kd-Trees

Verallgemeinerung von Quadtrees,
bzw. Verallgemeinerung von binären Suchbäumen
auf k Dimensionen.
(Letzteres war der Ansatz von Friedman, Bentley & Finkel
1977; das "k" im Namen war ursprünglich für die
Dim. gedacht, also 2-d-Trees; 3-d-Trees, etc.; inzwischen
aber einfach nur kd-Tree, oder k-d-Tree (ältere Schreibw.))

Konstruktions-Algo:
Geg.: $P = \{p^1, \dots, p^n\} \subseteq \mathbb{R}^d$
wähle Splitachse i (d.h., i-te Koord)
bestimme Median m von $\{p_i^1, \dots, p_i^n\}$
bilde Teilmengen

$$P^- := \{p \in P \mid p_i \leq m\}$$
$$P^+ := \{p \in P \mid p_i > m\}$$

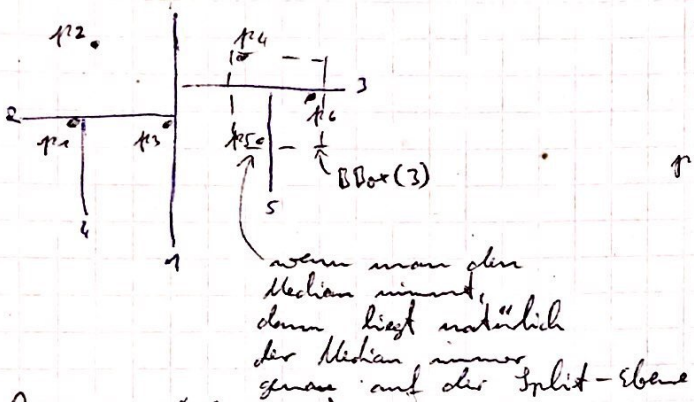
Rekursion mit P^-, P^+
Knoten v speichert
- Kinder zu P^- und P^+
- Median m und Splitachse i
- evtl. BBox (P)

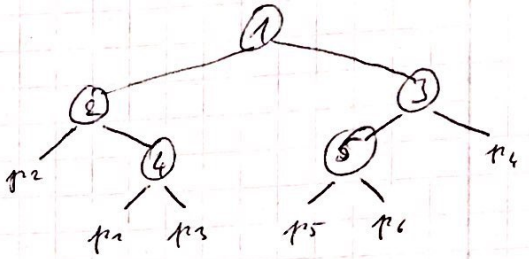
Abbruch der Rekursion bei $|P| = 1$

Terminologie:
- i heißt auch "Diskriminator" (insbes. bei Datenbanken)
- m und i definieren eine "Splitting-Ebene"

Genauso kann man kd-Trees über Objekten aufbauen, z.B. Records einer Datenbank die nicht unmittelbar Orte im Raum sind; man benötigt nur auf jeder Komponente eine totale Ordnung.

Beispiel:





Tiefe: $O(\log n)$

Bemerkung:

Ein kd-Tree induziert eine Partitionierung des Raumes.

Bezeichne mit $R(v) \subseteq \mathbb{R}^d$ die Teilregion des Knotens v.
(nicht verwechseln mit BBox(P(v)) !)

$R(root) = [-\infty, +\infty]^d$.

(In der Praxis startet man oft mit der BBox(P) an der Wurzel und aktualisiert $R(v)$ bei der Traversierung.)

Implementierung (in 2D, for sake of simplicity):

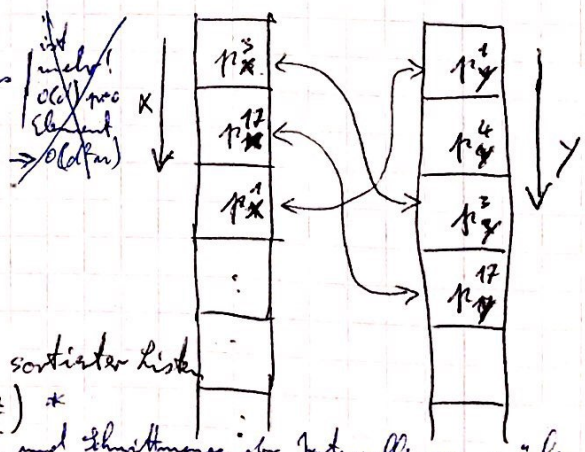
sortiere P 1x entlang x $\Rightarrow$ "x-Liste" und 1x entlang y $\Rightarrow$ "y-Liste"

verzeige die Listen

Median finden ist jetzt $O(1)$

Speicheraufwand $O(d \cdot n)$, während des Aufbaus Listen aufteilen ist $O(d \cdot n)$

(aufteilen braucht man doch gar nicht mehr!)



Laufzeit für Aufbau:

Preprocessing $\rightarrow O(d \cdot n \log n)$ ← Spalten aller sortierter Listen

Rekursion: $T(n) = O(d \cdot n) + 2\,T(\tfrac{n}{2})$ *

$\Rightarrow$ Gesamtlaufzeit $O(d \cdot n \log n)$

* ist für hinter verfolgen und Schnittmenge über Intervalle zu prüfen

Platz: $O(n)$

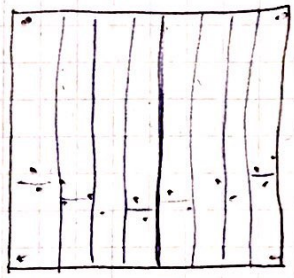
*) Der Term $O(n)$ kommt von der Medianwahl: beim ersten Mal findet man den Median, z.B. in der x-Liste, in $O(1)$; dann teilt man die Liste in 2 Hälften und macht die Rekursion; jetzt wird z.B. entlang y aufgeteilt: man muß die ganze y-Liste sequenziell durchgehen, um herauszufinden, welche nach dabei sind; so verteilt man bei jeder Rekursion einen Satz von d Indexintervallen nach unten.

Variationen:

0. Speichere den Median-Pkt in Knoten ($P = P^- \cup P^+ \cup \{\bar{p}\}$)
1. Split-Ebene:
   a) "longest spread kd tree": wähle diejenige Achse i, entlang der die Pkte die breiteste Verteilung haben, wo also BBox($P(v)$) größte Ausdehnung hat.

   Problem: falls Pkte auf einer Mannigfaltigkeit (ungefähr) liegen, deren Dim. $< d$ ist, führt diese Heuristik oft zu langen, schmalen Zellen.

   Beispiel:



   b) "longest side kd tree": splitte entlang der Koord., wo $R(v)$ am längsten ist.
   (Ist also (fast) völlig unabhängig von P)

   [Dickerson, Duncan, Goodrich: K-d trees are better when cut on the longest side; ESA 2000]

   c) Zyklisch: erst x, dann y, dann z, wieder x, etc..

2. "Binning" ("bucketing"):
   Abbruch bei $|P| \leq b$.
   (Asymptotische Laufzeiten bleiben gleich.)
   $b \approx 20..30$

   [Grenspoun, Gadia, Talbot: Acceleration of Binning Nearest Neighbor Methods]

Bemerkung:
Bei kd-Trees fällt die Anzahl Pkte assoziiert mit $R(v)$ auf dem Weg nach unten exponentiell; bei Quadtrees fällt die Größe der Knoten exponentiell (Größe = Seitenlänge).

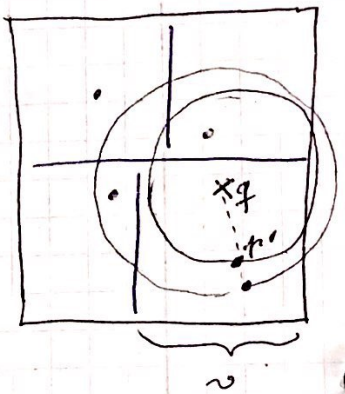
Hier passt "Stackless kd-Tree Traversal" (warped kd-trees)

Das Nearest-Neighbor-Problem ("closest point problem"):

Input
Gegeben Pktmenge $P \subseteq \mathbb{R}^d$ und $q \in \mathbb{R}^d$.

Output
Gesucht $p^* \in P$, so daß $\forall p \in P: \|p^* - q\| \leq \|p - q\|$.
p^* heißt "NN".

Idee: bestimme das Blatt v im kd-Tree mit $q \in R(v)$;
das liefert einen Kandidaten p';
nun muß man nur noch Blätter besuchen,
die die Kugel $K(q, r)$ um q mit Radius $\underbrace{\|q - p'\|}_{r}$
schneiden.



Algorithmus:
nn(v, ~~input~~ p, ~~input~~ r):
input: p = aktueller Kandidat als NN
$r = \|p - q\|$
output: new candidate pr
precond: $K(q, r)$ überlappt $R(v)$ // possibly same as inp.

[Friedman, Bentley, Finkel: An algo for finding best matches; '77]

// K = Kugel mit Radius r um M.pkt q

if v ist Blatt:
  $p' :=$ nearest neighbor aus $P(v)$ zu q
  $r' := \|q - p'\|$
  if $r' < r$ then: $(p, r) := (p', r')$
else:
  // Rekursion in das nähere Kind
  if $q_i \leq m$:
    $p, r := nn(v_l, p, r)$
  else:
    $p, r := nn(v_r, p, r)$
  // hier kann $K(q, r)$ schon kleiner geworden sein

// sei m, i die Split-Ebene in v
// seien v_l, v_r = linkes / rechtes Kind von v

// hier braucht man keinen Bounds overlap ball test! (?)

// Rekursion in das entferntere Kind

if $q_i \le m$:

if $K(q,r)$ überlappt $R(v_r)$:

$p, r := nn(v_r, p, r)$

else:

if $K(q,r)$ überlappt $R(v_l)$:

$p, r := nn(v_l, p, r)$

end if

if $K(q,r) \subseteq R(v)$:

$p^* := p$ ist die Lsg.; Rekursion abbrechen

return p, r

(**) Diese Teile könnte man doch sicher schon in die vorigen if-Zweige einbauen

(*) braucht man in einer echten Impl. nicht, da er, sobald dieser Fall eingetreten ist, sowieso nie wieder in den Bereich eines Vater-Knotens einläuft, wegen (**)

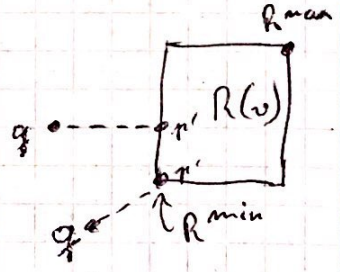
*) heißt bei Friedman et al. "ball within bounds test"; könnte man auch weglassen, der Algo geht dann halt weiter hoch, aber nie wieder runter in einen anderen Ast

(**) "bounds overlap ball test":

$K(q,r)$ K "überlappt" $R \Leftrightarrow d(q, R) < r \Leftrightarrow$

$\| q - p' \| < r$, wobei $p' \in R$ der nächste Pkt zu $q = (q_1, \dots, q_d)$ ist! (Achtung: $p' \in P(v)$ nicht!)

$$p'_j = \begin{cases} R_j^{min} & , q_j < R_j^{min} \\ q_j & , R_j^{min} \le q_j \le R_j^{max} \\ R_j^{max} & , R_j^{max} < q_j \end{cases}$$



Aufruf: $nn(root, NULL, \infty)$

Analog macht man k-NN-Suche.

Und analog geht "farthest neighbor"-Suche.

Mit kd-Trees kann man sehr einfach auch sog. "orthogonal range queries" machen, also alle Pkte liefern, die in einer achsen-parallelen Box liegen → Üb.aufgabe
Ray-Tracing mit kd-Trees: siehe CG2!

Laufzeit:
Klar ist $T(n) \in \Omega(\log n)$ und $T(n) \in O(n)$;
bessere Schranken für worst-case gibt's nicht (bis jetzt = 2000)
(Friedman et al. zeigen in ihrem Paper [Dickerson, Duncan, Goodrich]
eine expected Laufzeit von $O(\log n)$, aber nur für best knowmen über die Verteilung der Elte, und für "widest spread" kd-Trees;
und komischerweise kommt kein d in der Laufzeit vor.)

Vorab die 4D-Videos!

Exkurs: **Die Curse of Dimensionality**

(Eine Erzählung ...)

Satz (o. Bew.):

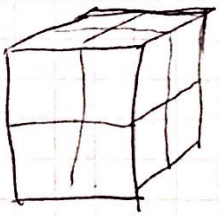
1. Ein kd-Baum über einer Menge von n Pkten im $\mathbb{R}^d$ erlaubt eine orthogonale Bereichsanfrage in Zeit $O(n^{1-\frac{1}{d}} + k)$ (↑ k = output) und Platz $O(n)$.
2. Jeder Algo zur orthog. Range Query über irgend einer Datenstruktur, die nur $O(n)$ viel Platz benötigt, hat mindestens Laufzeit $\Omega(n^{1-\frac{1}{d}} + k)$. [Rolf Klein's Buch, S. 134]

(Wenn man also nur $O(n)$ viel Platz spendiert, dann sind kd-Trees die opt. Datenstruktur für orthog. Range Queries.)

Ein einfaches Rechenbeispiel:

Geg. $N = 10^7$ uniform verteilte Pkte in einem Würfel im $\mathbb{R}^d$.

Unterteile den Würfel in $c = 2^d$ Zellen (wie bei Octree)



Anzahl Zellen $c = 2^d$

erwartete Anzahl Pkte pro Zelle $p = \frac{N}{c}$

Mindest-Anteil leere Zellen $e = \frac{c-N}{c}$ (konservative Annahme: ein Pkt pro Zelle)

d	p	e
10	$9{,}8 \cdot 10^3$	0%
30	0,009	99,1%
100	$8 \cdot 10^{-24}$	~100%

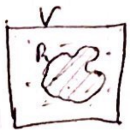
M. a. W.: je höher die Dimension, desto mehr Punkte kann man in den Würfel mit fester Seitenlänge packen, ohne daß die Dichte steigt!

Zur Erinnerung:

Sei Volumen $V \subset \mathbb{R}^d$ und Gebiet $R \subseteq V$ gegeben.

Sei $P \subseteq \mathbb{R}^d$ eine uniform in V verteilte zufällige Pktmenge.

Dann ist die erwartete Anzahl Pkte in $R = \frac{Vol(R)}{Vol(V)} \cdot |P|$.



[Mario Köppen: The curse of dimensionality; Folie 1/Seite]

Betrachte die Hyperkugel $K_d \subseteq \mathbb{R}^d$:

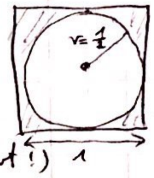
$Vol(K_d) = r^d \cdot \frac{\pi^{d/2}}{(d/2)!}$, falls d gerade

Jetzt Einheitskugel in Einheitswürfel

$$\frac{Vol(K_d)}{1^d} = \frac{(r^2\pi)^{d/2}}{(d/2)!} = \frac{0{,}78^{d/2}}{(d/2)!} \rightarrow 0 \quad \text{für } d \rightarrow \infty$$

(gilt auch für beliebige r!)

(und das sogar sehr schnell, obwohl K alle Hyperseiten berührt!)

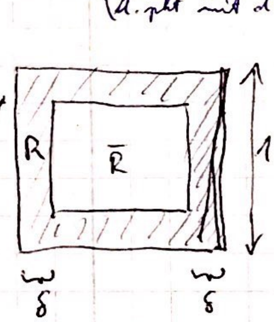


D.h., das meiste Volumen befindet sich in den Ecken!

Und die erwartete Anzahl Pkte in der Kugel $\rightarrow 0$! (Naheliegend, da Entfernung eines Pkt zu festen Pkten immer ... mit d steigt)

Betrachte den dicken Rand des Hyperwürfels (hypercube shell)

$$Vol(R) = 1 - \underbrace{(1-2\delta)^d}_{Vol(\bar{R})}$$

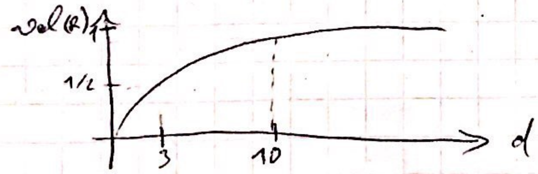


Wahrsch.keit, daß ein Pkt $p \in P$ den Abstand ≤ 0.1 von der Oberfläche des 1-Würfels hat (also in R liegt):

d	Pr (= Vol(R) · 100 %)
3	49 %
10	89 %
30	99,8 %

→ Je höher die Dimension, desto mehr Punkte liegen an der Oberfläche; m.a.W.

→ Darts im hoch-dimensionalen Raum ist noch viel schwieriger! :)



Übungsaufgabe: Wie wächst die Anzahl der "approximate NNs" mit d?

Lsg.: $V_\varepsilon \sim (1+\varepsilon)^d - 1$

Kombinatorische Zahlen für einen Hyperwürfel:

Ecken $= 2^d$, # Kanten $= d\,2^{d-1}$

Quadrate Polygone $= \frac{d(d-1)}{2}\,2^{d-2}$ (# k-dim Facetten $= \binom{d}{k}\,2^{d-k}$)

Hyperseiten $= 2d$ ← Hyperwürfel der Dim $d-1$

Länge der Diagonale $= \sqrt{d}$

(Man kann sich also einen Hyperwürfel ein wenig so vorstellen wie ein Igelbild mit einem kugelförmigen "Kern", dessen Volumen verschwindend klein ist/wird, und exponentiell vielen "Zacken" (wie am Igel), in denen fast das ganze Volumen enthalten ist.) [Köppen S. 5]

← { Never compare $Vol(K_d)$ with $Vol(K_{d+1})$!!
(think "cm" versus "cm²" !)

Zum Volumen der Hyperkugel: man sieht da manchmal eine Kurve der Art 

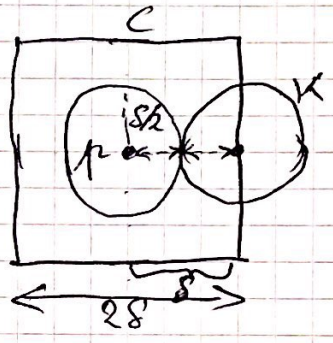
Das ist aber Quatsch, denn es ist völlig sinnlos, den Inhalt in versch. Dimensionen zu vergleichen

Message: man darf nur Vol(Kugel) / Vol(Würfel) über verschiedenen Dimensionen vergleichen

z.B. Eläle in 2D mit Vol in 3D! s. Wikipedia-Artikel "Hypersphere"

Remember the Closest Pairs algorithm:
Phase 3 involved a "sparsity" argument.
Consider pts $P \subseteq \mathbb{R}^d$ with min. dist. δ
Take any $p \in P$, place cube C centered at p with "radius" δ.
Consider $L \subseteq P$ inside C.
Question: $|L| = ?$

Consider balls K_d around each $q \in L$ with radius $\frac{\delta}{2}$.
No two balls can intersect!



Upper bound:

$$|L| \leq \text{max \# balls in } C \approx \frac{\text{vol}(C)}{\text{vol}(K)} = \frac{(2\delta)^d}{\frac{\pi^{d/2}}{(d/2)!} \cdot \left(\frac{\delta}{2}\right)^d}$$

$$= 4^d \cdot \frac{(d/2)!}{\pi^{d/2}}$$

$$\approx 4^d \frac{(d/2)^{d/2}}{\pi^{d/2}} = 16^{d/2} \cdot \frac{(d/2)^{d/2}}{\pi^{d/2}} \approx (2.54\, d)^{d/2}$$

(Stirling's approx.)

Good news: in case d is fixed $\Rightarrow |L| \in O(1)$
Bad news: $|L| \to \infty$ as $d \to \infty$

Approximate Nearest Neighbors

Wegen der Curse of Dimensionality und der unteren Schranke von $\Omega(n^{1-1/d})$ für orthog. Range Queries glauben viele, daß es keinen Sinn macht, Algos für NN in hohen Dimensionen zu suchen, die signifikant besser als $O(n)$ sind (und nur Platz $O(n)$ brauchen).

Was kann man tun? Mehr Platz spendieren: etwas unsaubere. Nicht exakten NN suchen (reicht für die allermeisten Appl.)

[Dickerson, Duncan, Goodrich: kd-trees are better when cut on the longest side, ESA 2000.
Duncan, Goodrich, Kobourov: Balanced aspect ratio trees...
Arya, Mount, Netanyahu, Silverman, Wu: An optimal algo for approx nearest neighbor searching -; 1998, S. 18+19]

Def.:
Sei $S \subseteq \mathbb{R}^d$ eine Menge von Pkten, $q \in \mathbb{R}^d$ ein Query-Pkt, $p^* \in S$ der NN, und $\varepsilon > 0$.
Dann heißt ein $p' \in S$ "$(1+\varepsilon)$-approximate nearest neighbor" $\Leftrightarrow$

$$d(p', q) \leq (1+\varepsilon) \cdot d(p^*, q) .$$

Bezeichnungen:
ab jetzt: v und $R(v)$ sind "Knoten" und "Region des Knoten" sind Synonyme.
Annahme: metric space, d.h. $d()$ ist Metrik, z.B. $d(p, q) := \|p - q\|$, $d(v, q) := d(R(v), q)$

$p_v :=$ der in v liegende Pkt, falls v Blatt ist
(sonst hatten wir ja $P(v) =$ Menge der in v liegenden Pkte)

Also:

Q = Priority-Queue mit Zeiger auf Knoten im k-d-Tree, sortiert nach $d(v,q)$ = Abstand zw. q und v; (der am kleinsten Abstand vorne)

p° := Pkt unendl. weit weg (= aktueller Kandidat des NN)
(Achtung: es ist sehr wichtig, daß man hier mit einem "unendlichen" Pkt startet! sonst könnte es passieren, daß die while-Schleife evtl gar nicht startet!)

v := Root des k-d-Trees

while $d(v,q) < \frac{1}{1+\varepsilon} d(p^\circ, q)$:

  while $v \neq$ Blatt : (v is inner node)

    seien v_1, v_2 die Kinder von v, und $d(v_1,q) \leq d(v_2,q)$

    füge v_2 in Q ein

  endwhile $v := v_1$

  if $d(p^\circ v, q) < d(p^\circ, q)$: { v ist jetzt Blatt }

  then

    $p^\circ := p^\circ v$

  $v \leftarrow$ extractMin(Q)
  $v :=$ minimales Element aus Q, lösche dieses aus Q;

endwhile

return p°

(Das erstaunliche ist, daß der Algo das p^* gar nicht kennen muß! Ist für $\varepsilon=0$ genau der normale Algo, nur eben in iterativer Form)

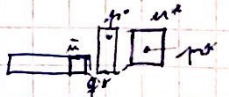
Zeit: $O(l \cdot \log n)$, l = # besuchter Blätter

Bew.:

1. Korrektheit: Sei u^* Blatt aus k-d-Tree, das p^* enthält

a) u^* wurde besucht $\Rightarrow$ Algo liefert $p^* \Leftrightarrow p^\circ$

b) u^* wurde nicht besucht $\Rightarrow p^\circ \neq p^* \Rightarrow$



$$d(p^*,q) \geq d(u^*,q) \geq d(\bar{u},q) \geq \frac{1}{1+\varepsilon} d(p^\circ,q)$$

(wobei $d()$ Metrik; nähere Knoten werden zuerst besucht; Abbruchkrit.)

wobei $\bar{u}$ der zuletzt besuchte Knoten sei. (Blatt oder innerer Knoten)

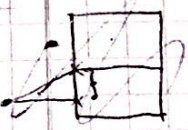
2. Zeit:

pro äußerer Schleifendurchlauf: 1x Knoten extrahieren, $O(\log n)$ mal Knoten einfügen; (innerer Loop)

Fibonacci-Heap verwenden $\Rightarrow O(\log n)$ für Extrakt., $O(1)$ amortisierte Kosten für Einfügen; $d(v,q)$ braucht $O(1)$ $\Rightarrow$ Beh.

Bem.:

1. In der Praxis bleibt Heap relativ klein → verwende normalen Heap.
2. ~~$d(v, q)$ braucht nur $O(1)$, nicht $O(d)$, weil man Distanz inkrementell aus Dist. zum Vater berechnen kann.~~
3. Analog geht "$(1-\varepsilon)$-farthest neighbor".

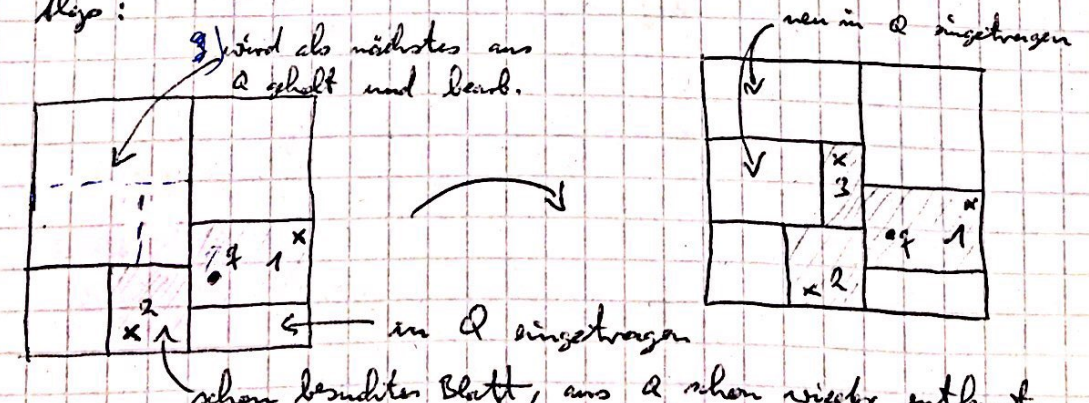


4. Gibt PD-Software "ANN library", die gibt inzwischen bessere. S. Folien verwendet aber eine wesentlich komplizierte Datenstruktur, weil damals noch nicht bekannt war, daß auch die alten "longest-side kd-Trees" diese schöne worst-case-Laufzeit haben.
5. # besuchter Blätter $\in O(\log^{d-1} n)$ [ESA 2000]; zeigt man, indem man Schranke für Anzahl der Knoten des kd-Tree findet, die einen Kreis "durchstechen". Genauer: $O\left(\left(\frac{\log n}{\varepsilon}\right)^{d-1}\right)$ gilt nur für "longest-side" kd-Tree!

Satz:

Zu einer Menge $S \subseteq \mathbb{R}^d$, $|S| = n$, und einem Query-Pkt $q \in \mathbb{R}^d$ findet man den $(1+\varepsilon)$-nearest neighbor in Zeit $O\left(\frac{\log^d n}{\varepsilon^{d-1}}\right)$.

Bsp zum Algo:



Beste ANN-Algos

Achtung: "beste" hängt von vielen Faktoren ab!
- Metrik oder nicht-metrischer Abstand
- Verteilung der Pkte (hohe Korrelation oder uniform)
- Dimension

Folgende zwei Algos funktionieren in praxi sehr gut für viele Daten

Alternatives Abbruchkriterium: besuche max L_{max} viele Blätter.

Alternatives Güte-Maß des ANN-Algo:

$$\text{precision} = \frac{\#\text{exakter NN's}}{\#\text{Queries}}$$

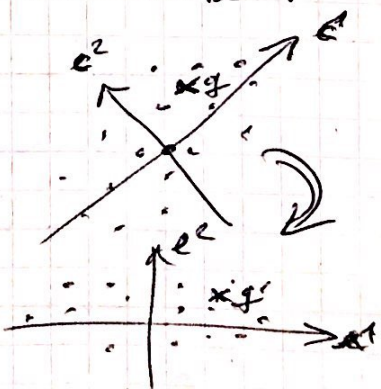
error = 1 − precision (eigtl. "error")

Randomized kd-Tree (RKD):
- Bestimme D Achsen (dimensions) mit höchster Varianz der Pkte
- Wähle eine davon zufällig
- Splitte entlang Median dieser Achse

Experimente zeigen: D=5 reicht

[Muja & Lowe: Fast approximate Nearest Neighbors with automatic algo configuration; 2009.
Silpa-Anan & Hartley: Optimised KD-trees for fast image descriptor matching; 2008.]

PCA-RKD:
- Bestimme Principal Components der Pkte P
- Transformiere Pkte → P'
- Baue RKD über P'
- Transformiere Query-Pkt q → q'
- Weiter mit bisherigem ANN-Algo.



RKD-Forest (auch mit PCA):

Baue mehrere (20-50) RKD-Trees über P (bzw P')

(Erinnert ein wenig an Random Forests!)

ANN-Search mittels RKD-Forest:

Maintain 1 p-queue only (one for all)

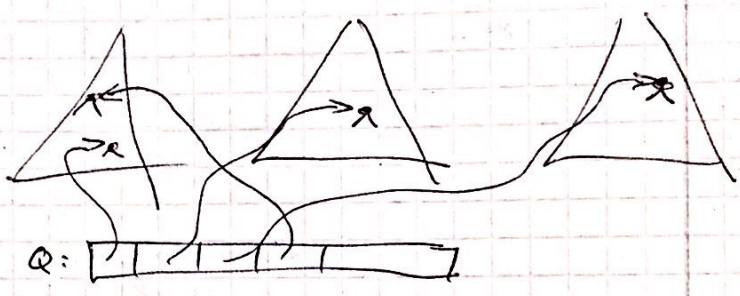
Descend each RKD-tree down to leaf closest to q (first iteration of outer while loop of ANN algo)

Choose closest p_i of all those leaves

put others in p-queue

Proceed with ANN-algo as before

Queue contains pointers to nodes in different trees!



Warum funktioniert das besser?

Bsp.: 1 Mio Pkte $\approx 2^{20}$ ⇒ Tiefe der kd-Trees = 20

Dimension = 100 ⇒ ANN schaut 80 Einträge von q gar nicht an!

⇒ q und p_i in Blatt sind nah zueinander in 20 Dim., aber in den 80 anderen nicht notwendigerweise

Hier hilft die Randomisierung

andere Betrachtung:

- Beim klassischen ANN-Algo hängen die nacheinander (aus der P-Queue kommenden) besuchten Zellen stark voneinander ab
- Bei obigem Algo liegen die Zellen aus der P-Queue in verschiedenen, randomisierten kd-Trees ⇒ weniger Abhängigkeit dazwischen!

k-Means-Tree:

1. Partitioniere P mittels k-means-Clustering in k Cluster
2. Erzeuge für jeden Cluster einen k-Means-Tree
3. Erzeuge Knoten mit k Kindern

ANN-Search mittels k-Means-Tree:

Traversiere Baum bis zum nächsten Blatt
Schreibe nicht-besuchte Kinder in P-Queue
Sortiere P-Queue bzgl. dist(q, cluster-center)
Stoppe nach Lmax besuchten Blättern

Bem.:

- k-means-Clustering benötigt auch NN (allerdings immer nur zu den k Mittelpunkten der Cluster)
- Randomisierung und Forest haben hier nichts gebracht (lt. Autoren)
- Aufbau ist rel. teuer ⇒ führe nur wenige Iterationen in k-means-Clustering durch, nicht bis Konvergenz laufen lassen. ANN wird kaum beeinträchtigt