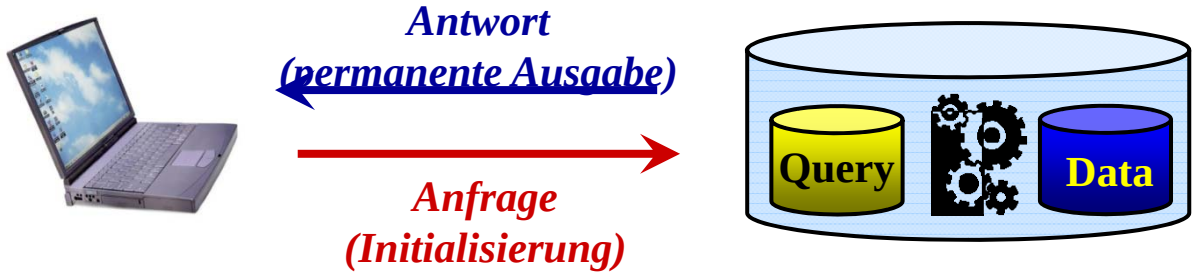
3.3 Methoden für kontinuierliche Anfragen (Continuous (Monitoring) Queries)

3.3.1 Allgemeines:

- Eine kontinuierliche Anfrage gibt zu einem gegebenen räumlichen Anfrageprädikat P die entsprechenden Ergebnisse für alle Zeitpunkte innerhalb eines (oder mehrere) gültige(n) Zeitbereiche(s)



- Typen von Objekten:
  1) Trajektorien aller beweglichen Objekte sind bekannt
  2) Objekte bewegen sich frei im Raum

3.3.2 Ansätze zu Methoden für kontinuierliche Anfragen

– Naiver Ansatz: Periodische Auswertung von Snapshot-Anfragen in der Gegenwart (siehe Kap. 3.2.3).

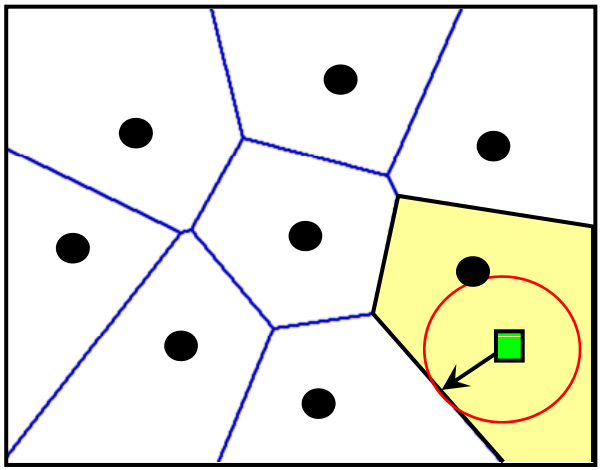
  Probleme:

  - Starke Redundanz von Auswertungsoperationen.
  - Es werden jedesmal ähnliche Antworten ausgegeben (Großer Overhead in der Antwortmenge).
  - Starke Verzögerung bei der Beantwortung von Anfragen (Echtzeitfähigkeit gefährdet!!!).

– Weitere Ansätze basieren auf den folgenden Prinzipien:

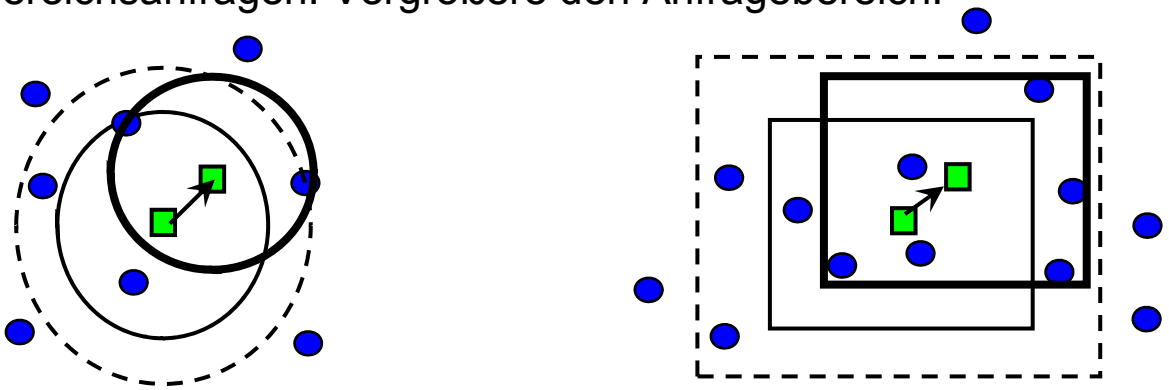
  - *Result Validation*
  - *Result Caching*
  - *Result Prediction*
  - *Incremental Evaluation*

– Prinzip des *Result Validation*:
  - Idee:
    – Verknüpfung von Anfrageergebnisse an Gültigkeitsbedingungen.
  - Gültigkeitsattribute:
    – **Gültigkeitszeit** $\mathbf{t_{valid}}$: Bestimmt das Zeitintervall für das die aktuelle Antwort ihre Gültigkeit behält.
    – **Gültigkeitsregion** $\mathbf{r_{valid}}$ (safe region): Solange ein Objekt sich in seiner Gültigkeitsregion aufhält behält die aktuelle Antwort ihre Gültigkeit.
  - Beispiel: Prinzip der Safe Regions bei der NN-Anfrage
    – DB-Objekte stationär.
    – Anfrageobjekt bewegt sich .
    – Über Voronoi-Diagramm ist die NN-Nachbarschaft aller DB-Objekte vorberechnet.
    – Solange sich das Anfrage-Objekt innerhalb einer Voronoizelle (safe region) aufhält muß keine Aktualisierung des aktuellen Ergebnis erfolgen.



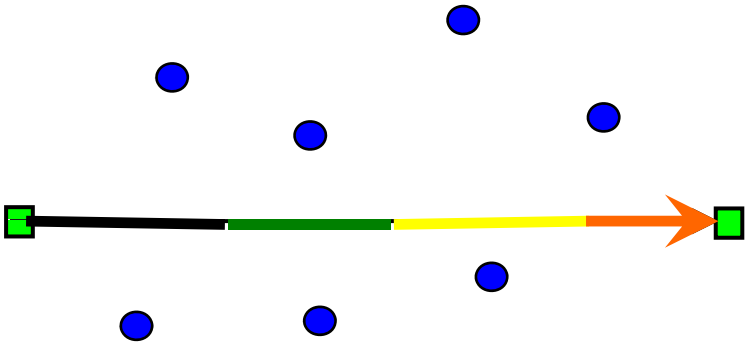
  - Herausforderung: Bestimmung von t_{valid} bzw. r_{valid}

– Prinzip des *Result Caching*:
  - Idee:
    – Antwortmengen von direkt aufeinanderfolgenden Anfragen sind sehr ähnlich zueinander.
    – Bei der Bearbeitung einer Anfrage werden mehr Objekte gesucht als für die aktuelle Antwort nötig sind (*caching* möglicher Ergebnisse zukünftiger Anfrageiterationen).
  - Beispiele:
    – kNN Anfragen: Suche mehr als die k nächsten Nachbarn.
    – Bereichsanfragen: Vergrößere den Anfragebereich.



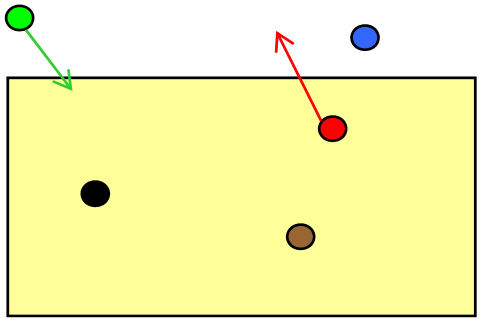
  - Herausforderung:
    – Bestimmung wieviel Vorberechnet werden muß (Cache Overhead)

– Prinzip des *Result Prediction*:
  - Idee:
    – Falls die zukünftige Bewegung der Objekte bekannt ist, kann das Anfrageergebnis vorberechnet werden.
    – Berechnung kann z.B. durch den TPR-Baum unterstützt werden.
  - Beispiel:
    – Kontinuierliche NN Anfrage.
    – Die Trajektorie ist in Segmente aufgeteilt, sodaß für jedes Segment das NN Anfrageergebnis eindeutig ist.



  - Die Anfrage wird als Snapshot-Anfrage ausgeführt.
  - Sie behält Ihre Gültigkeit für einen längeren Zeitraum.
  - Sobald sich die Trajektorie ändert wird das Anfrageergebnis aktualisiert.

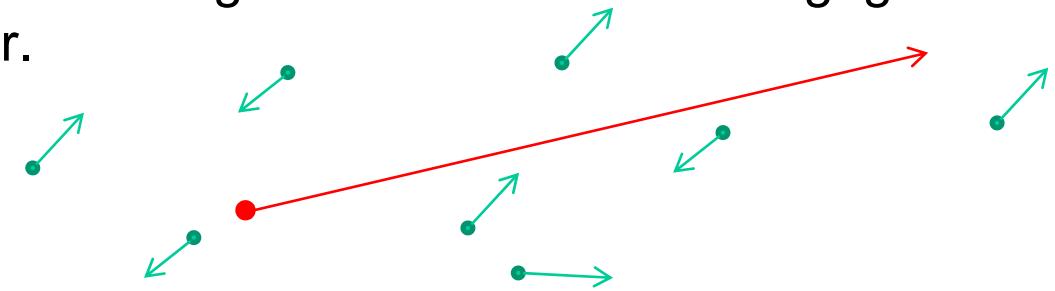
- Prinzip des *Incremental Evaluation*:
  - Idee:
    - Änderungen sind oft nur sehr lokal.
    - Anfrage wir einmalig ausgeführt.
    - Dann wird lediglich die Antwortmenge aktualisiert falls notwendig.
  - Unterscheidung zwischen *positiven* und *negativen* Aktualisierungen.
  - Positive Aktualisierung: Hinzufügen von Objekten
  - Negative Aktualisierung: Entfernen von Objekten
  - Nur Objekte die die Grenze der Anfrageregion überschreiten sind zu berücksichtigen.



  - Herausforderung:
    - Erkennung wann ein Objekt die Grenze der Anfrageregion überschreitet.

3.3.3 Methoden für Kontinuierliche Anfragen mit bekannten Trajektorien

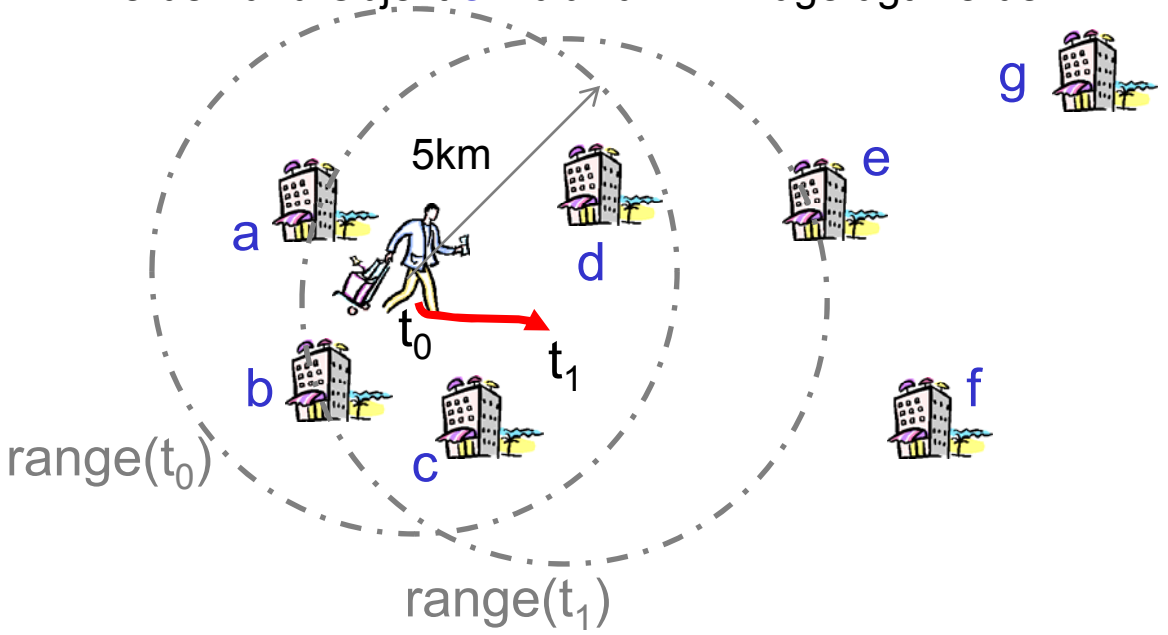
– Annahme:
  - Trajektorien der Objekte sind bekannt
    (zumind. bis zu einem gegebenen gültigen Zeithorizont)
  - Objekte bewegen sich innerhalb eines gegebenen Zeithorizonts linear.



– Im Folgenden werden zwei Ansätze diskutiert:
  - Zeit-parametrisierte Fenster- und NN-Anfragen (TP-Queries) [TP02]
    – Prinzip des ***Result Validation***.
  - Kontinuierliche NN-Anfragen (CNN) [TPS02]
    – Prinzip des ***Result Prediction***.

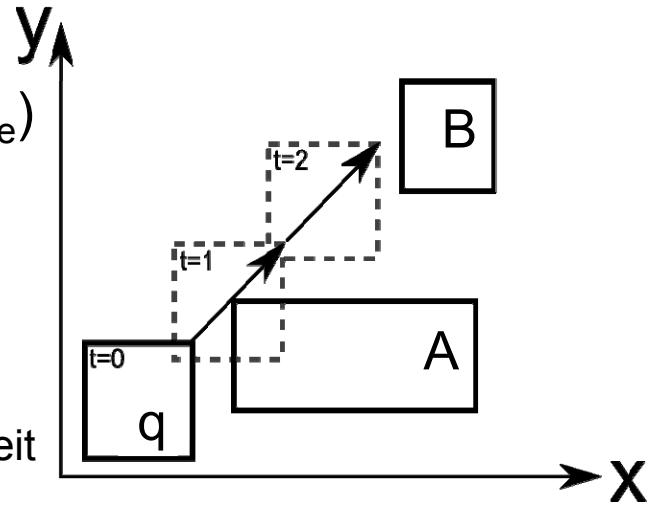
– Zeit-Parametrisierte (TP-) Anfragen in ST-DBS
  - Basieren auf dem Prinzip *Result Validation* (siehe 3.3.2).
  - Eine Zeit-Parametrisierte Anfrage gibt Triple (R,T,C) als Antwort zurück:
    – R: Menge von Objekten, die das Anfrageprädikat zum Zeitpunkt t_0 (Gegenwart) erfüllen.
    – T: Entspricht Zeitpunkt t_1 bis zu dem das (aktuelle) Anfrageergebnis gültig bleibt. (Gültigkeitszeit)
    – C: Menge von Objekten die das aktuelle Ergebnis R zum Zeitpunkt t_1 beeinflussen.
  - Mit den Mengen *C* und *R* kann das initiale Ergebnis zum Zeitpunkt T aktualisiert werden.
  - Herausforderung: Berechnung von T und C
  - Unterstützung der Anfrage durch Index:
    – Unterstützung der Anfrage durch Verwendung des TPR-Baums zur Indexierung der Datenbankobjekte.
    – Anfrage beschränkt auf den Zeithorizont für den die gegebenen Objekttrajektorien (inkl. Anfrageobjekttrajektorie) gültig sind.

- Beispiel:
  - Ein Reisender fragt alle Hotels im Umkreis von 5km zur aktuellen Position (Position zu Zeitpunkt t_0) an.
  - Zusätzlich zur aktuellen Ergebnismenge R= {a,b,c,d}, die aktuell in der Nähe des Reisenden liegt,
  - enthält das Ergebnis die Zeit T = t_1 = t_0 + 7 Min., bis zu der die Antwort aktuell ist (gegeben die Richtung und Geschwindigkeit des Reisenden),
  - und zusätzlich die Objekte C = {a,b,e}, Objekten die das aktuelle Ergebnis R zum Zeitpunkt t_1 beinflussen, d.h. Objekte a, b müssen aus R entfernt werden und Objekt e muß zu R hinzugefügt werden.

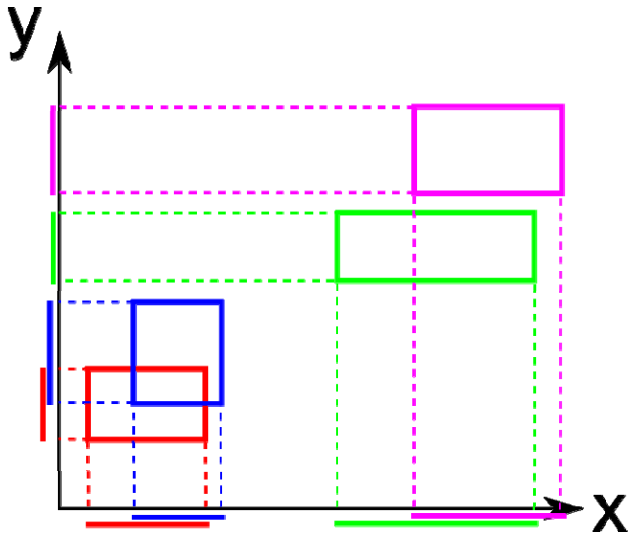


– Zeit-Parametrisierte Fensteranfrage (TP-Window Query)
[TP02]

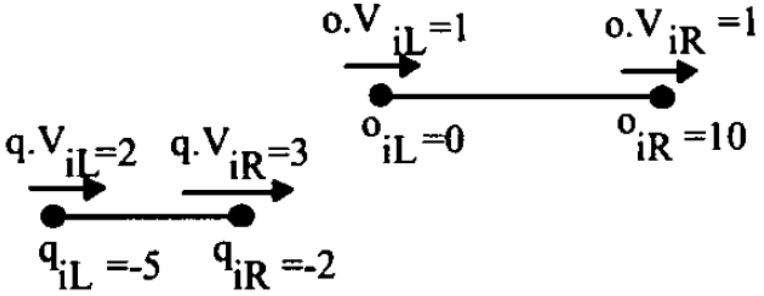
- Annahme:
  - Anfrage-Fenster und Objekte bewegen sich linear.
  - Objekte sind in einem TPR-Baum organisiert.
- Berechnung von T über *Influence Time* $T_{INF}(o,q)$ eines Objektes o bzgl. Anfragefenster q.
- T_{INF} wird nicht nur für Objekte sondern auch für Seitenregionen berechnet.
- T ergibt sich aus: $T = \min_{o \in DB}\{T_{INF}(o,q)\}$
- Für die Berechnung von $T_{INF}(o,q)$ benötigt man die Schnitt-Periode $[T_s,T_e)$ innerhalb der das Objekt o von q geschnitten wird.
- Beispiel:
  - $T_s(A) = 0{,}5$; $T_e(A) = 1{,}5$; $T_s(B) = 2{,}5$
  - Ergebnis ändert sich zum ersten mal zur Zeit $T_s(A) = 0.5 \Rightarrow T_{INF}(A,q) = 0.5 \Rightarrow T = 0{,}5$



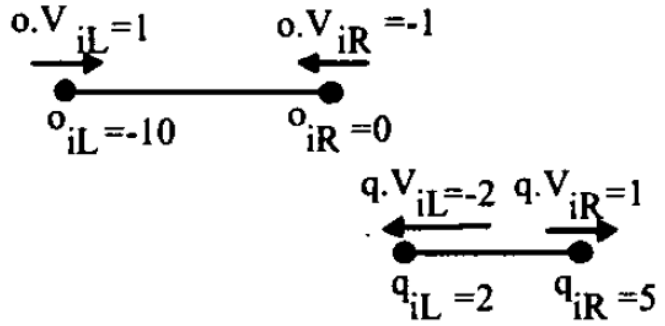
- Berechnung der Schnittperiode für Seitenregionen:
  - Schnittperiode $[T_s,T_e)$ über Schnitt zweier dynamischer Rechtecke berechnen.
  - d-dimensionales Rechteck gegeben über:
    - » Koordinaten: $((o_{1L}, o_{1R}), \ldots, (o_{dL}, o_{dR}))$
    - » Ausdehnungsgeschwindigkeit: $((o.V_{1L},o.V_{1R}),\ldots,(o.V_{dL},o.V_{dR}))$
  - Zwei (statische) Rechtecke schneiden sich, wenn sie sich bzgl. aller Dimensionen schneiden.



- Wann schneiden sich dynamische Rechtecke?
  - Schnitt in jeder Dimension $1 \leq i \leq d$ separat berechnen, d.h. die Schnittintervalle $[T_{is},T_{ie})$ berechenen.
  - Schnitt beginnt, wenn sich die Untergrenze des einen Rechtecks mit der Obergrenze des anderen Rechtecks trifft (und sich alle anderen Dimensionen bereits schneiden)



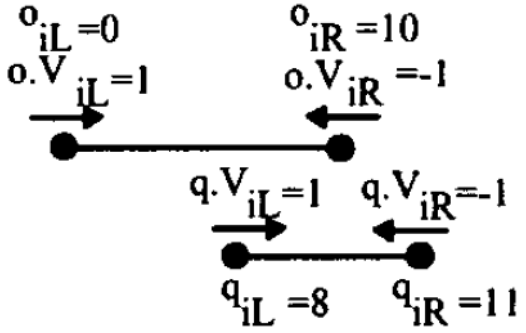
Fall 1: q liegt links von o



Fall 2: q liegt rechts von o

  - $T_{iLR} = (o_{iL}-q_{iR})/(q.V_{iR}-o.V_{iL}) = 1$ (Fall 1: rechte Grenze q schneidet linke Grenze o),
    $T_{iLR} = \infty$, falls $o.V_{iL} \geq q.V_{iR}$.
  - $T_{iRL} = (o_{iR}-q_{iL})/(q.V_{iL}-o.V_{iR}) = 2$ (Fall 2: linke Grenze q schneidet rechte Grenze o),
    $T_{iRL} = \infty$, falls $o.V_{iR} \leq q.V_{iL}$.
  - $T_{is} = \min(T_{iLR},T_{iRL})$.

- Schnitt endet, wenn sich die Obergrenze des einen Rechtecks mit der Untergrenze des anderen Rechtecks trifft (alle anderen Dimensionen können sich weiter schneiden).
- T_{iLR} und T_{iRL} werden wie zuvor berechnet.
- Für die Fälle 1 und 2 ergibt sich für T_{ie}: $T_{ie} = \max(T_{iLR}, T_{iRL})$.
- Für Fall 3: q wird von o zum Zeitpunkt t_0 bereits geschnitten:



Fall 3: q schneidet o

- $T_{is} = 0$.
- $T_{ie} = \min(T_{iLR}, T_{iRL}) = \min(5.5, 1) = 1$.
- $[Ts,Te) = \bigcap_{i=1,..,d} [T_{is}, T_{ie})$
- $T_{INF}(o,q) = T_s$ (Fall 1 u. 2); $T_{INF}(o,q) = T_e$ (Fall 3).

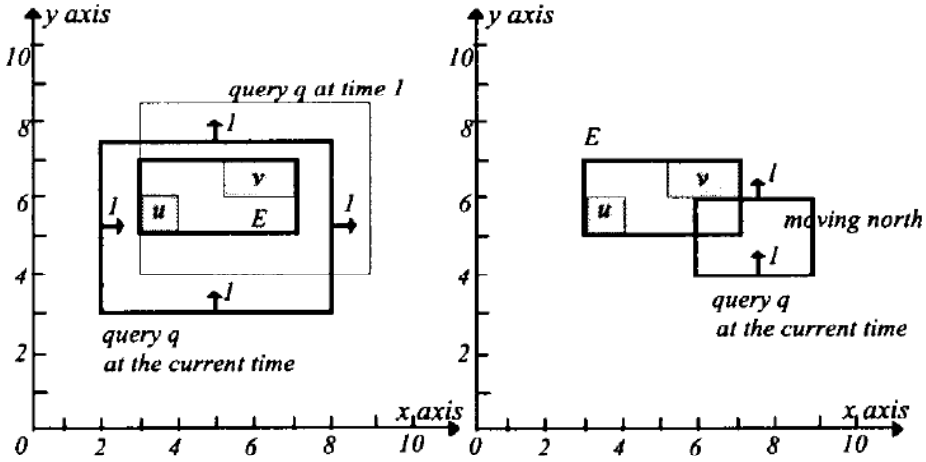
- Algorithmus zur Schnittberechnung:

Compute_Intersection_Period (o,q)

$[T_s, T_e[= [0,\infty[$

For each dimension i	Betrachte den Schnitt bei jeder Dimension einzeln
$T_{iLR} = (o_{iL}-q_{iR})/(q.V_{iR}-o.V_{iL})$	Zeitpunkt, zu dem der rechte Punkt von q den linken von o passiert
if $T_{iLR} < 0$ then $T_{iLR} = \infty$	Die Punkte treffen sich nie, falls $T_{iLR} < 0$
$T_{IRL} = (o_{iR}-q_{iL})/(q.V_{iL}-o.V_{iR})$	Zeitpunkt, zu dem der rechte Punkt von o den linken von q passiert
if $T_{iRL} < 0$ then $T_{iRL} = 0$	Die Punkte treffen sich nie, falls $T_{iRL} < 0$
if $[o_{iL}, o_{iR}]$ does not intersect $[q_{iL}, q_{iR}]$	Zu t=0 schneiden sich de beiden Intervalle noch nicht, dann:
$T_{is} = \min(T_{iLR}, T_{iRL})$	Berechne Beginn …
$T_{ie}=\max(T_{iLR},T_{iRL})$	… und Ende der Intervall-Überschneidung.
else	Punkte schneiden sich zu t = 0 (aktueller Zeitpunkt)
$T_{is} = 0$	Startpunkt ist dann 0, und …
$T_{ie} = \min(T_{iLR}, T_{iRL})$	… Endzeitpunkt ist das Minimum der zwei Überschneidungen
$[T_s, T_e[= [T_s, T_e[\cap[T_{is}, T_{ie}[$	Schneide die Intervallschnittzeiten aller Dimensionen, dadurch berechnet sich das Zeitintervall, in dem das Objekt o die Query q schneidet.
return $[T_s, T_e[$	T_{INF} ist T_S, falls keine Überschneidung zu t=0 und T_E sonst

- Berechnung von T_{INF} für einen inneren Knoten E:
  - Ziel: Pruning von Teilbäumen, Vermeidung der Exploration von E falls möglich.
  - T_{INF} der Objekte innerhalb E kann konservativ abgeschätzt werden: $T_{INF}(E,q) \leq T_{INF}(e,q)$, $\forall e \in$ Teilbaum unter E.
  - Fallunterscheidung für die Berechnung von $T_{INF}(E,q)$: Gegeben sei [Ts,Te) für E und q:
    - » $T_{INF}(E,q) = T_s$, falls zum Zeitpunkt t_0 E q **nicht schneidet**.
    - » $T_{INF}(E,q) = 0$, falls zum Zeitpunkt t_0 E q nur **teilweise schneidet**.
    - » $T_{INF}(E,q) = T_{PI}(E,q)$, falls zum Zeitpunkt t_0 E in q **vollständig enthalten** ist. $T_{PI}(E,q)$ entspricht dem Zeitpunkt an dem E von q nur teilweise geschnitten wird.



- Berechnung von $T_{PI}(E,q)$:
  - Ähnlich zur Berechnung von $T_{INF}(E,q)$.
  - Dieses mal müssen nicht entgegengesetzte Intervallenden von E und q verglichen werden, sondern die gleichen
  - Wieder Reduktion auf einzelne Dimensionen möglich
  - Relevant ist die Dimension, bei der E zuerst nicht mehr in q enthalten ist

Compute_$T_{PI}(E,q, [T_s,T_e[)$

$T_{PI} = \infty$
For each dimension i
  $T_{iLL} = (E_{iL}-q_{iL})/(q.V_{iL}-E.V_{iL})$
  $T_{iRR} = (E_{iR}-q_{iR})/(q.V_{iR}-E.V_{iR})$
  if $T_{iLL} \in [T_s,T_e[$ and $T_{iRR} \in [T_s,T_e[$
    $T_{iPi} = \min(T_{iLL}, T_{iRR})$
  if $T_{iLL} \in [T_s,T_e[$ and $T_{iRR} \notin [T_s,T_e[$
    $T_{iPi} = T_{iLL}$
  if $T_{iLL} \notin [T_s,T_e[$ and $T_{iRR} \in [T_s,T_e[$
    $T_{iPi} = T_{iRR}$
  if $T_{iLL} \notin [T_s,T_e[$ and $T_{iRR} \notin [T_s,T_e[$
    $T_{iPi} = \infty$
  $T_{PI} = \min(T_{PI}, T_{iPi})$

Return T_{PI}

T_{PI}: Zeit, zu der E nicht mehr vollständig in q enthalten ist
Betrachte wieder jede Dimension einzeln
T_{iLL}: Zeit, zu der der linke Punkt von q den linken Punkt von E trifft
T_{iRR}: Zeit, zu der der rechte Punkt von q den rechten Punkt von E trifft
Es ist immer der kleinere Zeitpunkt relevant. Liegen beide im Zeitintervall, zu dem sich E und q schneiden, wird das Minimum gewählt.

Liegt nur T_{iLL} im Zeitintervall, wird T_{iLL} gewählt

Liegt nur T_{iRR} im Zeitintervall, wird T_{iRR} gewählt

Liegt keiner im Zeitintervall, verlässt E niemals q, und T_{iPi} wird auf unendlich gesetzt
Interessant ist das Minimum über alle Dimensionen, weil eine Dimension ausreicht, damit E nicht mehr vollständig in q enthalten ist

– Zeit-Parametrisierte NN-Anfrage (TP-NN Query) [TP02]

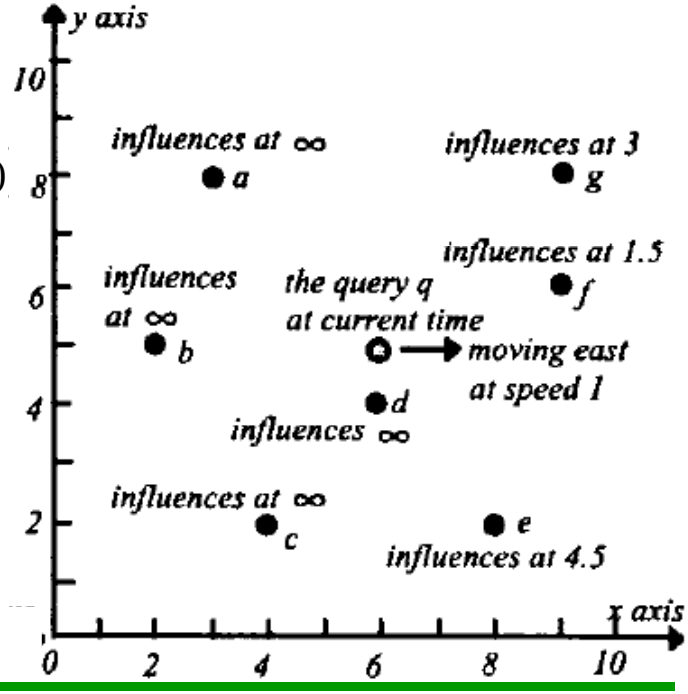
- Berechnung von T über die Berechnung von $T_{INF}(o,q)$, $\forall o \in DB$
- $T_{INF}(o,q)$ = minimale Zeit, zu der o näher an q ist als der aktuelle (d.h. zum Zeitpunkt t_0) Nächste Nachbar P_{NN} von q.
- Berechnung von $T_{INF}(o,q)$ für Objekte
  - $T_{INF}(o,q)$: Zeit zu der dist(o(t),q(t)) = dist(P_{NN}(t)q(t)) und t > 0
  - Umformung in $At^2+Bt+C=0$ mit (Euklidischer Distanz):

$$A = \sum_{i=1}^{n} \left[(o.V_i - q.V_i)^2 - (P_{NN}.V_i - q.V_i)^2 \right]$$

$$B = \sum_{i=1}^{n} 2\left[(o_i - q_i)(o.V_i - q.V_i) - (P_{NN\,i} - q_i)(P_{NN}.V_i - q.V_i) \right]$$

$$C = \sum_{i=1}^{n} \left[(o_i - q_i)^2 - (P_{NN\,i} - q_i)^2 \right]$$

  - Falls Ungleichung für kein t gelöst werden kann: setzte $T_{INF} = \infty$
  - Sonst: $T_{INF}(o,q)$ = t mit dem die Ungleichung gelöst werden kann.



- Berechnung von $T_{INF}(o,q)$ für Schlüssel innerer Knoten
  - Konservative Abschätzung möglich über MINDIST, dann Berechnungen aber sehr komplex (Fallunterscheidungen in MINDIST)
  - Stattdessen: MINDIST unterschätzen, um korrektes Ergebnis zu ermöglichen => orthogonale Distanz zu einer ausgewählten Kante/Seitenfläche des MBRs
    - » Fall 1: MINDIST würde über Eckpunkt des MBRs berechnet werden: Gewählte Kante ist die vom MBR am weitesten entfernte Kante, die mit dem Eckpunkt verbunden ist
    - » Fall 2: MINDIST würde über Kante berechnet werden: Wähle diese Kante
  - Sei l die gewählte Kante, dann kann wiederum die Ungleichung aufgestellt werden:

$$MINDIST(l(t), q(t)) \le dist(P_{NN}(t)q(t)) \Leftrightarrow$$

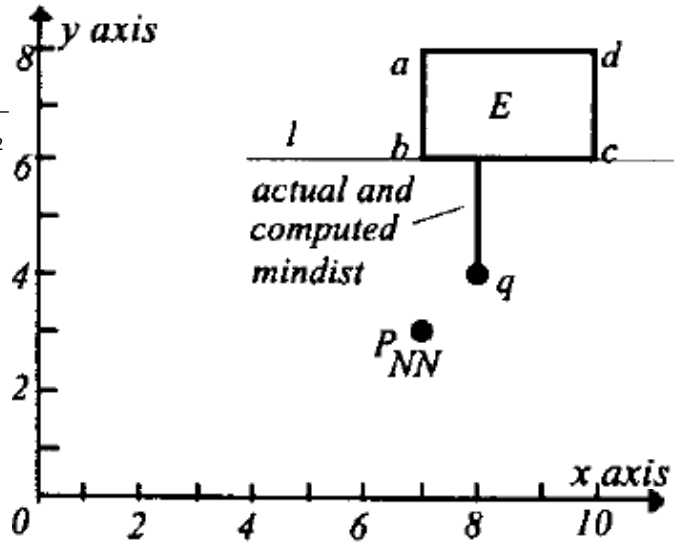
$$|(q_i - l_i) + t(q.V_i - l.V_i)| \le \sqrt{\sum_{i=1}^{n}[(PNN_i - q_i) + t(PNN.V_i - q.V_i)]^2}$$

  - Umformung ergibt eine Gleichung der Form $At^2+Bt+C\le 0$ mit

$$A = (l.V_i - q.V_i)^2 - \sum_{i=1}^{n}\left[P_{NN}.V_i - q.V_i\right]^2$$

$$B = 2(o_i - q_i)(o.V_i - q.V_i) - \sum_{i=1}^{n} 2(P_{NN\,i} - q_i)(P_{NN}.V_i - q.V_i)$$

$$C = (l_i - q_i)^2 - \sum_{i=1}^{n}\left[(P_{NN\,i} - q_i)^2\right]$$



- Kontinuierliche Nächste-Nachbarn-Anfrage (CNN Query)
  - Basiert auf dem Prinzip des *Result Prediction* (siehe Kap. 3.3.2)
  - Gegeben: [TPS02]
    - Menge *P* von statischen Objekten (Datenbankobjekte)
    - Trajektorie q eines beweglichen Anfrageobjektes als Liniensegment q=[s,e]
  - Gesucht: Ermittlung von zusammenhängenden Trajektorien/Zeit-Abschnitte für die gilt:
    - Konsistenz (bzgl. des Anfrageprädikates)
      - » Ein Trajektorien/Zeit-Abschnitt heißt *konsistent* bzgl. eines (räumlichen) Anfrageprädikates, wenn für je zwei Punkte in dem Abschnitt das entsprechende Anfrageergebnis gleich ist.
    - Vollständigkeit (Maximalität)
      - » Ein konsistenter Trajektorien/Zeit-Abschnitt T heißt *vollständig*, wenn kein weiterer konsistenter Trajektorien/Zeit-Abschnitt T'≠ T existiert, der T vollständig enthält.
  - Beispiel-Anfrage:

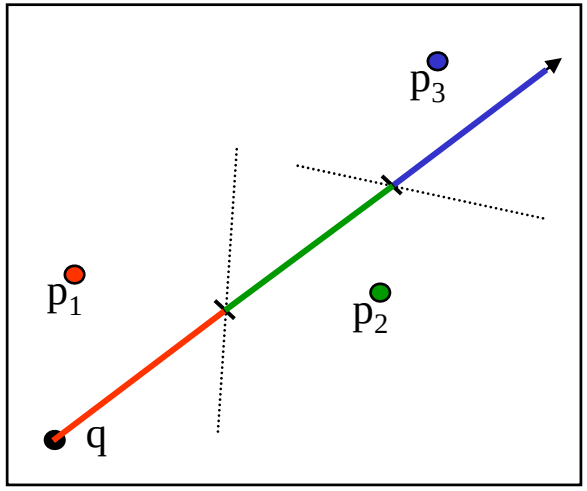
    „ Auf meiner Fahrt von Siegen nach Erfurt, suche mir alle nächstgelegenen Tankstellen“

- Zerlegung der Trajektorie von q in vollständige und konsistente Teilsegmente $t_i = [s_i, s_{i+1}]$, so dass:

$$s_i, s_{i+1} \in q$$

$$\forall p_1, p_2 \in [s_i, s_{i+1}]: \exists n_1 \in NN(p_1), \exists n_2 \in NN(p_2): n_1 = n_2$$

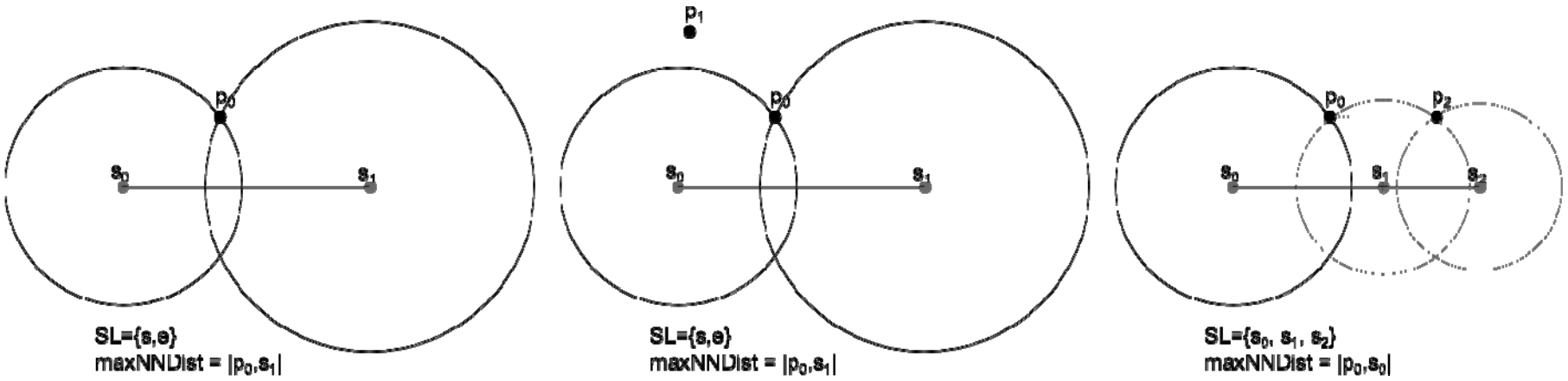
wobei $s_0 = s$ und $NN(p_i)$ = nächster Nachbar von p_i (deterministisch)



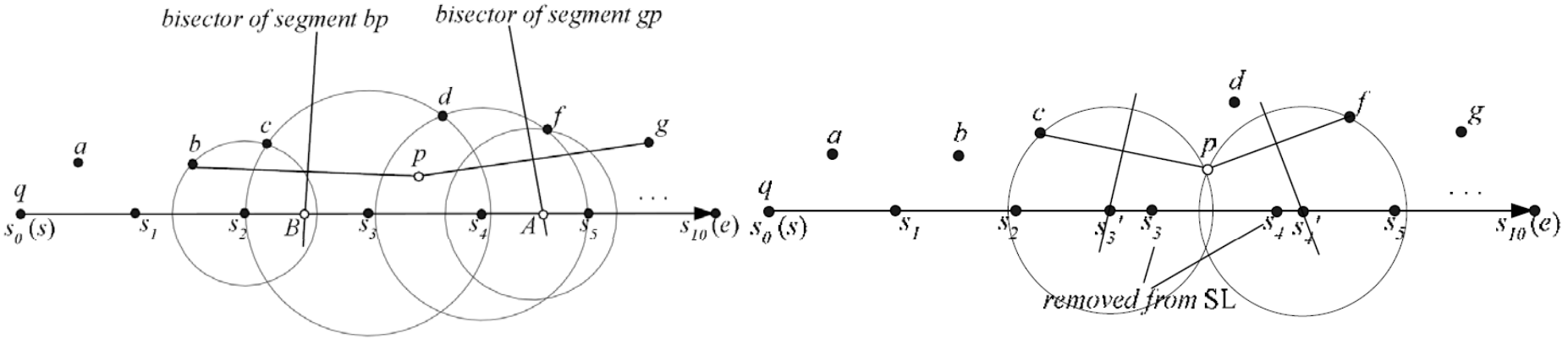
Bem.: Konsistenz der Teilsegmente bezieht sich auf eine nicht-deterministische NN-Suche (Konsistenz bzgl. der offenen Intervalle).

- Ausgabe von Tupeln $(t_i, NN(s_i))$ aller Teilsegmente t_i mit entsprechenden Anfrageergebnis $NN(s_i)$.

- Idee:
  - SL: Liste aller Segment-Endpunkte von q=[s,e], sortiert nach Entfernung zu s.
  - Initialisierung von SL mit Start- und Endpunkt: SL_{init} = [s,e],
  - NN(s) = NN(e) = p_0 (p_0 beliebig)
  - Fällt ein weiterer Punkt p_i in die NN-Distanz dist(NN(s_i),s_i) eines Segment-Endpunktes $s_i \in SL$, wurde ein neuer NN für Punkte auf q gefunden
  - In diesem Fall berechne das Äquidistanzpotential AD zwischen NN(s_i) und p_i sowie den Schnittpunkt sp zwischen AD und q.
  - Füge sp in SL ein, setzte NN(sp) = {NN(s_i), p_i}



- Kontinuität
  - Sei S_{COVER}={sl ∈ SL| dist(sl, NN(sl)) > dist(sl,p)}
  - Kontinuität: alle $s_i \in S_{COVER}$ sind benachbart: S_{COVER} ={s_i,…, s_{i+k}}
  - Ermöglicht binäre Suche über $s_i \in S_{COVER}$ (linkes Bild)
  - Weitere $s_j \in S_{COVER}$ sind immer Nachbarn von s_i. (Performanz!)
  - Ermöglicht einfaches Update von SL:
    - *$u=NN(s_{i-1})\cap NN(S_i)$; $v=NN(s_{i+k}) \cap NN(S_{i+k+1})$;*
    - SL = SL \ S_{COVER}
    - *$s_{i'}$:neuer Segmentpunkt am Schnitt zwischen q und ⊥(u, p)* mit $NN(s_{i'}) = p$
    - *$s_{i+1'}$: neuer Segmentpunkt am Schnitt zwischen q und⊥(v, p) mit $NN(s_{i+1'})=p$*



- Algorithmus (Depth-First):

SL_0 =[s,e] //Splitpunkte; SL sei sortiert nach aufsteigender Distanz zu s
NN(e) = NN(s) = Ø //Aktuelle NN‘s sind noch nicht bekannt

Function CNN-depth(R-Tree-Node N, SL, maxNNdist(SL)):
Falls N is Leaf Node:
Für alle p in N:
S_{COVER}: {sl ∈ SL| dist(sl, NN(sl)) > dist(sl,p)} // S_{COVER} ={s_i,…, s_{i+k}};
u=NN(s_{i-1})∩NN(s_i); v=NN(s_{i+k}) ∩NN(s_{i+k+1});
SL = SL \ S_{COVER}
Füge $s_{i'}$:neuer Segmentpunkt am Schnitt zwischen q und ⊥(u, p) mit NN($s_{i'}$) =p in SL ein
Füge $s_{i+1'}$: neuer Segmentpunkt am Schnitt zwischen q und⊥(v, p) mit NN($s_{i+1'}$)=p in SL ein
update maxNNdist(SL) falls notwendig
Falls N Directory Node:
Für alle p in N sortiert nach MINDIST(p,q):
Falls MINDIST(p,q) > maxNNdist(SL): continue; //kann immer upgedated werden, deshalb billig
Falls ∄sl in SL: MINDIST(p,sl) < dist(sl, NN(sl)): continue;
SL = CNN-depth(p, SL, maxNNdist(SL))
return SL

Ergebnis: alle Tupel ([s_i, s_{i+1}], {NN(s_i)| NN(s_i)=NN(s_j)})

- Algorithmus (Best-First):

Function CNN-best(R-Tree-Node N, q=[s,e]):

Setzte SL_0= {s,e} //Splitpunkte; SL sei sortiert nach aufsteigender Distanz zu s

Setzte aktuelle NN‘s: NN(e) = NN(s) = Ø //Aktuelle NN‘s sind noch nicht bekannt

APL = [N]: LIST of R-Tree-Node ORDERED by ASCENDING MINDIST to q

maxNNdist(SL) = ∞

while APL ≠ Ø

Hole ersten Entry F aus APL

Falls F is Leaf Entry (=Datenpunkt):

S_{COVER}: {sl ∈ SL| dist(sl, NN(sl)) > dist(sl,p)} // S_{COVER} ={s_i,…, s_{i+k}}

u=NN(s_{i-1})∩NN(S_i); v=NN(s_{i+k}) ∩NN(S_{i+k+1});

SL = SL \ S_{COVER}

Füge $s_{i'}$:neuer Segmentpunkt am Schnitt zwischen q und ⊥(u, p) mit NN($s_{i'}$) =p in SL ein

Füge $s_{i+1'}$: neuer Segmentpunkt am Schnitt zwischen q und⊥(v, p) mit NN($s_{i+1'}$)=p in SL ein

update maxNNdist(SL) falls notwendig

Falls N Directory Node:

Für alle p in N:

Falls MINDIST(p,q) > maxNNdist(SL): prune p;

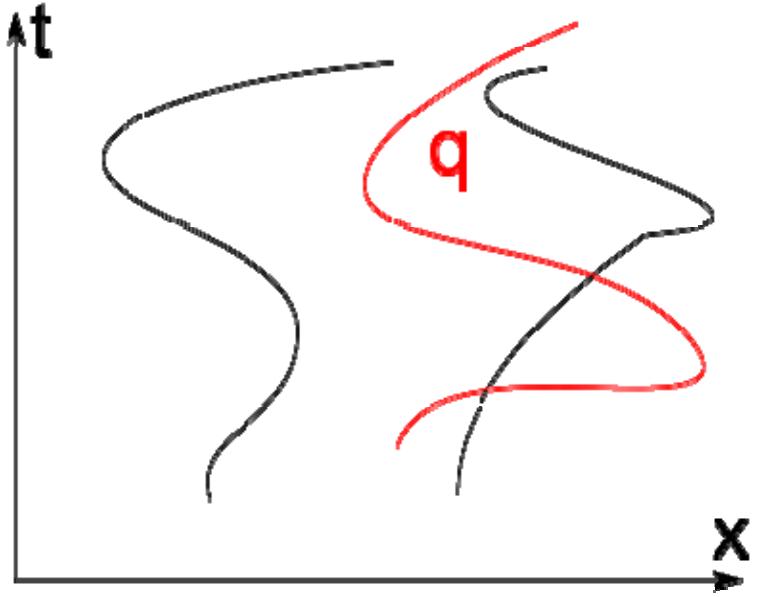
Falls ∄sl in SL: MINDIST(p,sl) < dist(sl, NN(sl)): prune p;

Füge p in APL ein

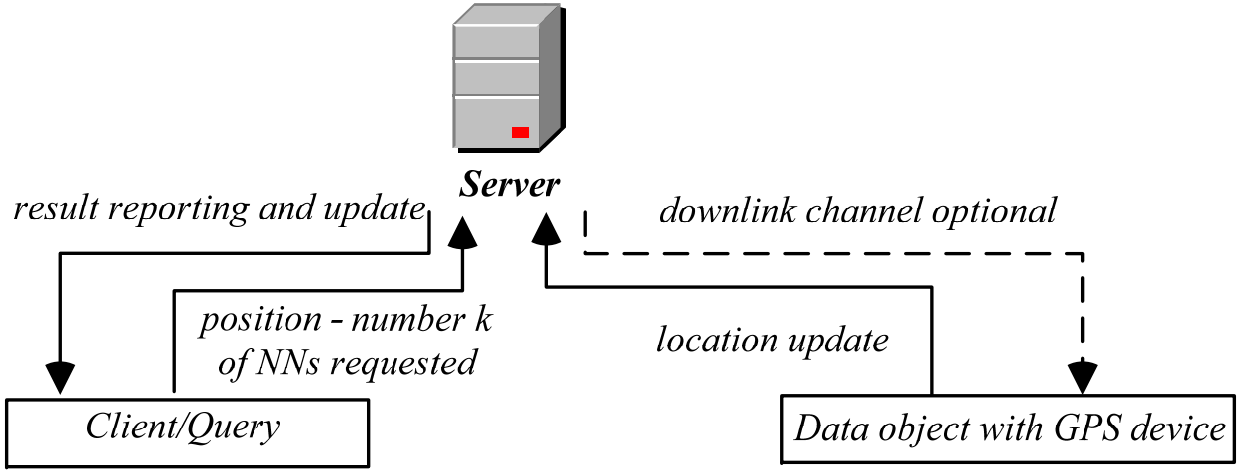
Ergebnis: alle Tupel ([s_i, s_{i+1}], {NN(s_i)| NN(s_i)=NN(s_j)})

3.3.4 Monitoring-Anfragen

- Query wird initialisiert und soll dann über einen längeren Zeitraum eine Ergebnismenge aktuell halten
- Oft Client/Server-basiert
- Oft keine Annahmen bzgl. der Beweglichkeit der Objekte (z.B. keine Annahme einer linearen Bewegung)

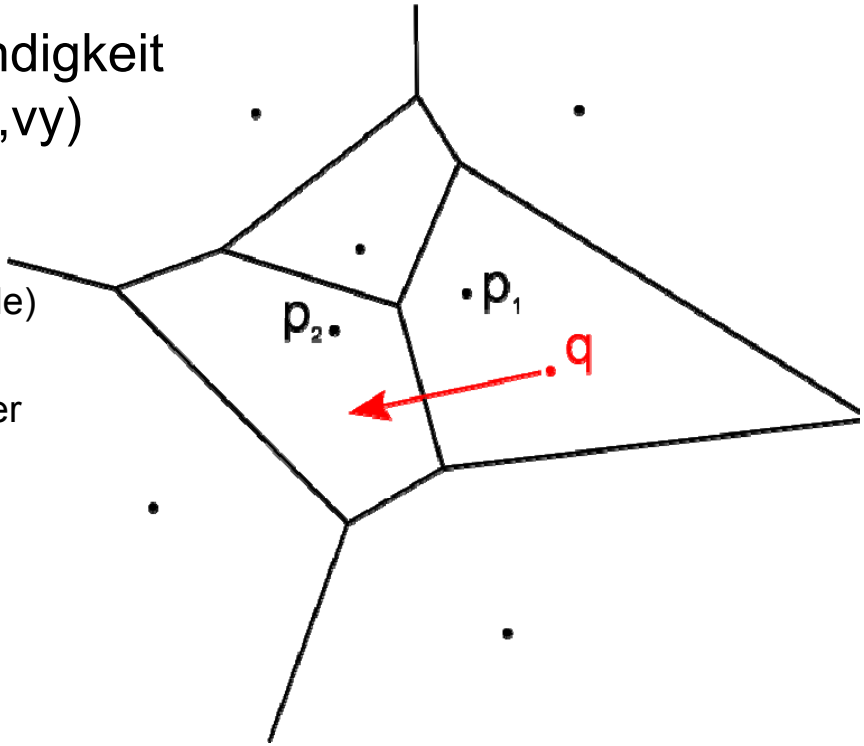


- Continuous NN Monitoring:
  - Annahmen:
    - Queries sowie Objekte in der Datenbank bewegen sich
    - Bewegung zufällig und damit unvorhersehbar
    - Zentraler Server überwacht die kNN jeder Query und sendet sie an einen Client
  - Ziele:
    - CPU-Overhead des Servers minimieren
    - Netzwerklast für Ortsupdates minimieren
  - Oft keine Verwendung komplexer Indices, da Updates meist sehr teuer

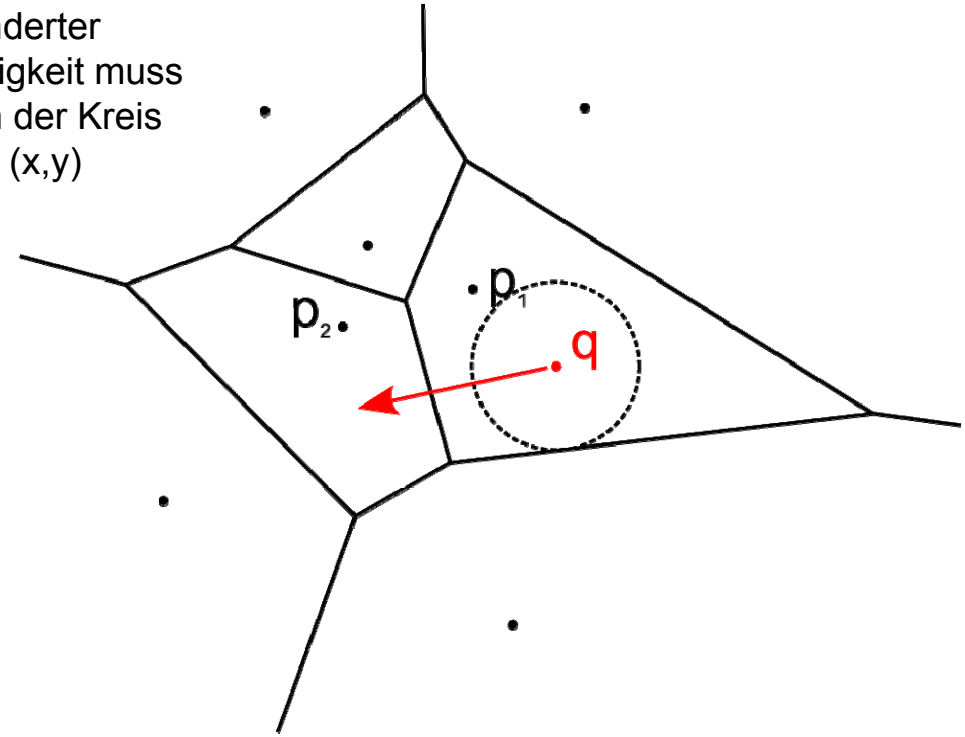


– NN-Queries mit Validitätsinformation [ZL01]

- Ziel: NN-Anfrage eines beweglichen Objektes mit der zusätzlichen Information, wie lange das Ergebnis gültig bleiben wird
- Querying the Presence/Future
- Voronoi-Diagramm der DB wird im Voraus berechnet
- Query q: Position und Geschwindigkeit des Query-Objektes: q = (x,y,vx,vy)
- Antwort: Tupel (p1, t, p2) mit
  - » p1: Punkt aus der DB, der aktuell q am nächten ist (über Voronoizelle)
  - » t: Validitätszeitraum
  - » p2: Punkt aus der DB, der nach der Zeit t q am nächtsten sein wird



- Problem: Validitätszeitraum erfordert Annahmen über das zukünftige Verhalten von Q
  - Lösung 1: nimm lineare Bewegung und konstante Geschwindigkeit an, sende Query erneut falls sich Richtung oder Geschwindigkeit ändert
  - Lösung 2:
    - » Berechne kürzeste Distanz d von q zu einer der Kanten der umgebenden Voronoizelle -> Minimale Distanz, die auch bei Richtungsänderung zurückgelegt werden muss, um die Voronoizelle zu verlassen
    - » Eine neue Query bei geänderter Richtung oder Geschwindigkeit muss erst gestellt werden, wenn der Kreis mit Radius d und Zentrum (x,y) verlassen wurde



[EGSV99]: M. Erwig, R. H. Güting, M. Schneider, M. Vizirgiannis. An Approach to Modeling and Querying Moving Objects Databases, In GeoInformatica 3(3), 1999.

[S99]: T. Sellis. Research Issues in Spatio-temporal Database Systems. In Proc. of SSD, 1999

[MGA03] M. F. Mokbel, T. M. Ghanem, W. G. Aref. Spatio-temporal Acces Methods. IEEE Data Engineering Bulletin, 2003.

[KGT99.pdf] G. Kollios, D. Gunopulos, V. J. Tsotras. On Indexing Mobile Objects. In Proc. of PODS, 1999.

[TP02]: Y. Tao, D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In Proc. of SIGMOD, 2002.

[TPS02]: Y. Tao, D. Papadias, Q. Shen. Continuous Nearest Neighbor Search. In Proc. of VLDB, 2002.

[ZL01] B. Zheng, D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In Proc. of SSTD, 2001.

[PJT00]: D. Pfoser, C. S. Jensen, Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In Proc. of VLDB, 2000.

[TFPL04]: Y. Tao, C. Faloutsos, D. Papadias, B. Liu. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In Proc. of SIGMOD, 2004.

[TPR00]: S. Saltenis, C. S. Jensen, S. T. Leutenegger, M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In Proc. of MOD, 2000.

[YPK05]:X. Yu, K. Q. Pu, N. Koudas. Monitoring k-Nearest Neighbor Queries Over Moving Objects. n.d.

[XHL90]: X. Xu, J. Han, and W. Lu. RT-Tree: An Improved R-Tree Indexing Structure for Temporal Spatial Databases. In Proc. of the Intl. Symp. on Spatial Data Handling, SDH, pages 1040–1049, July 1990.

[NS98]: M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In Proc. of the ACM Symp. on Applied Computing, SAC, pages 235–240, Feb. 1998.

[TP01]: Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In Proc. of the Intl. Conf. on Very Large Data Bases, VLDB, pages 431–440, Sept. 2001.

Auf die Veröffentlichungen kann vom Uni-Netzwerk aus zugegriffen werden.