch erzeugt werden, also

$$E\left[\sum_{r=5}^{n} card\left(F_{conflict}(p_r)\right)\right] = O(n)$$

Zeile 20 ist linear in der Anzahl der Knoten und Kanten, die gelöscht werden und dies bleibt *O*(*n*), da kein Element doppelt erzeugt wird.

5.6. Konvexe Hüllen in 3D

Die Zeilen 17-19 werden für alle Horizontkanten *e in L* ausgeführt und erfordern *O*(card (*P*(*e*))) Zeit. Also

$$O\left(\sum_{e \in L} card(P(e))\right)$$

Aufwand in Stufe *r* und

$$\sum_{r=5}^{n} \sum_{e \in L_r} card(P(e))$$

mit *e* in L_r für alle Horizonte L_r , r = 5, ... , n. Ohne Beweis akzeptieren wir *O*(*n* log *n*) hier. (BKOS, Abschnitt 9.5 und 11.3)

QED

Theorem 6.5: Die konvexe Hülle von *n* Punkten in IR^3 kann in *O*(*n* log *n*) erwarteter Zeit berechnet werden.