
Kapitel 4

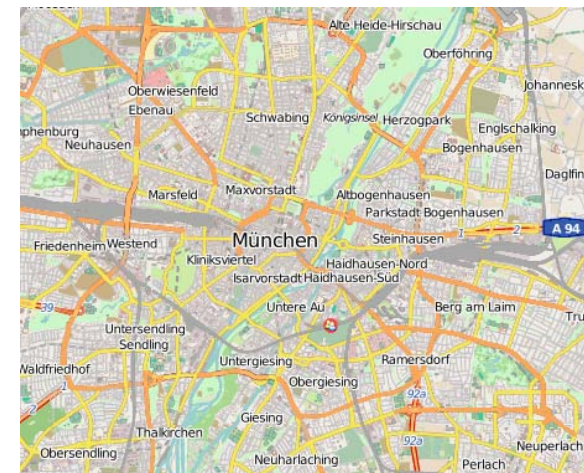
Anfragemethoden auf Verkehrsnetzwerke

Skript zur Vorlesung: Neue Trends zur Suche in modernen Datenbanksystemen
Wintersemester 2013/14, LMU München

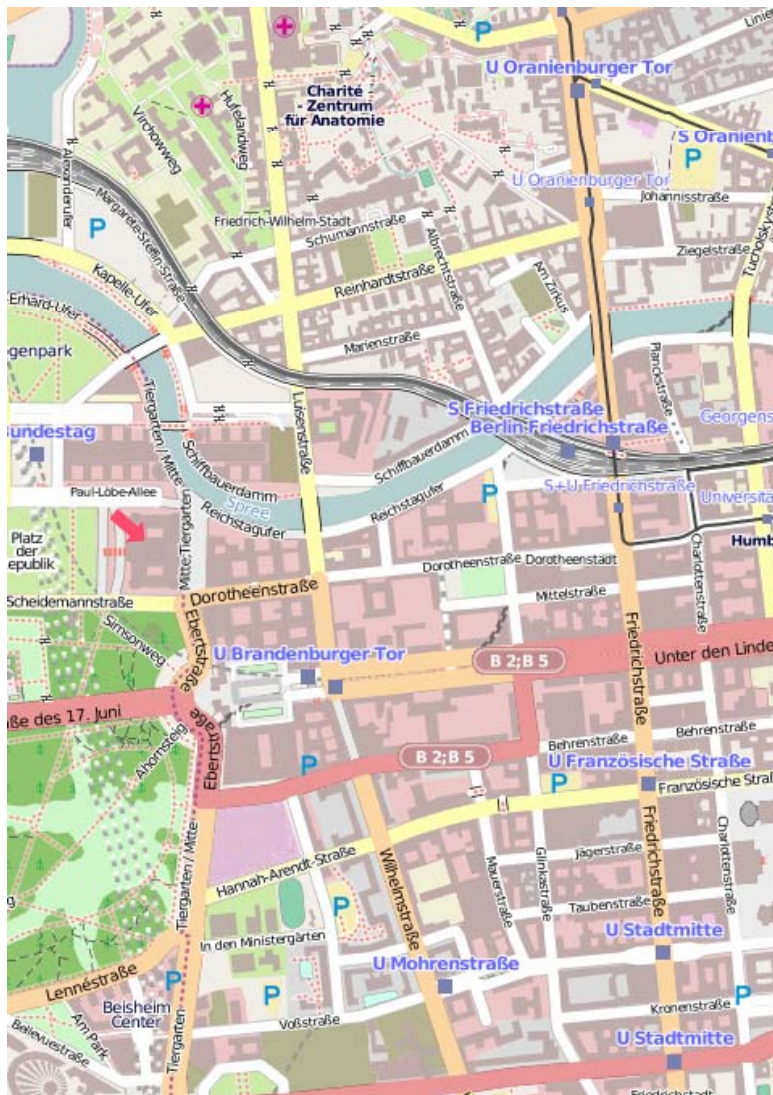
© 2011 PD Dr. Matthias Renz

4.1 Motivation

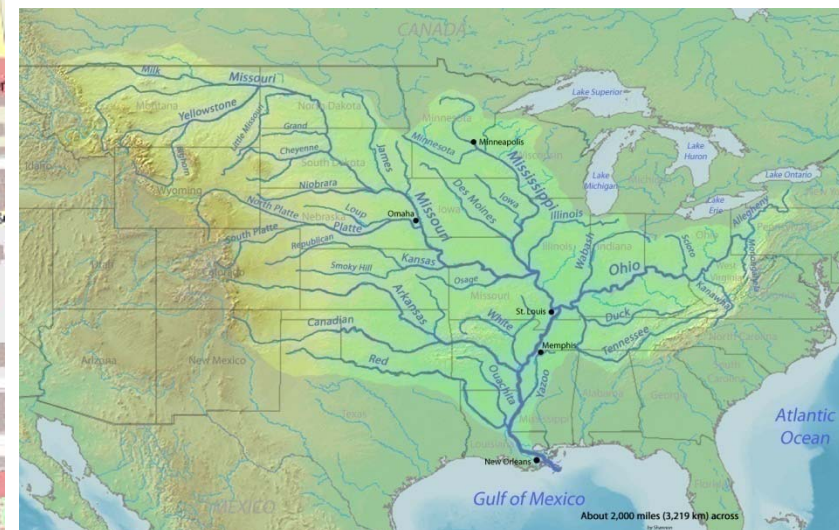
- Bisher: Distanz zwischen zwei Punkten über einfacher Distanzmaße wie z.B. die Euklidische Distanz berechnet.
- In vielen Anwendungen, z.B. im Straßenverkehr, existieren Einschränkungen bzgl. möglicher Bewegungsrichtungen.
- In Verkehrsnetzwerken wird als Distanz zwischen zwei Punkten i.d.R. die Länge des kürzesten Pfades (Netzwerkdistance) zwischen den Punkten angenommen
- (Beispiel: Distanzangaben bei der Navigation)
- Neue Herausforderung:
Berechnung der Distanz
(bzw. Nachbarschaft/Ähnlichkeit)
ist sehr teuer
=> Minimierung der
Distanzvergleiche



Straßen-, Fluss- und Schienennetze



Maximilian Dörbecker



– Beispiel-Anfragen

- Schienennetzwerke:

- “Finde die Stationen auf der ICE-Strecke von München nach Berlin.”
- “Finde alle Stationen, die direkt vom Marienplatz erreicht werden können.”
- “Finde die Linien, die Starnberg und Universität verbinden.”
- “Was ist der vorletzte Halt der U-Bahn nach Messestadt-West?”

- Flussnetzwerke:

- “Was sind die Namen aller direkten und indirekten Zuflüsse der Donau?”
- “Was sind alle direkten Zuflüsse des Rheins?”
- “Welche Gewässer wären von einem Chemieunfall in der Breg betroffen?”

- Straßennetzwerke:

- “Finde den kürzesten Pfad von der LMU zur HU Berlin.”
- “Finde das nächste Computergeschäft nach zu laufender Strecke.”
- “Finde den kürzesten Pfad, um mehrere Shops zu beliefern.”
- “Verweise Kunden auf den nächsten Kundendienst.”

4.2 Modellierung von Verkehrsnetzwerken

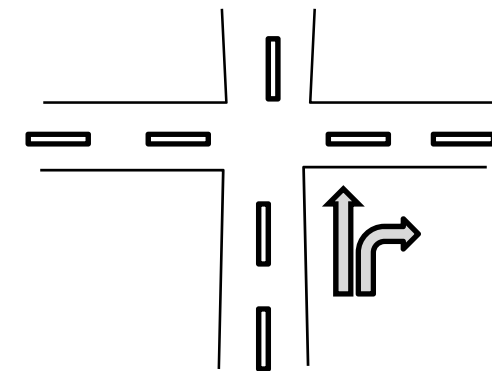
Drei-Ebenen-Modell:

1. Konzeptuelles Datenmodell: Graphen
2. Logisches Datenmodell (beschreibt das DB-Schema)
 - Datentypen
 - Graph, Vertex, Edge, Path, ...
 - Operationen (Queries)
 - is connected(..), shortest-path(), weitere Nachbarschaftsanfragen, ...
3. Physisches Datenmodell (beschreibt wie die Daten abgespeichert werden)
 - Hauptspeicherbasierte Darstellung
 - Festplattenbasierte Darstellung

4.2.1 Konzeptuelles Modell

- Straßennetze können als Graph $G=(V,E)$, bestehend aus einer Menge von Knoten V sowie einer Menge von (gerichteten) Kanten $E \subset \{(v_i, v_j) \mid v_i, v_j \in V\}$ modelliert werden
- Repräsentation 1: (üblich)
 - Knoten $v_i \in V$: Kreuzungen, Ende einer Straße, weitere relevante Punkte (z.B. Änderung der Geschwindigkeitsbeschränkung)
 - (Gerichtete) Kante $e_i \in E$: Straßenstück zwischen zwei Knoten, modelliert topologische Information
- Repräsentation 2:
 - Knoten $v_i \in V$: Straßen
 - Kanten $e_i \in E$: Kreuzungen zwischen Straßen

Wann sinnvoll?

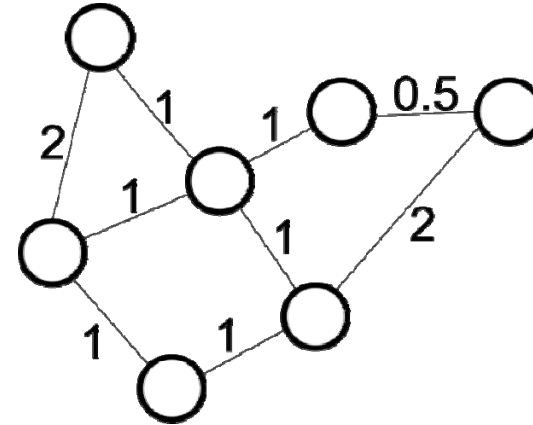


Mit Rep. 1 möglich?

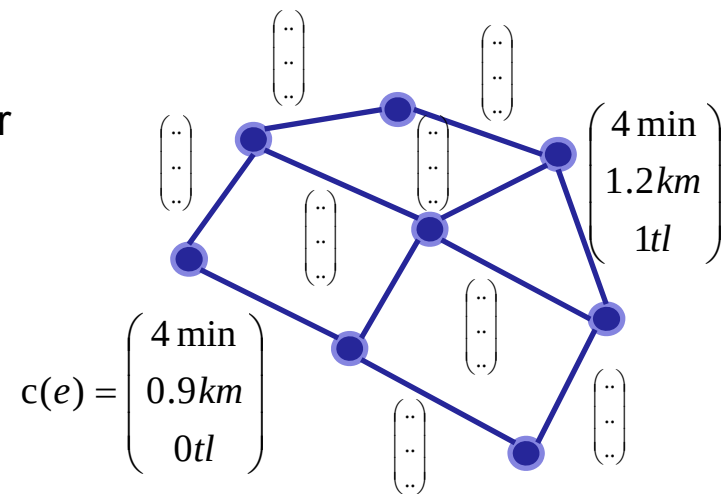
- Kanten haben Gewichte (Kosten), z. B.

- Reellwertig

- Entfernung zwischen zwei Knoten
 - Fahrtdauer
 - Benzinverbrauch
 - gewichtete Kombination mehrerer Merkmale



- Vektorielle Kombination mehrerer reellwertiger Gewichte [KRS10]



4.2.2 Logisches Datenmodell – Datentypen

– Beschreibt das Schema einer Räumlichen Netzwerk Datenbank (Spatial Network Database SNDB)

- Vertex, Attribute:
 - label
 - isVisited
 - location (räumliche Graphen)
- DirectedEdge, Attribute:
 - startNode
 - endNode
 - label
- Graph, Attribute:
 - Set<Vertex>
 - Set<DirectedEdge>
- Path: Attribute
 - Sequence<Vertex>

4.2.3 Physisches Datenmodell

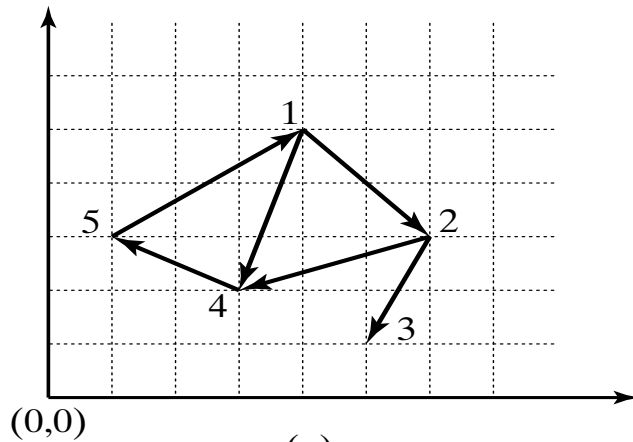
– Hauptspeicherbasiert:

- Adjazenzmatrix: $M[A, B] = 1$ gdw. $\text{edge}(\text{vertex } A, \text{vertex } B)$ existiert
- Adjazenzlisten: Bildet vertex A auf eine Liste von Nachfolgern von A ab
- Beispiele: Abbildung (a), (b) und (c) auf der nächsten Folie

– Festplattenbasiert

- Normalisiert -- Tabellen, eine für Knoten, die andere für Kanten
- Denormalisiert – Tabelle für Knoten + Adjazenzlisten
- Beispiele: Siehe Abbildung (a), (d) und (e) auf der nächsten Folie

– Beispiel



	Destination				
	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	1	0
3	0	0	0	0	0
4	0	0	0	0	1
5	1	0	0	0	0

(b) Adjacency-matrix

1	→	2	4
2	→	3	4
3	→	N	
4	→	5	
5	→	1	

(c) Adjacency-List

Node (R)

id	x	y
1	4.0	5.0
2	6.0	3.0
3	5.0	1.0
4	3.0	2.0
5	1.0	3.0

Edge (S)

source	dest	distance
1	2	$\sqrt{8}$
1	4	$\sqrt{10}$
2	3	$\sqrt{5}$
2	4	$\sqrt{10}$
4	5	$\sqrt{5}$
5	1	$\sqrt{18}$

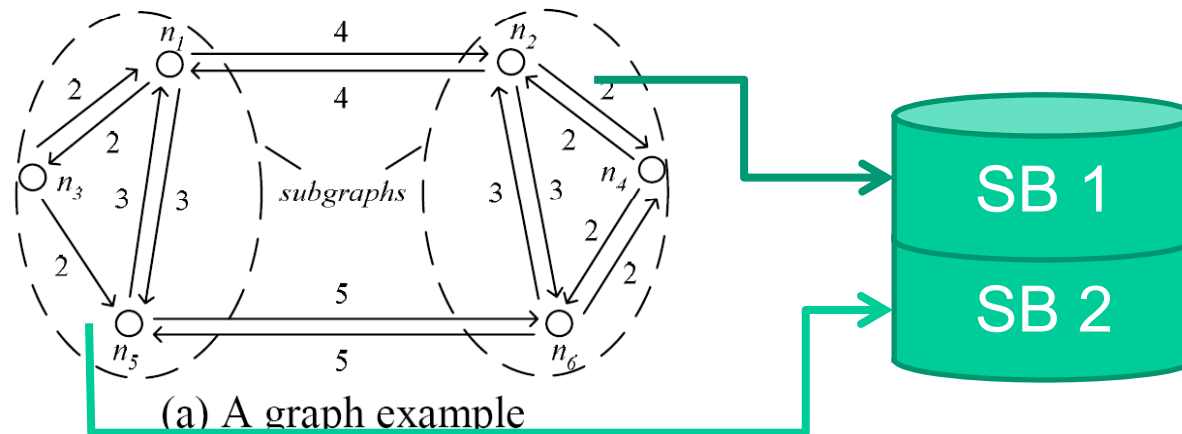
(d) Node and Edge Relations

id	x	y	Successors	Predecessors
1	4.0	5.0	(2,4)	(5)
2	6.0	3.0	(3,4)	(1)
3	5.0	1.0	()	(2)
4	3.0	2.0	(5)	(1,2)
5	1.0	3.0	(1)	(4)

(e) Denormalized Node Table

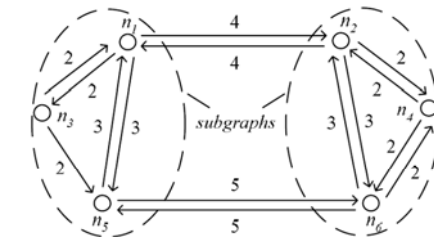
4.2.4 Effiziente Organisation des Straßen-Netzwerkgraphen

- Adjazenzlisten geeignet für Räumliche Netzwerkgraphen
- Ziel: Adaptierung der Adjazenzliste für Sekundärspeicherorganisation (Festplatte)
 - Effizienter Zugriff auf räumlich benachbarte Netzwerkelemente
 - Minimierung der I/O Kosten bei Anfragen
- Idee: Gruppiere Listen von adjazenten Knoten in gleichen Speicherblöcken (Seiten)

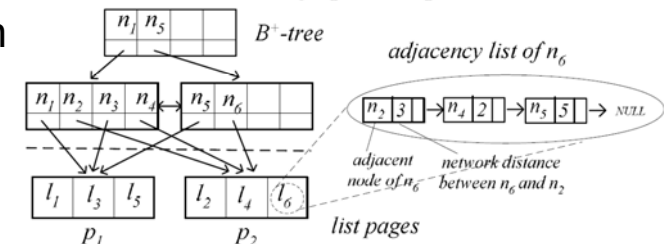


– Connectivity-Clustered Access Method (CCAM)

- Einbettung der Netzknoten in den ein-dimensionalen Raum
 - Einbettung über raumfüllende Kurve, z.B. Z-Kurve
 - Z-Kurve erhält räumliche Nachbarschaft, d.h. räumlich nah beieinander liegende Knoten im Originalraum liegen auch nahe beieinander im eingebetteten Raum
- Adjazenzlisten nah beieinanderliegender Knoten werden in einer Seite (*list page*) abgespeichert
- List pages werden über einen B+-Baum auf den entsprechenden Knoten-Ids indexiert
- Eigenschaften:
 - Unterstützt Anfragen bzgl. der topologischen Verbundenheit im Netzwerkgraph z.B. shortest path, graph traversal - Anfragen
 - (Klassische) räumliche Anfragen mit Bezug zum Euklidischen Raum nicht gut unterstützt



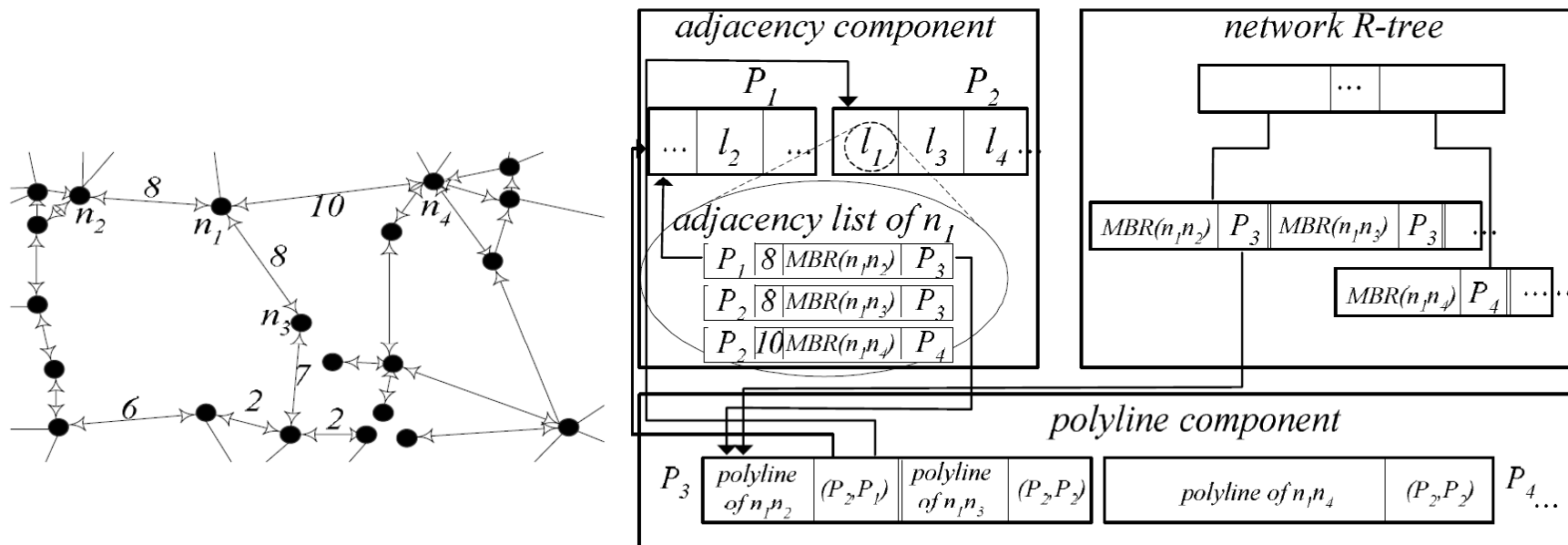
(a) A graph example



(b) The CCAM structure

– SNDB-Index-Architektur [PZMT03]

- Integriert Informationen über Raum und Verbundenheit
- Unterstützt (konventionelle) räumliche Anfragen als auch netzwerktopologische Anfragen
- 3-Komponenten-Architektur:
 - Adjazenz-Komponente (adjacency component)
 - Räumliche Komponente (network R-tree)
 - Polyline Komponente (polyline component)



- Folgende Funktionen werden im Folgenden benötigt. Sie basieren auf den vorgestellten Datenstrukturen, sollen aber nicht näher erläutert werden.
 - *check_entity(segment, point)*: gibt *true* zurück, falls *point* auf *segment* liegt
 - *find_segment(point)*: gibt das Segment zurück, auf dem *point* liegt. Liegt *point* auf mehreren Segmenten, wird das zuerst gefundene Segment zurückgegeben
 - *find_entities(segment)*: Gibt alle Punkte zurück, die auf *segment* liegen
 - *compute_ND(point1, point2)*: Berechnet Netzwerkdistanz zweier beliebiger Punkte *point1* und *point2*

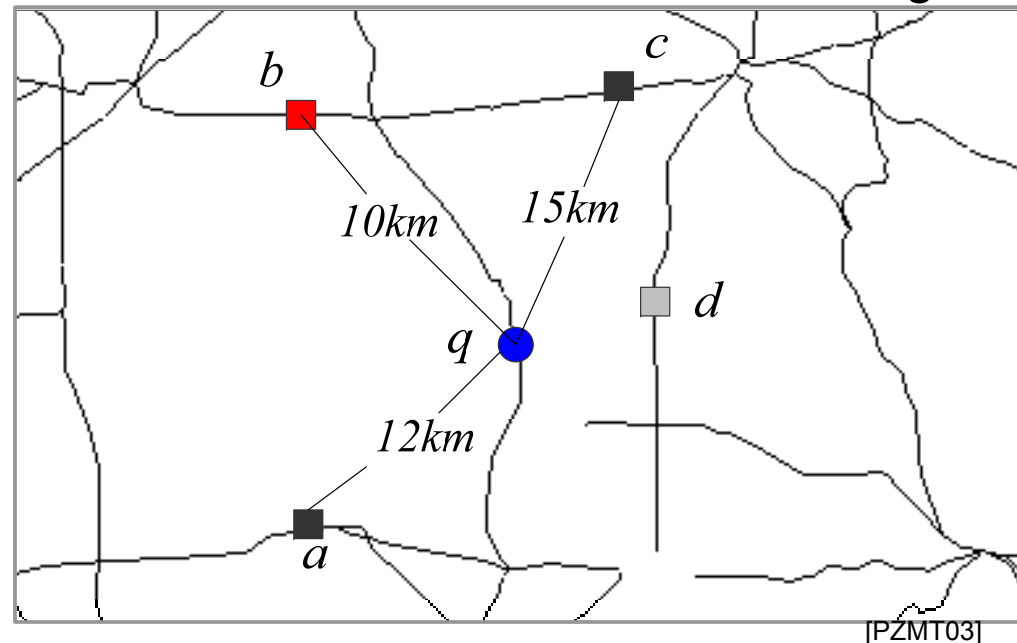
4.3 Räumliche Anfragen in Straßennetzwerken

4.3.1 Bereichsanfragen

- Ziel: Bestimme alle Objekte in einem bestimmten Umkreis unter Beachtung des darunterliegenden Netzwerkes:

$$Rq(q,r,DB) = \{x \in DB \mid \text{dist}_{\text{net}}(q,x) < r\}$$

Beispiel: „Finde alle Tankstellen in einer Umgebung von 15



– *Bereichsanfrage über Range Euclidean Restriction (RER)* [PZMT03]

- Bereichsanfrage auf DB bzgl. der **euklidischen Distanz** aller Objekte zu q
- Weil $\text{dist}_{\text{net}}(q,p) \geq \text{dist}_{L_2}(q,p)$ kommt es nicht zu *false misses*
- Es können aber sehr viele *false hits* entstehen
- Eliminierung der *false hits* über einmalige Network Expansion, um Kosten zu sparen

– Vorgehen

1. Suche Menge S' aller Punkte, die höchstens die Euklidische Anfragedistanz e zum Anfragepunkt Q haben
2. Beginne eine Traversierung des Straßennetzes bei q entsprechend Dijkstra
3. Teste für jede mit dem Dijkstra-Algorithmus besuchte Kante k , ob ein Punkt p aus S' auf der Kante liegt.

Filter:

Teste ob p in $\text{MBR}(k)$ liegt,

Beschleunigung über Sortierung von S' bzgl. einer Dimension

Refinement:

Teste ob p auf k liegt.

– Durch dieses Vorgehen muss der Netzwerkgraph nur einmal durchlaufen werden

- Algorithmus

RER(q, e):

result = $\emptyset$

$S' = \text{Euclidean-range}(q, e)$

$n_i, n_j = \text{find_segment}(q)$

$Q = \langle (n_i, d_N(q, n_i)), (n_j, d_N(q, n_j)) \rangle$

De-queue node n in Q with smallest $d_N(q, n)$

while $d_N(q, n) \leq e$ and $S' \neq \emptyset$

for each non-visited adjacent node n_x of n

for each point s in S'

if $\text{check_entity}(n_x, n, s)$

result = result $\cup \{s\}$; $S' = S' - \{s\}$

en-queue $(n_x, d_N(q, n_x))$

de-queue the next node n in Q

end while

// S' gibt alle Kandidaten mit euklidischer Distanz e zu q zurück

// Hole das Segment des Netzwerkgraphen, auf dem q liegt

// Q : Prioritätswarteschlange, sortiert nach Netzwerkdistanz zu q

// hole erstes Segment aus der Prioritätswarteschlange

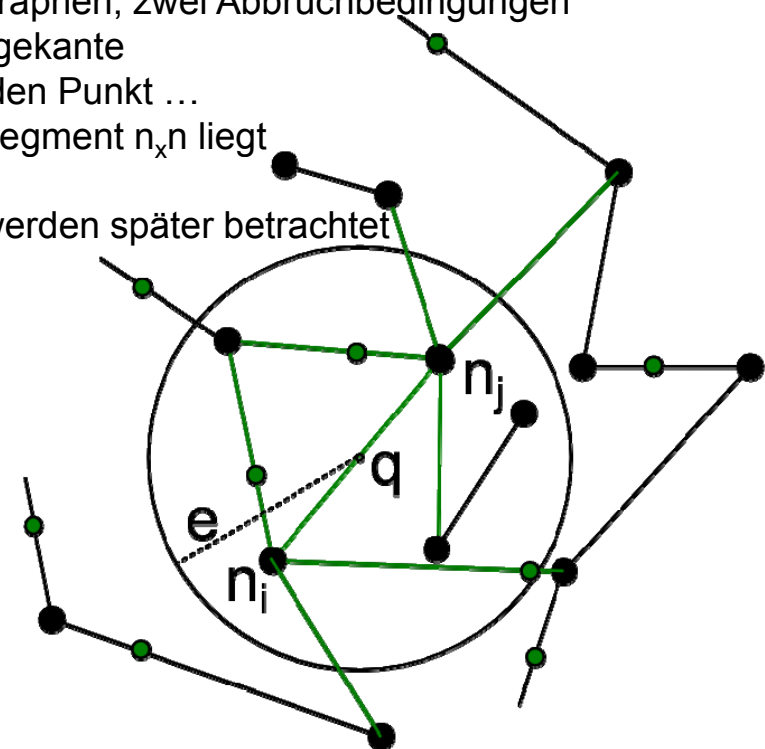
// Traversiere den Graphen, zwei Abbruchbedingungen

// Hole eine Nachfolgekante

// ... und teste für jeden Punkt ...

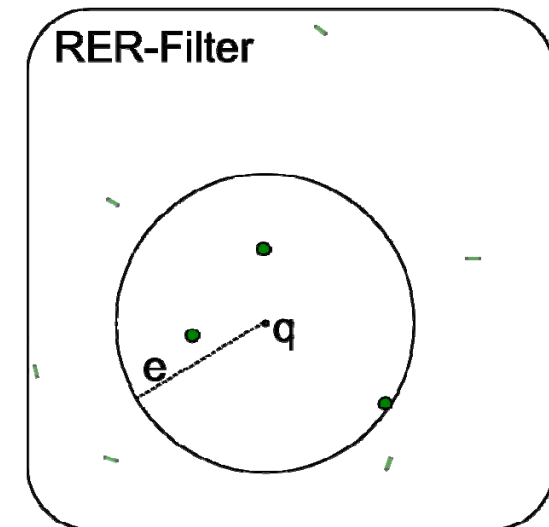
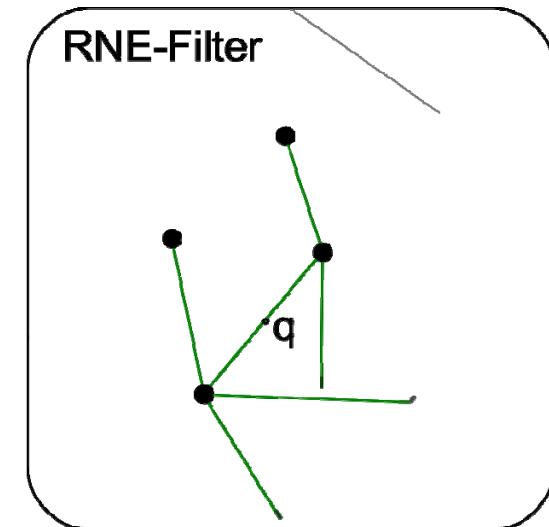
// ... ob er auf dem Segment $n_x n$ liegt

// die Kinder von n_x werden später betrachtet



– *Range Network Expansion (RNE)* [PZMT03]

- Berechne Kandidatensegmente QS mit Entfernung e zur Anfrage q (Entfernung mittels Netzwerkdistanz).
- Ergebnisse werden anhand dieser Kandidatensegmente ermittelt
- Damit können viele Anfragen gleichzeitig über den die Objekte verwaltenden R-Baum bearbeitet werden (Intersection Join): In jeden Teilbaum der Punktmenge, der mindestens ein Segment schneidet, wird abgestiegen
- Beim rekursiven Aufruf eines Teilastes werden nicht alle Elemente aus QS an den Teilbaum übergeben, sondern nur schneidende.
- Zuletzt müssen Duplikate entfernt werden, die direkt auf einem Knoten lagen und zweimal zurückgegeben wurden



– Algorithmus

RNE(node_id, QS, result):

if (node_id is an intermediate node)

 compute QS_i for each entry E_i in node_id //join

 for each entry E_i in node_id

 if($QS_i \neq \emptyset$)

 RNE(E_i .node_id, QS_i , result)

Else

$result_{node_id} = \text{plane-sweep}(node_id.entries, QS_i)$

 sort $result_{node_id}$ to remove duplicates

$result = result \cup result_{node_id}$

//QS: Ergebnis einer Bereichsanfrage auf die

//Segmente unter Verwendung der Netzwerkdistanz

//Falls aktueller Knoten des Punkt-Index ein innerer

//Knoten ist ...

//... berechne für jedes Kind die Segmente, die das Kind

//schneiden ...

//... und rufe die Funktion rekursiv auf, wenn die

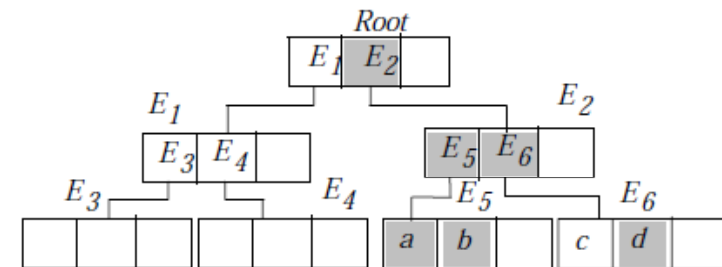
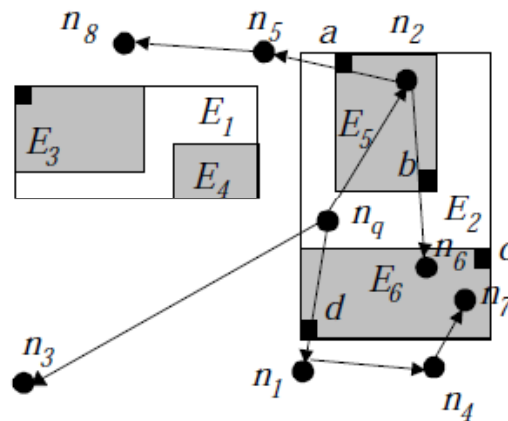
//Schnittmenge nicht leer ist

//Falls der Knoten ein Blatt ist, berechne über einen

//Plane-Sweep-Algorithmus alle Punkte, die auf das

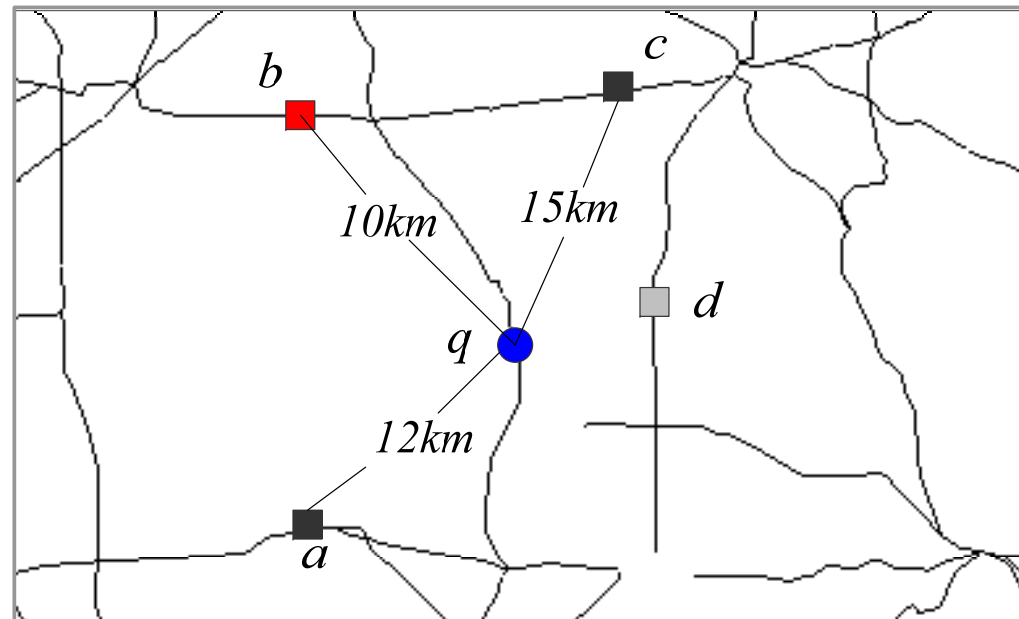
//Segment fallen

//Und entferne ggf. Duplikate



4.3.2 k-Nächste-Nachbarn-Anfragen

- Ziel: Bestimme die k-nächsten Nachbarn bzgl. der zurückzulegenden Strecke.
- Beispiel: „Finde das mir am nächsten gelegene Hotel“
- Entspricht nicht unbedingt der euklidischen Distanz - im Beispiel hat d die kürzeste euklidische Distanz, aber eine sehr hohe Netzwerkdistanz.



[PZMT03]

Berechnung der k-NN-Anfrage über *Incremental Euclidean Restriction (IER)* [PZMT03]

1. Berechne k-NN Kandidaten resultCandidates = $\{p_{E1}, \dots, p_{Ek}\}$ bzgl. euklidischer Distanz d_E über Standardalgorithmus

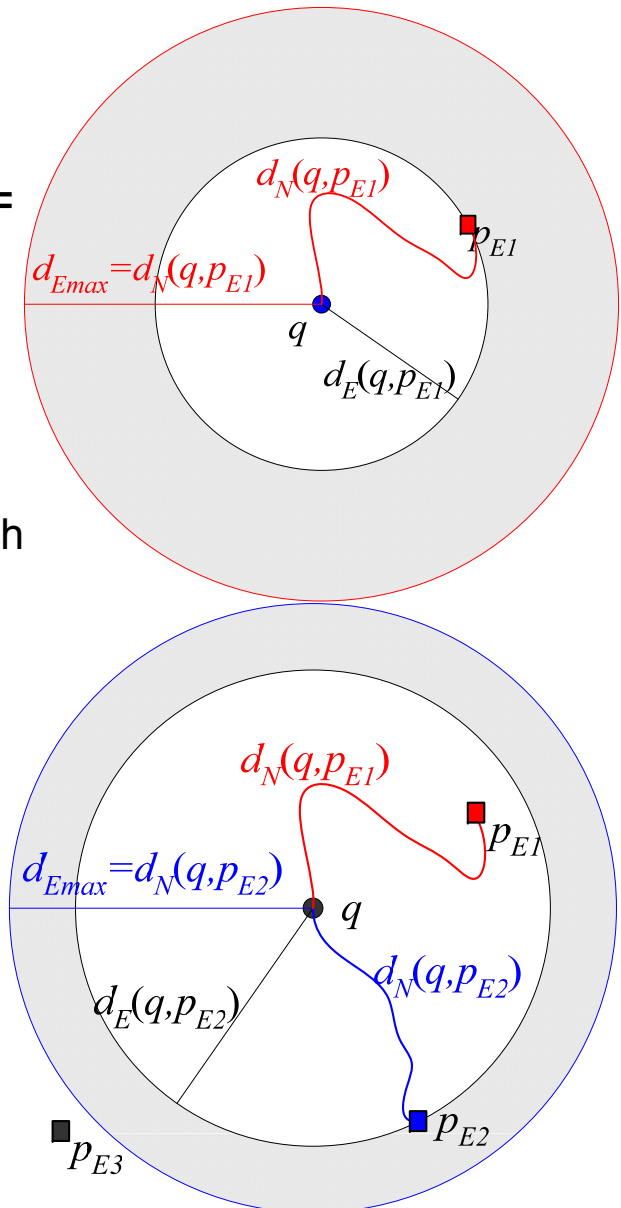
2. $\forall p_{Ei}$ berechne Netzwerkdistanz $\text{dist}_{\text{net}}(q, p_{Ei})$

- $d_{\text{Emax}} = \max_{i=1..k} \text{dist}_{\text{net}}(q, p_{Ei})$
- Sortiere Kand. in resultCandidates aufsteigend nach dist_{net}
- Da $\text{dist}_{\text{net}}(q, p) \geq \text{dist}_{L_2}(q, p)$ haben alle möglichen Kandidaten c eine Distanz $\text{dist}_{L_2}(q, c) \leq d_{\text{Emax}}$ (Pruningkriterium)

3. Suche nächsten Kand. p_{Ei} bzgl. $d_E \leq d_{\text{Emax}}$:

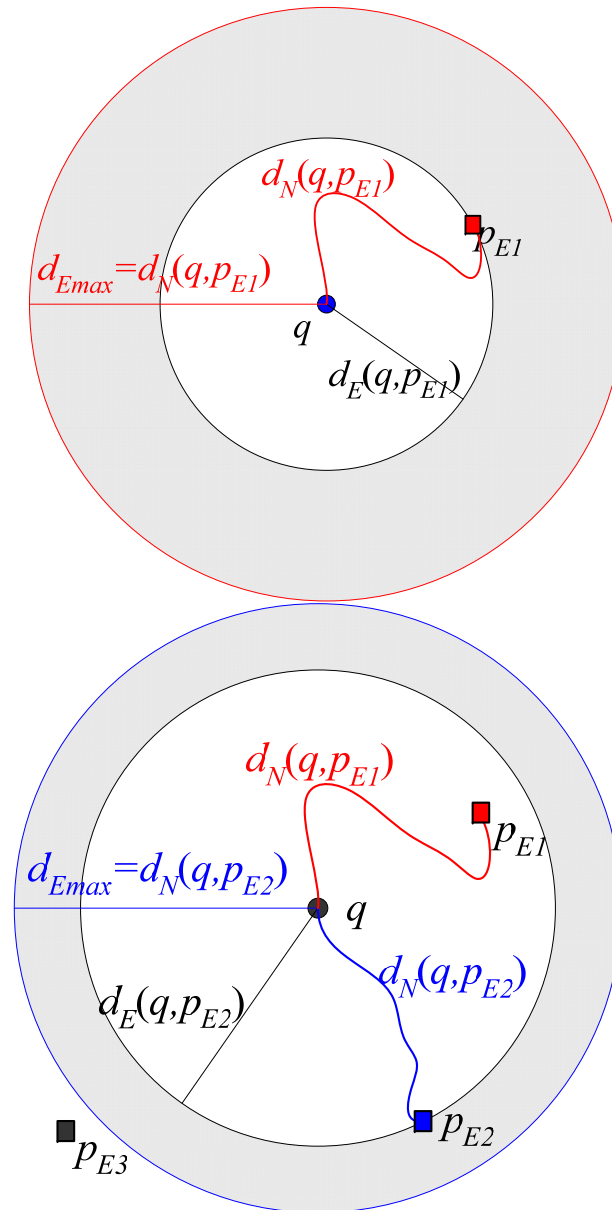
- Falls $\text{dist}_{\text{net}}(q, p_{Ei}) < d_{\text{Emax}}$
ersetze p_{Ek} mit p_{Ei} in resultCandidates
 $d_{\text{Emax}} = \text{dist}_{\text{net}}(q, p_{Ei})$
- Wiederhole Schritt 3 bis $d_E > d_{\text{Emax}}$

Funktioniert gut, wenn NN-Ranking bzgl. eukl. Distanz und Netzwerkdistanz ähnlich sind



Algorithm IER (q, k)/* q is the query point */

1. $\{p_1, \dots, p_k\} = \text{Euclidean_NN}(q, k)$;
2. for each entity p_i
3. $d_N(q, p_i) = \text{compute_ND}(q, p_i)$
4. sort $\{p_1, \dots, p_k\}$ in ascending order of $d_N(q, p_i)$
5. $d_{E_{\max}} = d_N(q, p_k)$
6. repeat
7. $(p, d_E(q, p)) = \text{next_Euclidean_NN}(q)$;
8. if $(d_N(q, p) < d_N(q, p_k))$ // p closer than the k^{th} NN
9. insert p in $\{p_1, \dots, p_k\}$ // remove ex- k^{th} NN
10. $d_{E_{\max}} = d_N(q, p_k)$
11. until $d_E(q, p) > d_{E_{\max}}$

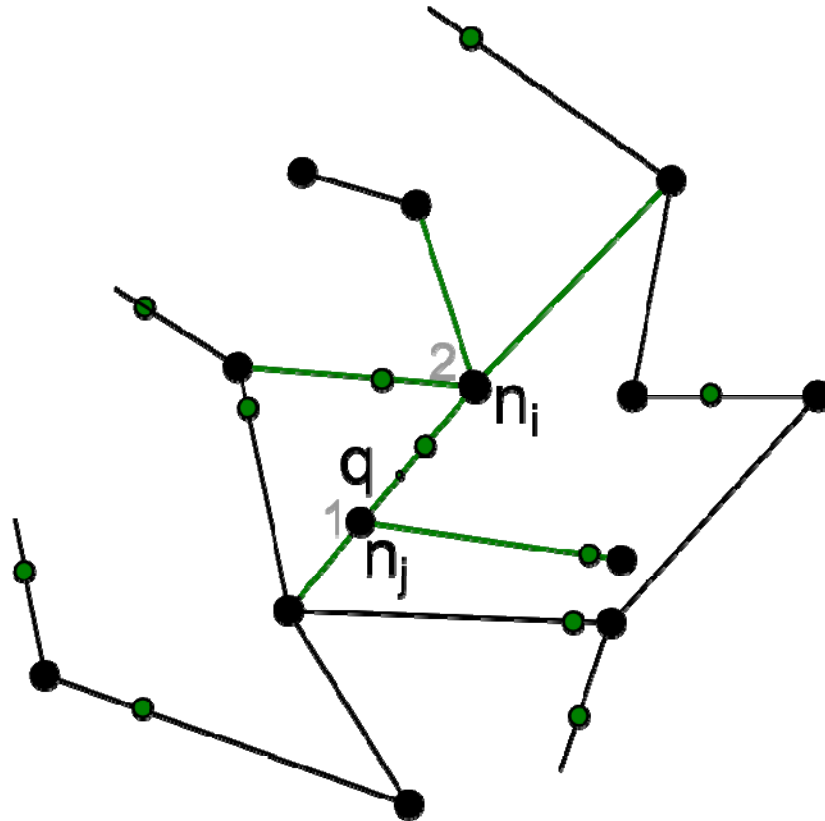
End IER**Figure 4.2:** Incremental Euclidean Restriction

Berechnung der k-NN-Anfrage

über *Incremental Network Expansion* (INE) [PZMT03]

- Sehr ähnlich zum Dijkstra-Algorithmus
- Das Netzwerk wird entsprechend des Dijkstra-Algorithmus durchsucht, bis die ersten k-NN-Kandidaten gefunden werden. Eine Variable speichert die aktuelle k-NN-Distanz.
- Werden weitere k-NN-Kandidaten gefunden, wird sowohl die k-NN-Distanz als auch der aktuelle k-NN-Kandidat ggf. aktualisiert
- Der Algorithmus terminiert, wenn alle Knoten in der Warteschlange eine größere Netzwerkdistanz zur Anfrage haben als die gespeicherte NN-Distanz erlaubt.

Algorithmus zur k-NN-Suche über INE_[PZMT03]



Algorithm INE (q, k)

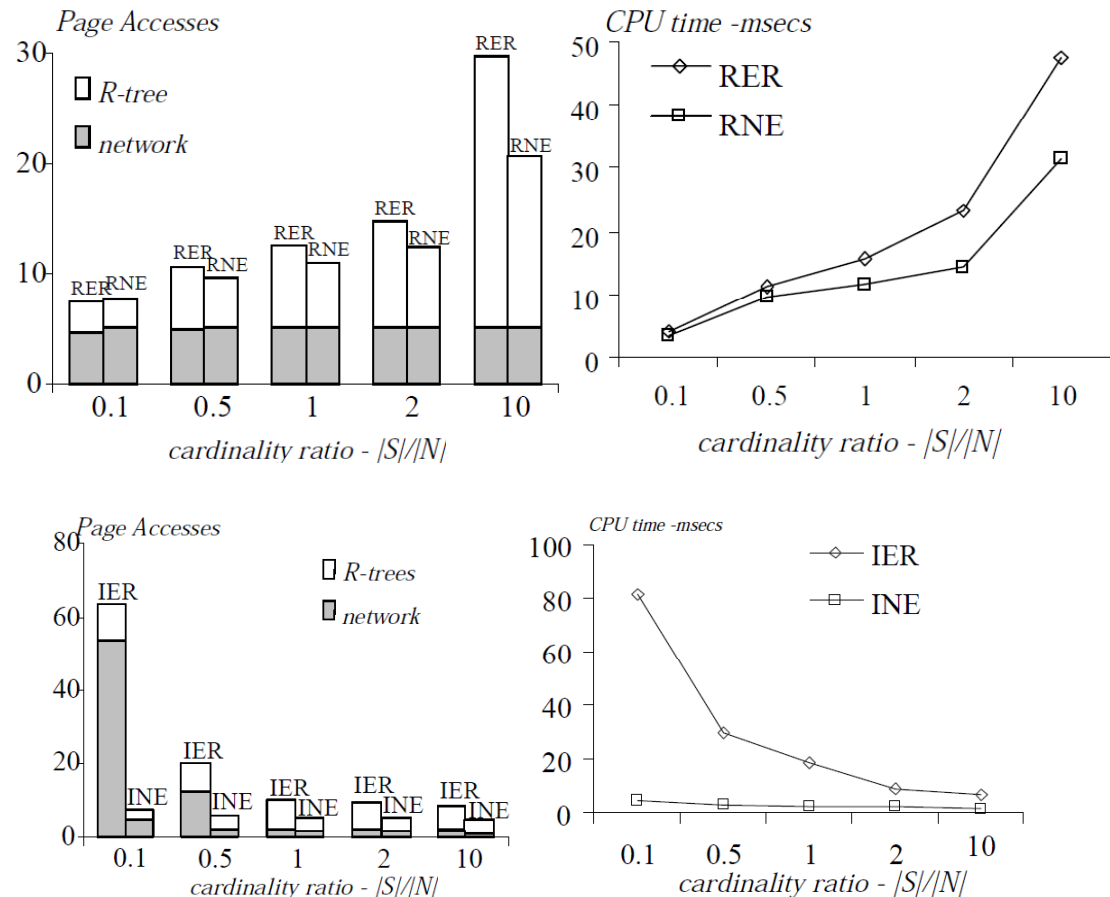
1. $n_i n_j = \text{find_segment}(q)$
2. $S_{\text{cover}} = \text{find_entities}(n_i n_j)$; // S_{cover} is the set of entities covered by $n_i n_j$
3. $\{p_1, \dots, p_k\}$ = the k (network) nearest entities in S_{cover} sorted in ascending order of their network distance ($p_m, p_{m+1}, \dots, p_k$ may be $\emptyset$ if S_{cover} contains $< k$ points)
4. $d_{N_{\max}} = d_N(q, p_k)$ // if $p_k = \emptyset$, $d_{N_{\max}} = \infty$
5. $Q = \langle (n_i, d_N(q, n_i)), (n_j, d_N(q, n_j)) \rangle$ // sorted on d_N
6. de-queue the node n in Q with the smallest $d_N(q, n)$
7. while ($d_N(q, n) < d_{N_{\max}}$)
8. for each non-visited adjacent node n_x of n
9. $S_{\text{cover}} = \text{find_entities}(n_x n)$;
10. update $\{p_1, \dots, p_k\}$ from $\{p_1, \dots, p_k\} \cup S_{\text{cover}}$
11. $d_{N_{\max}} = d_N(q, p_k)$
12. en-queue $(n_x, d_N(q, n_x))$
13. de-queue the next node n in Q

End INE

Figure 4.4: Incremental Network Expansion

• Performanz von RER, RNE, IER, INE

- Geprüft wurden
 - Seitenzugriffe (HDD)
 - CPU-Zeit
 - S = Menge aller Objekte
 - N = Menge aller Netzwerksegmente



- Network Expansion meist besser als Euclidean Restriction