2.2.3 Indexbasierte *k*-Nächste-Nachbarn Anfrage

– Einfache Tiefensuche

- Unterschied zur Range-Query
  - Nächste Nachbar kann beliebig weit vom Anfragepunkt weg liegen
  - Gestalt der Query zunächst unbekannt
  - Es kann zunächst nicht anhand der Seitenregion entschieden werden, ob eine Seite gebraucht wird
  - Ob eine Seite gebraucht wird, hängt auch von dem Inhalt der anderen Seiten ab
  - Kennt man NN-Distanz, würde Range Query ausreichen
  - Kennt man ein beliebiges Objekt, kann man dessen Abstand als obere Schranke für die NN-Distanz nutzen
  - Kennt man mehrere Objekte, kann man den geringsten Abstand als obere Schranke für die NN-Distanz nutzen

- Umformulierung des RQ-Algorithmus:
  - Verwende als ε die kleinste Distanz zu den bisher gefunden Nachbarn

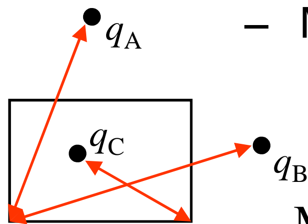
```
Globale Variable: stopdist = +∞;

NN-Index-Simple-TS(pa, q)          // pa = Diskadress z.B. der Wurzel des Indexes
    result = ∅;
    p := pa.loadPage();
    IF p.isDataPage() THEN
        FOR i=0 TO p.size() DO
            IF dist(q, p.getObject(i)) ≤ stopdist THEN
                result := getObject(i);
                stopdist = dist(q, p.getObject(i));
    ELSE                     // p ist Directoryseite
        FOR i=0 TO p.size() DO
            IF MINDIST(q, p.getRegion(i)) ≤ stopdist THEN
                result := NN-Index-Simple-TS(p.childPage(i), q)
    RETURN result;
```

- Nachteil des einfachen Tiefensuch-Algorithmus
  - Initialisierung: stopdist = $+\infty$
  - Dadurch: Start mit beliebigem Pfad
  - Folge: die ersten gefundenen Objekte sind meist sehr weit vom Anfrageobjekt entfernt => stopdist ist wenig selektiv
  - Verbesserung: beginne Pfad, der möglichst nah zum Anfrageobjekt liegt

– Tiefensuche (Depth-first search nach [RKV 95])

[Roussopoulos, Kelley, Vincent. Proc. ACM Int. Conf. Management of Data (SIGMOD), 1995]

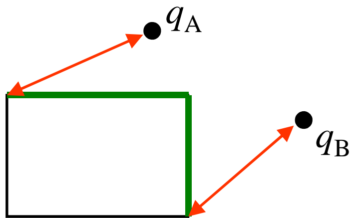
- Vermeidet langsame Einschränkung des Suchraums durch
  - Verwendung der Seitenregionen zur Abschätzung der k-NN-Distanz
  - Priorisierung der Tiefensuche nach Distanz der Seitenregion zur Query
- Neben MINDIST weitere Abschätzungen der NN-Distanz durch:
  - MAXDIST
    - » Maximale Distanz zwischen Query und allen Punkten der Seitenregion
    - » NN-Distanz kann nicht schlechter als MAXDIST werden



$$\text{MAXDIST}(region, q) = \sqrt{\sum_{0<i\le d} \max\{(q_i - region.UB_i)^2, (q_i - region.LB_i)^2\}}$$

  - MAXDIST aller bisher bekannten Seitenregionen wird zum frühzeitigen Abschneiden von Teilbäumen (Verbesserung der Pruning-Distanz) verwendet.

- MINMAXDIST
  - » MBRs als Seitenregionen: maximale NN-Distanz noch besser abzuschätzen
  - » Auf jeder Kante des MBR muss ein Punkt liegen (sonst ist MBR nicht minimal)
  - » Intuition: „nächstliegende Kante, weitester Punkt“



$$MINMAXDIST(region, q) = \sqrt{\min_{0<k\le d}\left(|q_i - rm_i|^2 + \sum_{\substack{i\ne k \\ 0<i\le d}} |q_i - rM_i|^2\right)}$$

wobei

$$rm_i = \begin{cases} region.LB_i & \text{if } q_i \le \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

$$rM_i = \begin{cases} region.LB_i & \text{if } q_i \ge \frac{region.LB_i + region.UB_i}{2} \\ regionUB_i & \text{else} \end{cases}$$

- Für andere Geometrien (nicht MBRs) sind MINDIST und MAXDIST analog definierbar; MINMAXDIST allerdings nicht
- Abschätzung von stopdist durch Minimum aus stopdist und MINMAXDIST (bzw. MAXDIST) aller bisher bekannten Seitenregionen (pruningdist)
- Vor dem rekursiven Abstieg: sortieren der Kindseiten nach MINDIST (experimentell als bestes Prioritätsmaß ermittelt)

- Algorithmus (NN-Variante, d.h. für k = 1):

```
Globale Variablen: pruningdist = +∞;
result = ∅; //

NN-Index-RKV(pa, q)              // pa = Diskadress z.B. der Wurzel des Indexes
   p := pa.loadPage();
   IF p.isDataPage() THEN
      FOR i=0 TO p.size() DO
         IF dist(q, p.getObject(i)) ≤ pruningdist THEN
            result := p.getObject(i);
            pruningdist = dist(q, result);
   ELSE                          // p ist Directoryseite
      FOR i=0 TO p.size() DO
            IF MINMAXDIST(q, p.getRegion(i)) < pruningdist THEN
            pruningdist = MINMAXDIST(q, p.getRegion(i));
      quicksort(p.getObjectArray(), MINDIST);
      FOR i=0 TO p.size() DO
         IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
            NN-Index-RKV(p.childPage(i), q);
   END IF;
```

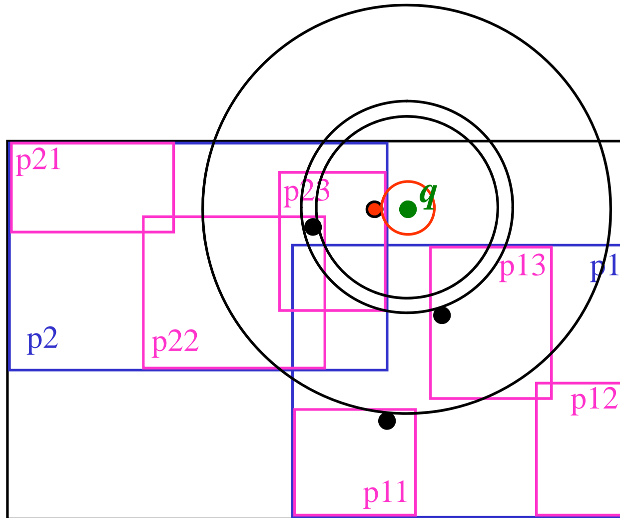
- Algorithmus (kNN-Variante, d.h. für k ≥ 1):

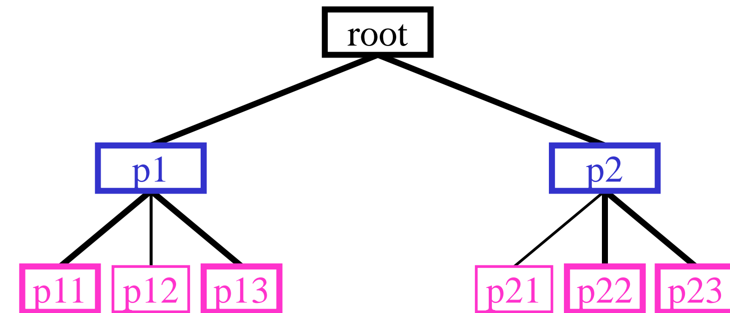
Globale Variablen: pruningdist = +∞;

result = ∅; // Heap mit (oid,dist()) tupel, erster Eintrag hat höchsten dist()-wert, maximale Heapgröße = k (Bei Überlauf wird letztes Element gelöscht)

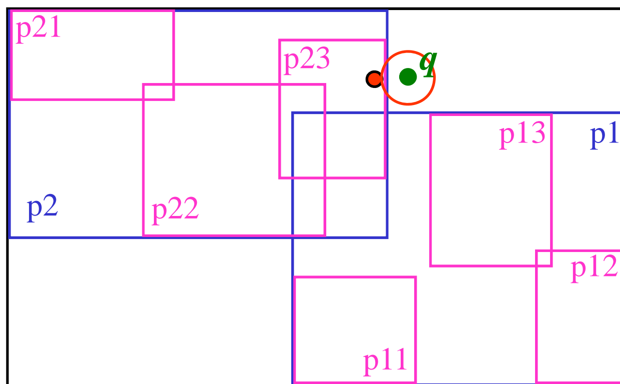
```
k-NN-Index-RKV (pa, q, k)          // pa = Diskadress z.B. der Wurzel des Indexes
  p := pa.loadPage();
  IF p.isDataPage() THEN
    FOR i=0 TO p.size() DO
      IF dist(q, p.getObject(i)) ≤ pruningdist THEN
        result.insert((p.getObject(i));
        pruningdist = dist(q, result.first);
  ELSE                             // p ist Directoryseite
    FOR i=0 TO p.size() DO   // Vorausgesetzt: k << # Objekte in
                             // den Teilbäumen unter p
      IF MAXDIST(q, p.getRegion(i)) < pruningdist THEN
        pruningdist = MAXDIST(q, p.getRegion(i));
    quicksort(p.getObjectArray(), MINDIST);
    FOR i=0 TO p.size() DO
      IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
        NN-Index-RKV(p.childPage(i), q);
  END IF;
```

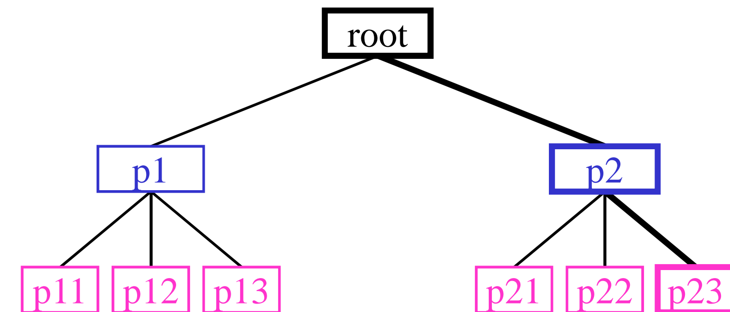
- Ablaufbeispiel
  - Einfache Tiefensuche



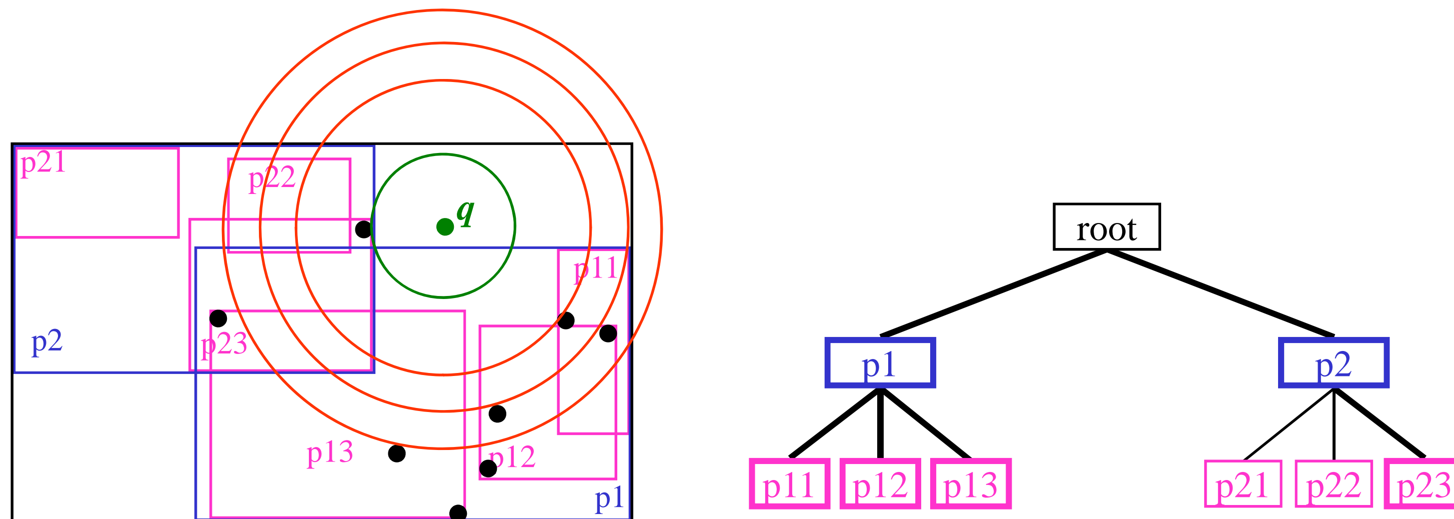


  - Tiefensuche nach RKV





- Fazit:
  - Priorisierung mit MINDIST bewirkt Reduktion der Seitenzugriffe von 7 auf 3
  - MAXDIST (bzw. MINMAXDIST) kann grundsätzlich die Pruning-Distanz verbessern, verhindert hier aber keine Seitenzugriffe
  - Trotz Priorisierung: Tiefendurchlauf kann prinzipiell stark fehlgeleitet werden wenn z.B. eine Seite auf dem ersten Level sehr nah am Queryobjekt liegt, ihre Kindseiten aber relativ weit weg



  - Bei Start mit p_2 hätte keine der Kindseiten von p_1 geladen werden müssen

– Priorität-basierte Suche (Best-first search nach [HS 95]

[Hjaltason, Samet. Proc. Int. Symp. on Large Spatial Databases (SSD), 1995]

- Statt rekursivem Durchlauf: Liste der aktiven Seiten (active page list APL)
  - Seite p ist aktiv genau dann wenn folgende Bedingungen erfüllt sind:
    - » p wurde noch nicht geladen
    - » Elternseite von p wurde bereits geladen
    - » MINDIST(*q*, p.getRegion()) ≤ pruningdist
  - APL wird mit Wurzel des Indexes initialisiert
  - Seiten in APL nach MINDIST zum Anfrageobjekt aufsteigend sortiert
  - Algorithmus entnimmt immer die erste Seite aus APL (mit kleinster MINDIST)
  - Entnommene Seite wird geladen und verarbeitet: („verfeinert“)
    - » Datenseiten werden wie bisher verarbeitet
    - » Directoryseiten: Kindseiten mit MINDIST ≤ pruningdist in APL einfügen
  - Ändert sich pruningdist werden Seiten mit MINDIST > pruningdist alternativ:
    - » aus APL entfernt
    - » als gelöscht markiert
    - » ohne explizite Markierung später ignoriert

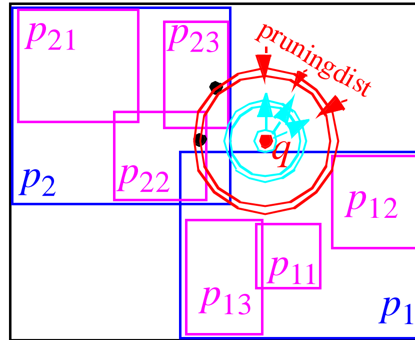
- Algorithmus:

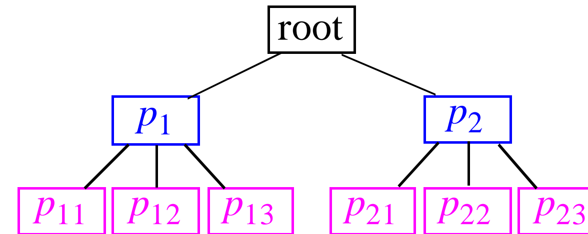
Globale Variablen: pruningdist = +∞;

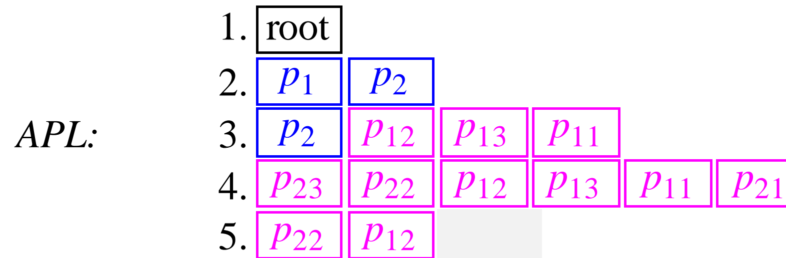
result = ∅; // Heap mit (o,dist(q,o)) tupel, erster Eintrag hat höchsten dist()-wert, maximale Heapgröße = k (Bei Überlauf wird letztes Element gelöscht).

```
k-NN-Index-HS(pa, q)            // pa = Diskadress z.B. der Wurzel des Indexes
    apl = LIST OF (dist:Real, da:DiskAdress) ORDERED BY dist ASCENDING
    apl = [(0.0, pa)]
    WHILE NOT apl.isEmpty() AND apl.first().dist ≤ pruningdist DO
        p := apl.pop_first_element;
        IF p.isDataPage() THEN …

                * siehe k-NN-Index-RKV  *

        ELSE                    // p ist Directoryseite
            FOR i=0 TO p.size() DO
                IF MINDIST(q, p.getRegion(i)) ≤ pruningdist THEN
                    apl.insert(MINDIST(q, p.getRegion(i)), p.childPage(i));
        END IF;
```

- Beispiel







- Eigenschaften
  - Allgemein
    - » Seiten werden nach aufsteigendem Abstand geordnet zugegriffen (blaue Kreise)
    - » pruningdist wird kleiner, sobald nähergelegenes Objekt gefunden (rote Kreise)
    - » Anfragebearbeitung stoppt, wenn beide Kreise sich treffen. Optimalität!!!
  - Speicherbedarf
    - » Wie bei Breitensuche kann gesamter unterste Directorylevel in APL stehen
    - » Dieser Fall is allerdings unwahrscheinlicher als bei Breitensuche
    - » Speicherkomplexität O(*n*) (Tiefensuche O(log *n*))

- Optimalität des Verfahrens

  [Berchtold, Böhm, Keim, Kriegel. ACM Smp. Principles of database Systems (PODS), 1997]

  – Prioritätssuche nach [HS 95] ist optimal bzgl. der Anzahl der Seitenzugriffe
  – Beweis (Überblick):
    » Lemma 1: jeder korrekte Algorithmus muss mind. die Seiten laden, die von der NN-Kugel um *q* berührt werden
    » Lemma 2: das Verfahren greift auf Seiten in aufsteigendem Abstand von *q* zu
    » Lemma 3: keine Seite *s* wird zugegriffen, mit MINDIST(*q*, *s*) > NN-Distanz(*q*)

  – **Lemma 1:** Ein korrekter NN-Algorithmus muss mind. die Seiten *s* laden, die MINDIST(*q*, *s*) ≤ NN-Distanz(*q*) erfüllen.

    Beweis: Angenommen eine Seite *s* mit MINDIST(*q*, *s*) ≤ NN-Distanz(*q*) wird nicht geladen. Dann kann diese Seite Punkte enthalten (als Datenseite; Directoryseiten können im entspr. Teilbaum Punkte speichern), die näher am Anfragepunkt liegen als der nächste Nachbar. Der nächste Nachbar ist also nicht als solcher validiert, da über Punkte in einem Teilbaum keine Infos bekannt sind, außer dass sie in der entsprechenden Region liegen.

- **Lemma 2:** Das Verfahren greift auf die Seiten des Index aufsteigend sortiert nach MINDIST zu.

  Beweis: Die Seiten werden in aufsteigender Reihenfolge aus der APL entnommen. Es muss also nur sichergestellt werden, dass nach Entnahme von Seite *s* keine Seiten *s*‘ mehr in APL eingefügt werden, mit MINDIST(*q*,*s*‘) < *r* := MINDIST(*q*,*s*). Alle Seiten, die nach Entnahme von s in APL eingefügt werden, sind entweder Kindseiten von *s* oder Kindseiten von Seiten *s*“ mit MINDIST(*q*,*s*“) ≥ *r*. Da die Region einer Kindseite in der Region der Elternseite vollständig eingeschlossen ist, ist die MINDIST einer Kindseite nie kleiner als die der Elternseite. Daher haben alle später eingefügten Seiten eine MINDIST ≥ *r*.

- **Lemma 3:** Das Verfahren greift auf keine Seite *s* zu, mit MINDIST(*q*,*s*) > NN-Distanz(*q*).

  Beweis: Nach Lemma 2 können nach Zugriff auf Seite *s* nur Punkte *p* gefunden werden, mit dist(*q*, *p)* > MINDIST(*q*, *s*). Wäre vor Zugriff auf *s* ein Punkt *p* mit dist(*q*, *p)* < MINDIST(*q*, *s*) gefunden worden, dann wäre s aus der APL gelöscht worden bzw. der Algorithmus hätte vor der Bearbeitung von *p* angehalten.

  □

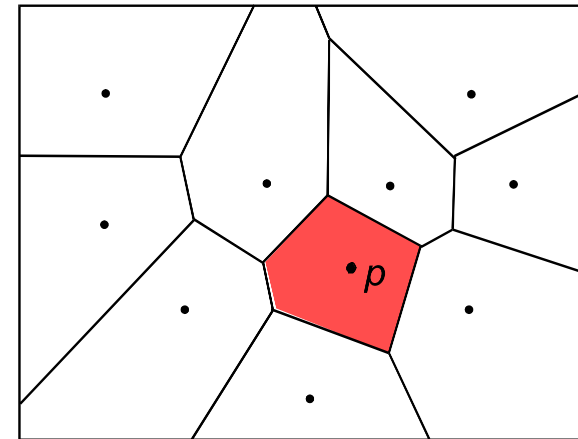
- Aus Lemma 1-3 ergibt sich, dass der Algorithmus nach [HS 95] optimal bzgl. der Anzahl der Seitenzugriffe ist.

– Hybrider Algorithmus: Voronoi-Diagramme

[Berchtold, Ertl, Keim, Kriegel, Seidl. Proc. Int. Conf. Data Engineeering (ICDE), 1998]

- Nur für Vektordaten!!!
- Idee:
  - Berechne für jeden Punkt *p* den Teil des Datenraumes in dem *p* der nächste Nachbar ist (Voronoi-Zellen)
  - Speichere Voronoi-Zellen in DB
  - NN-Anfrage entspricht Punktanfrage mit *q* auf den Voronoi-Zellen

    => Punkt *p*, in dessen Voronoi-Zelle *q* liegt, ist der NN von *q*



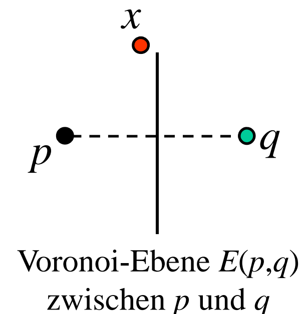
- Problem:
  - Voronoi-Zellen sind konvexe Polygone
  - Ab $d > 2$ sehr komplex (große Anzahl Eckpunkte)
- Lösung: Approximation der Voronoi-Zellen (z.B. mit MBR)
  - Nur Filterschritt, da MBRs sich überlappen können, und q in mehrere dieser MBRs liegen kann

    => Verfeinerung der Kandidaten mit exakten Punktdistanzen

2.2.4 Indexbasierte Reverse k-Nächste-Nachbarn Suche

[Tao, Papadias, Lian. Int. Conf. Very Large Databases (VLDB), 2004]

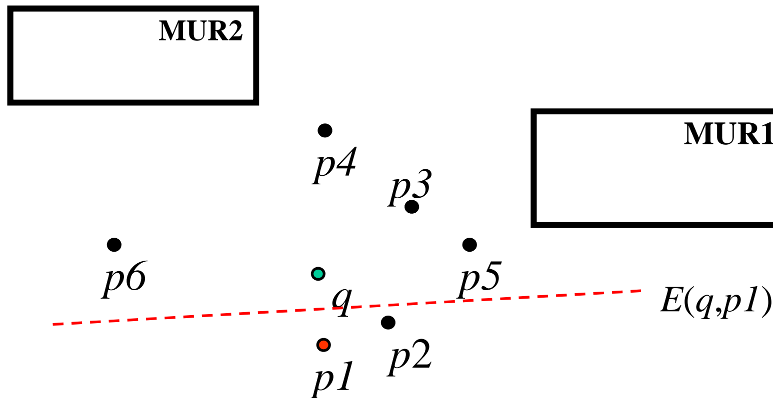
– Idee:
- Geometrisches Pruning:
  – Gegeben: Voronoi-Ebene zwischen q und beliebigen Punkt p.
  – Liegt ein Punkt x auf der Seite von dieser Voronoi-Ebene, kann q nicht NN von x sein und damit $x \notin \text{RNN}(q)$.



Voronoi-Ebene $E(p,q)$ zwischen p und q

- Iterative Berechnung der RkNN-Ergebnisse (Filter-Verfeinerung)
  – Berechne ein NN-Ranking der DB
  – Solange noch Objekte im Ranking sind:
    » Rufe getNext() auf
    » Wenn aktueller Punkt p nicht „hinter" einer Voronoi-Ebene liegt, dann konstruiere neue Voronoi-Ebene $E(p,q)$; p wird zur Kandidatenmenge hinzugefügt
    » sonst (p liegt hinter einer Voronoi-Ebene): p kein Kandidat (muß aber für Verfeinerungsschritt (s.u.) berücksichtigt werden).
  – Punkte, die die Ebenen bestimmen, müssen verfeinert werden, d.h. für diese Punkte muss jeweils eine kNN-Anfrage berechnet werden

- Beispiel:



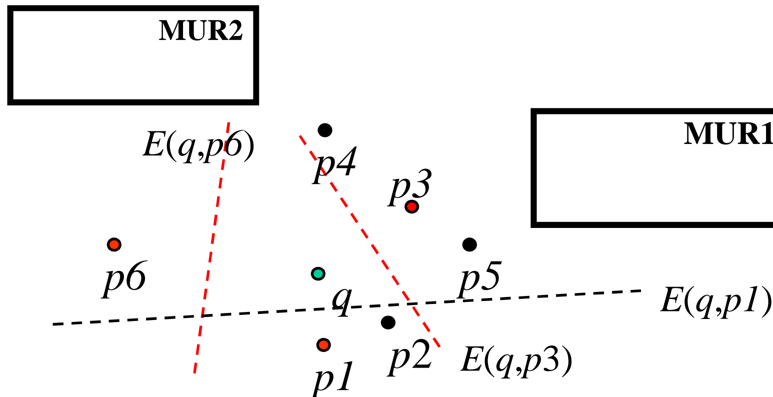
Bisherige Kandidaten:

{p1}

Inhalt des Rankings

*p*2: → ausschließen (pruning)
*p*3: → evaluieren
*p*4: → evaluieren
*p*5: → evaluieren
*p*6: → evaluieren
MUR1: verfeinern
MUR2: verfeinern

- Evaluierung von *p3*
  - » Bilde Ebene *E*(*q*,*p*3) → *p4, p5* und MUR1 nicht mehr evaluieren/verfeinern (diese liegen nun hinter *E*(*q*,*p*3))
  - » MUR2 muss weiterhin verfeinert werden



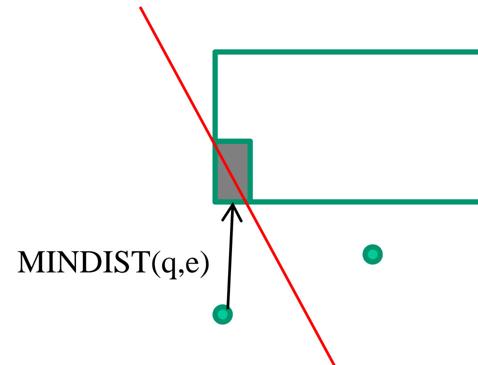
Bisherige Kandidaten:

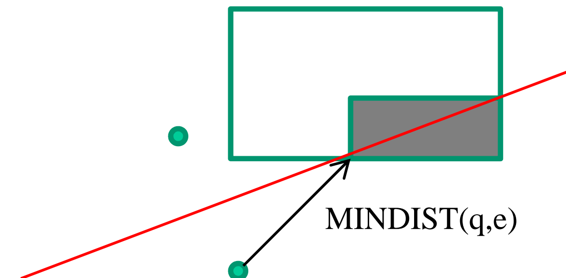
{p1, p3, p6}

Inhalt des Rankings (ungeordnet):

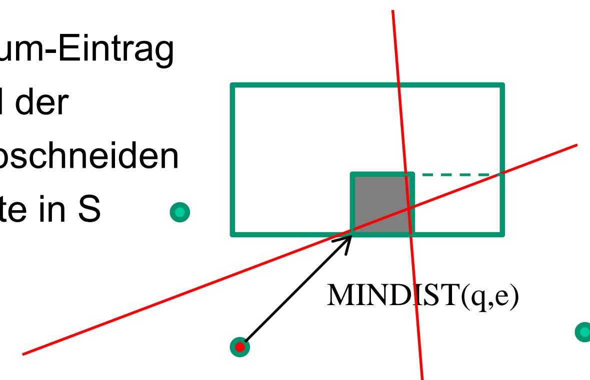
MUR2: verfeinern

- Trimmen
  - Partielles abschneiden (trimmen) von Seitenregionen (Rechtecken) bzgl. einer Pruningebene





  - Anpassung der MINDIST(q,e) nach dem Trimmen
    - ➔ führt eventuell zur Erhöhung der MINDIST(q,e)
    - ➔ erhöht die Chance, dass Seitenregion *e* früher ausgefiltert (geprunt) werden kann.
  - Funktion ***trim****(q,S,e)*:
    - » q:=Anfrageobjekt, S:=Objektmenge, e:=R-Baum-Eintrag
    - » Berechnet **minimale Distanz** zwischen q und der Region N^{resM}, die durch das kollektive partielle Abschneiden (trimmen) der Seitenregion von e durch die Punkte in S gebildet wird.
    - » Wenn $N^{resM} = \emptyset$, dann $trim(q,S,e) = \infty$



– Algorithmus (Filter):

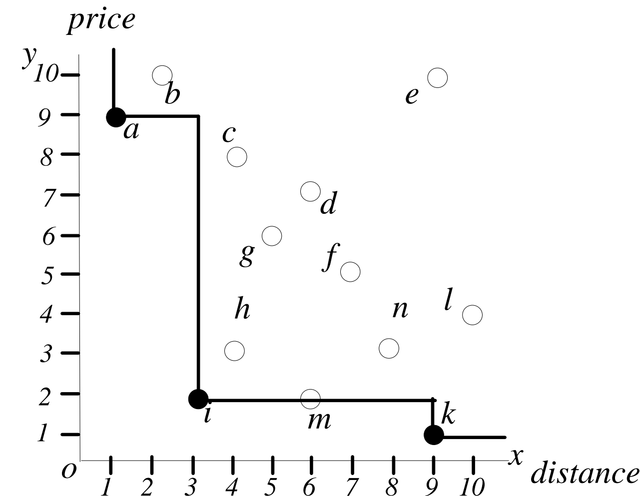
```
TPL-filter(q) // Objekt q ist Anfrageobjekt
  apl = LIST OF (dist:Real, e:entry) ORDERED BY dist ASCENDING
  apl = [(0.0, root)]
  S_cnd = ∅; S_rfn = ∅;
  WHILE NOT apl.isEmpty() DO
    (d,e) := apl.pop_first_element;
    IF trim(q,S_cnd,e.getRegion)=∞ THEN
       S_rfn=S_rfn ∪ {e};
    ELSE // e ist (oder enthält) evtl. ein Kandidat
      IF e is a point object THEN
         S_cnd=S_cnd ∪ {e};
      ELSE // e ist ein Directory-Eintrag
         FOR i=0 TO e.size() DO
            IF trim(q,S_cnd,e.getRegion(i) = ∞ DO
               S_rfn = S_rfn ∪ {e};
            ELSE
               apl.insert(trim(q,Scnd,e.getRegion(i)),e.getChild(i));
            END IF;
      END IF;
    END IF;
```

Quelle: [Tao, Papadias, Lian. Int. Conf. Very Large Databases (VLDB), 2004]

2.2.5 Indexbasierte Skyline-Anfrage

– Anforderungen:

- Gegeben:
  – Menge von d-dimensionalen Punkte (Objekte)
  – Indexierung mittels R-Baum



- Gesucht:
  – Alle Objekte, die von keinem anderen Objekt dominiert werden.
- Ziele:
  – Wenig Seitenzugriffe
  – Wenig Dominanzüberprüfungen (Objektvergleiche)
  – Möglichst früh erste Ergebnisse ausgeben

- Grundsätzlich viele unterschiedliche Ansätze
  – Hauptspeicher-basiert ←→Sekundärspeicher-basiert
  – Iterative Berechnung ←→ Nicht-Iterative Berechnung

  Skyline-Anfrage Varianten:
  – Mit explizitem Anfrageobjekt(en) (dynamische Skyline)
  – zusätzliche Bedingungen
  – andere Skylinevarianten: z.B: Top-k-Dominanz, etc.

- Bekannteste Ansätze die auf Sekundärspeicher beruhen:

  (Zusammenfassung aus [Papadias et al., ToDS 2005])
  – Divide-and-Conquer, Block-Nested Loop [Borzsonyi et al., 2001]
  – Sort First Skyline [Chomicki et al., 2003]
  – Bitmap, Index [Tan et al., 2001]
  – Nearest-Neighbor [Kossmann et al., 2002][Papadias et al., ToDS 2005]

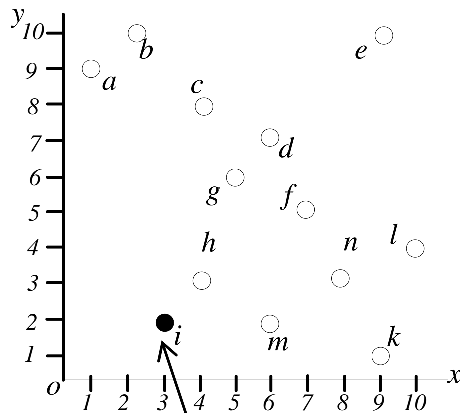
    Eigenschaften:
    » Sekundärspeicherbasiert
    » Erfüllen alle drei Ziele:
      - wenig Seitenzugriffe und Dominanzüberprüfung mittels Index (R-Baum).
      - Erste Ergebnisse werden frühzeitig ausgegeben durch iterative Verarbeitung.

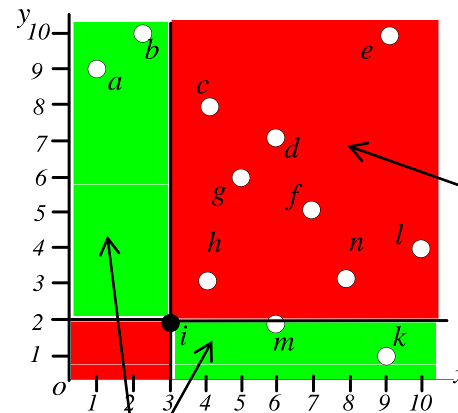
– Nächste-Nachbarn-Skyline (NNS) Algorithmus:

[Kossman et al., VLDB 2002]

Prinzip:

- Benutzt Nächste-Nachbarn-Suche zur (rekursiven) Partitionierung des Suchraums





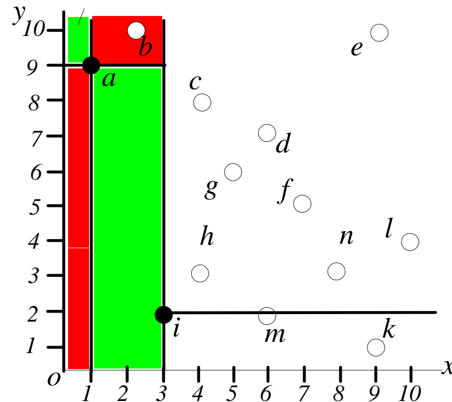
Raumpartition mit Objekten die von Objekt i dominiert werden => Objekte gehören nicht zum Ergebnis (*true drops*).

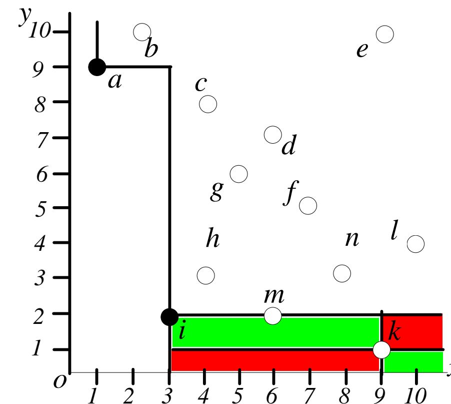
Nächster Nachbar (des Koordinaten-Ursprungs) → erstes Skyline-Ergebnis

Raumpartitionen mit weiteren Skyline-Kandidaten

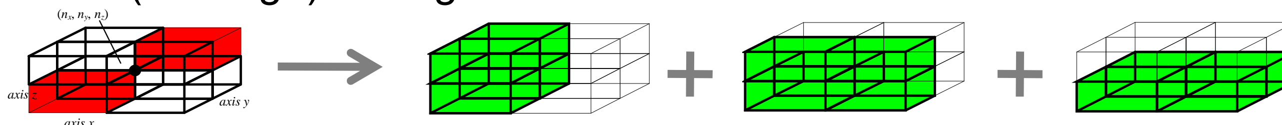
- Nächste-Nachbar-Suche kann durch R-Baum Index beschleunigt werden (z.B. Alg.: k-NN-Index-HS, siehe Folie 63).

- Nächste-Nachbar-Suche wird zur weiteren Partitionierung **in jeder** Kandidaten-Suchregion rekursiv fortgesetzt.





- Vorteile:
  - Verwendung von effizienten Methoden zur NN-Suche.
  - Erste (relevante) Resultate können schnell ausgegeben werden.
- Nachteile:
  - Im d-dimensionalen Raum führt jedes gefundene Skyline-Objekt (Punkt) zu d weiteren Fensteranfragen.
    → viele redundante Anfragen → Duplikateliminierung
    → viele (unnötige) Anfragen auf leeren Raum



– Branch-and-Bound Skyline (BBS) Algorithm:

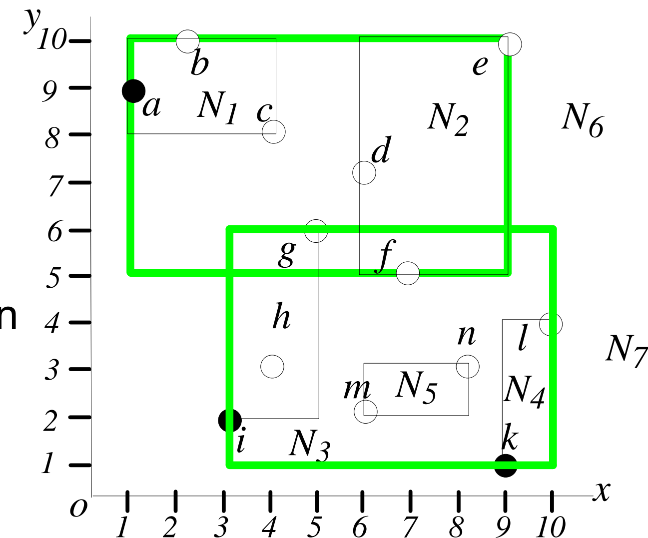
[Papadias et al., ToDS 2005]

Prinzip:

- Idee: Datenorientierte Suche statt Datenraumorientierte Suche
  - Vermeidung von Suche in “leeren” Datenraumpartitionen.
  - Kandidaten werden direkt über einen Index (R-Baum) ermittelt.
- Prioritäts-basierte Suche des nächsten Skyline-Objektes
  - Priorität entsprechend Manhattan-Distanz zum Koordinaten-Ursprung
  - Iterative Verfeinerung des Index (R-Baum) mittels Prioritätsliste (vgl. k-NN-Index-HS, Folie 63)

  → Verwendung eines Heaps aufsteigend sortiert über MINDIST(e,(0,0)), e := Seitenregion oder Objekt (Punkt)

  → Verwendung einer Liste mit bereits gefundenen Skylineobjekten zum Prunen von anderen Seitenregionen / Objekten

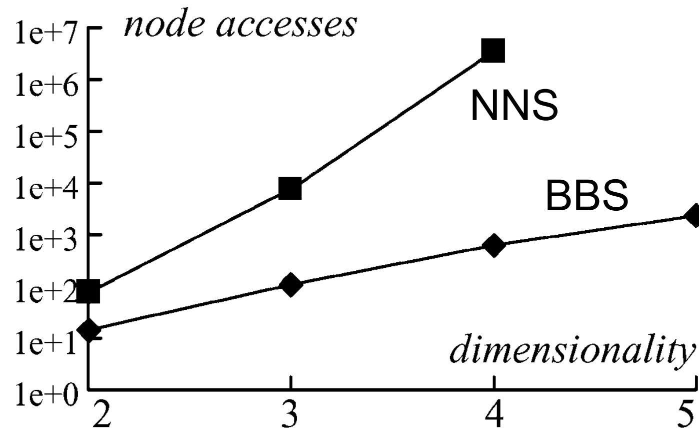


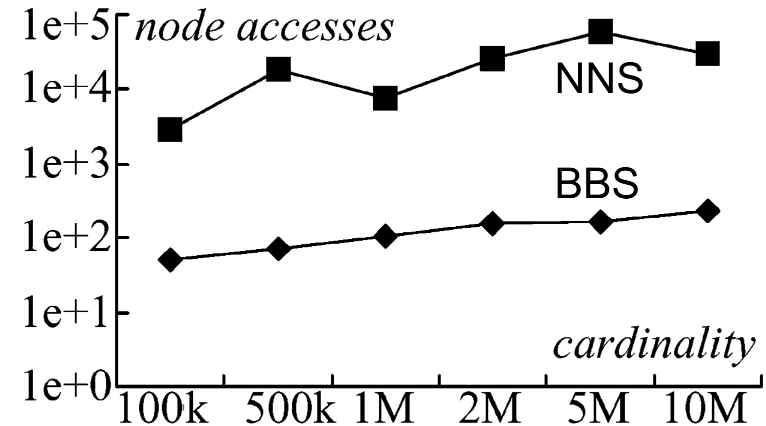
- Algorithmus:

```
Algorithm BBS (R-tree R)
S=Ø
Füge alle Einträge der Wurzel R in den Heap ein
Solange Heap nicht leer:
    entferne ersten Eintrag e
    wenn e von einem Punkt in S dominiert wird, verwerfe e
    sonst (e ist nicht dominiert)
        wenn e kein Datenpunkt ist
            für jedes Kind e_i von e
                falls e_i nicht von einem Punkt in S dominiert wird, füge e_i in den Heap ein
        sonst (e ist ein Datenpunkt)
                füge e in S ein
```

- Vorteile:
  - Vorzeitige Ausgabe von ersten Resultaten
  - Keine unnötige Partitionierung des Datenraums → geeignet auch für Suchraumdimensionen > 3 (im Gegensatz zu NNS)
  - BBS ist optimal bzgl. der Seitenzugriffe im R-Baum (I/O-optimal)

- Experimenteller Vergleich: NNS ←→ BBS
  – Datensatz: 1 Mio. Objekte gleichmäßig verteilt





  – BBS schlägt NNS bzgl. I/O-Kosten über mehrere Größenordnungen