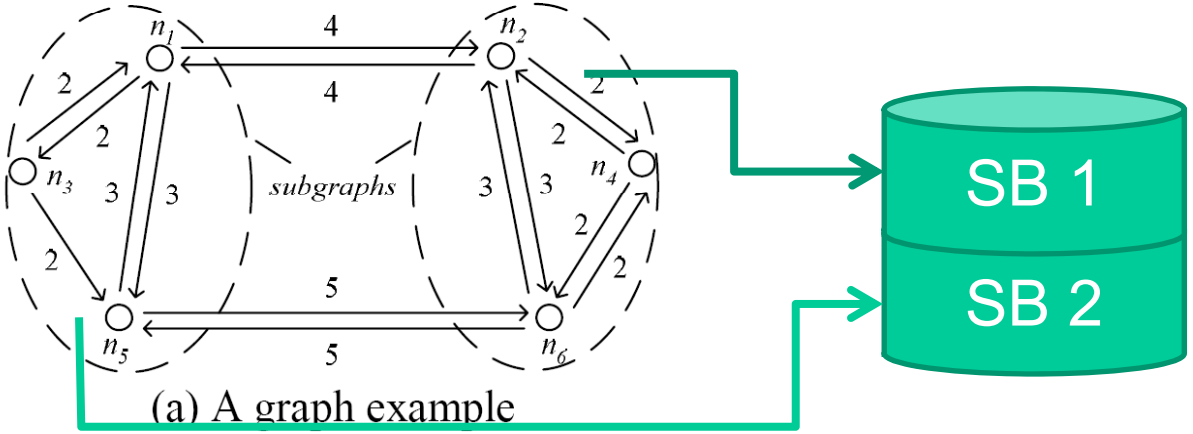
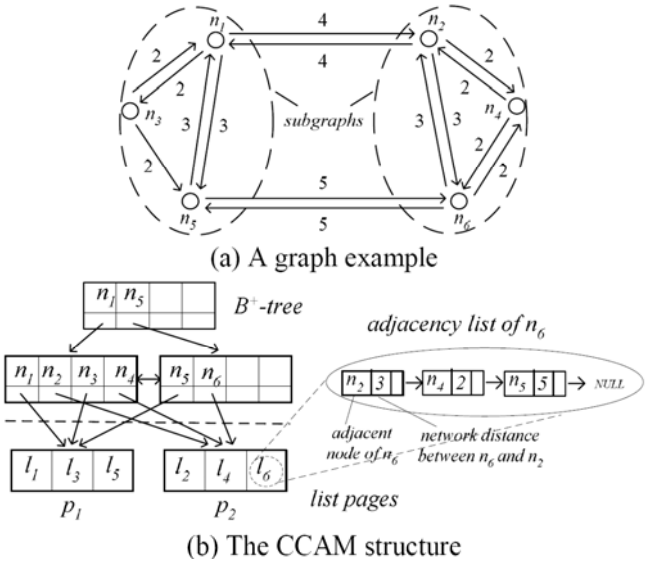
4.3.3 Effiziente Organisation des Straßen-Netzwerkgraphen

- Adjazenzlisten geeignet für Räumliche Netwerkgraphen
- 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)

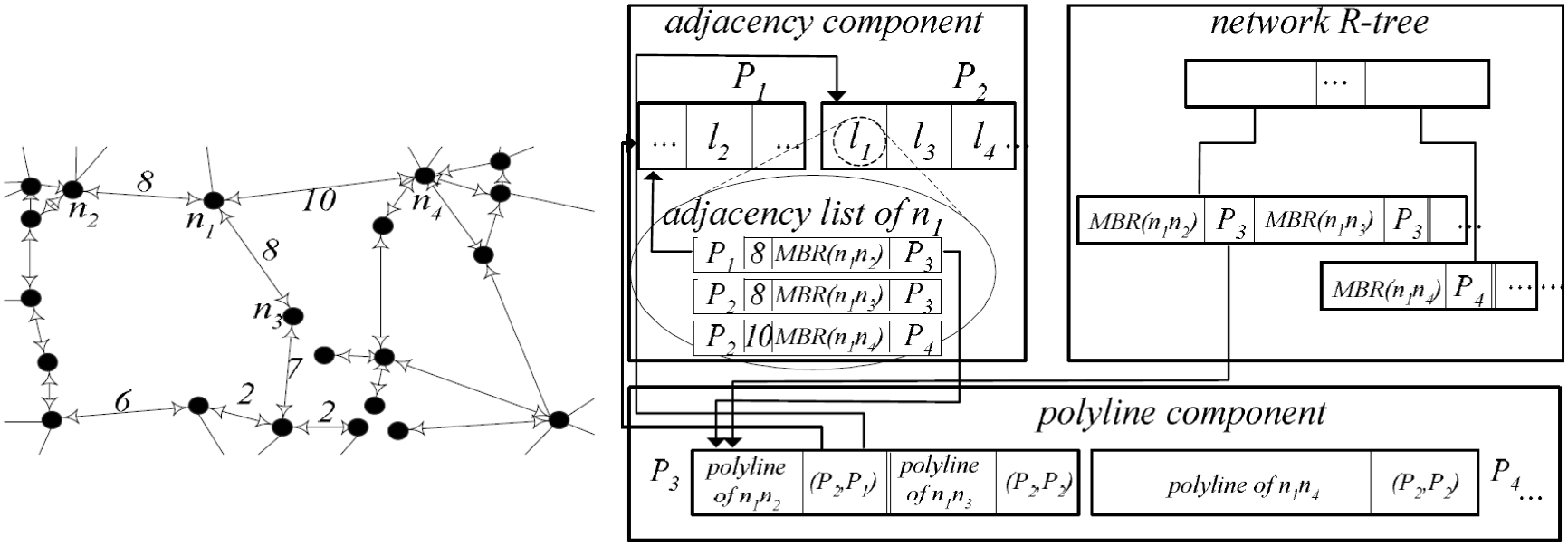


– Connectivity-Clustered Access Method (CCAM)

- Einbettung der Netzwerkknoten 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



– 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*

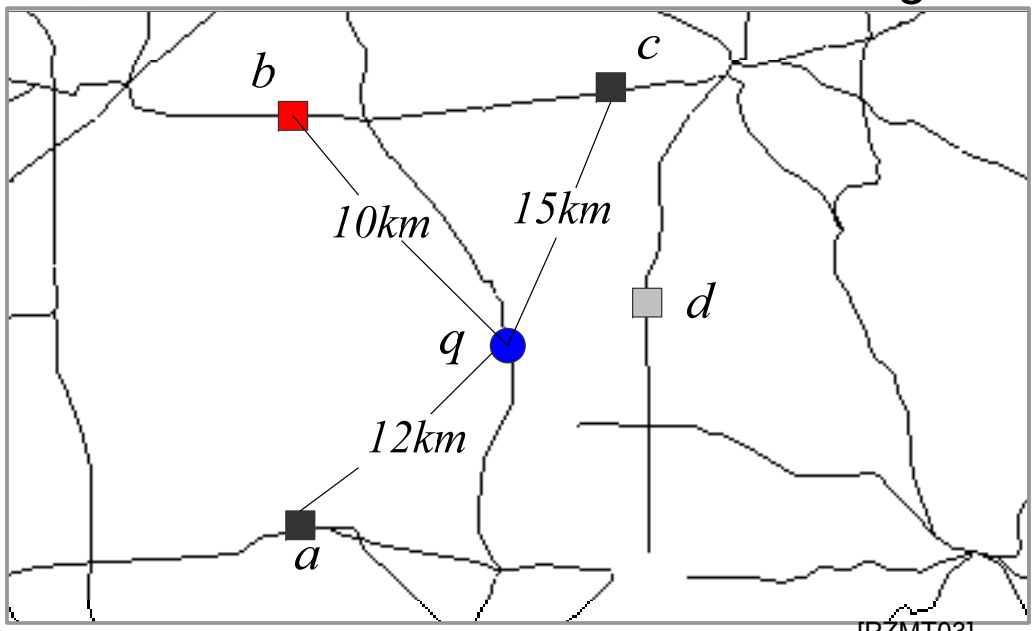
4.4 Räumliche Anfragen in Straßennetzwerken

4.4.1 Bereichsanfragen

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

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

Beispiel: „Finde alle Tankstellen in einer Umgebung von 15



[PZMT03]

– *Bereichsanfrage über Range Euclidean Restriction*[PZMT03]
  - Bereichsanfrage auf DB bzgl. der euklidischen Distanz aller Objekte (**Punkte**) zu q
  - Weil $dist_{net}(q,p) \geq dist_{L_2}(q,p)$ 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 MBR(k) liegt,

     Beschleunigung über Sortierung von S‘ bzgl. einer Dimemsion

     Refinement:

     Teste ob p auf k liegt.

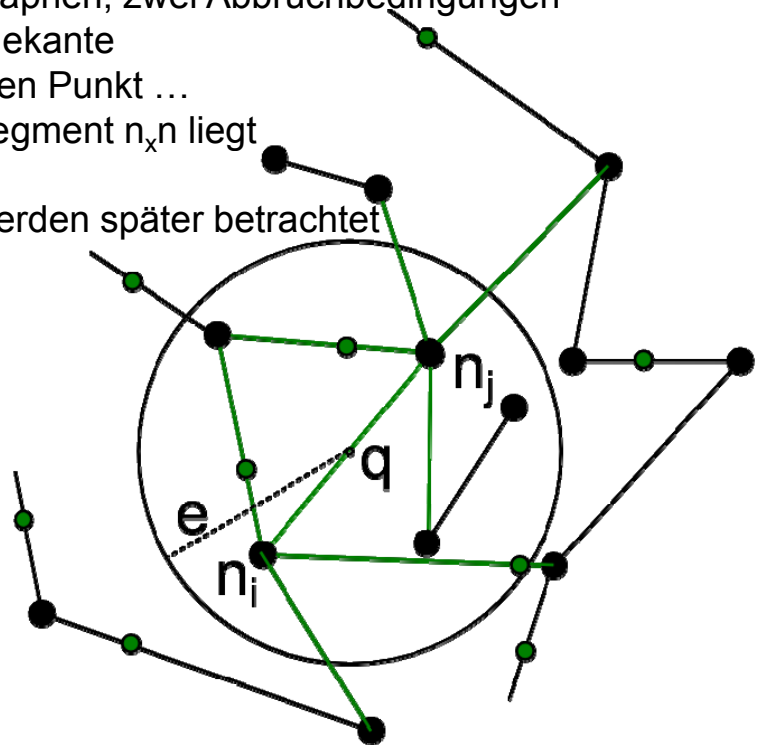
– Durch dieses Vorgehen muss der Netzwerkgraph nur einmal durchlaufen werden

- Algorithmus

RER(q, e):

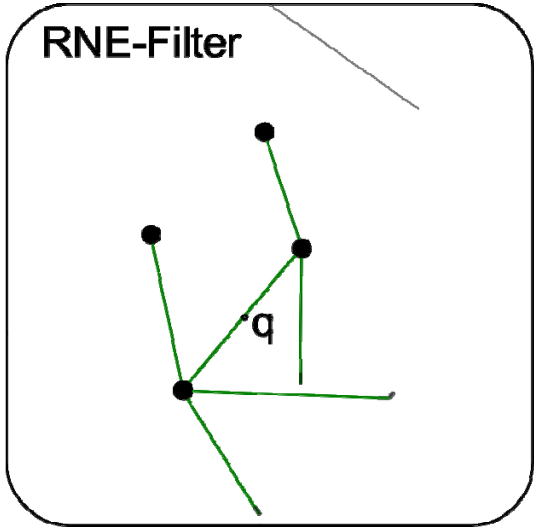
result = Ø
S‘ =Euclidean-range(q,e) //S‘ gibt alle Kandidaten mit euklidischer Distanz e zu q zurück
$n_i n_j$=find_segment(q) //Hole das Segment des Netzwerkgraphen, auf dem q liegt
Q=<(n_i,$d_N(q,n_i)$), (n_j,$d_N(q,n_j)$)> //Q: Prioritätswarteschlange, sortiert nach Netzwerkdistanz zu q

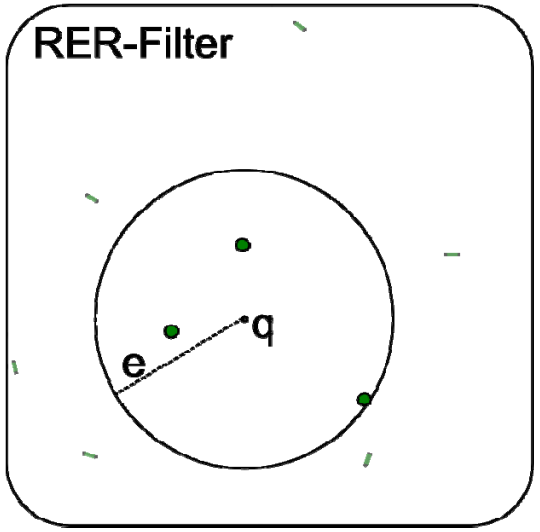
De-queue node n in Q with smallest $d_N(q,n)$ //hole erstes Segment aus der Prioritätswarteschlange
while $d_N(q,n) \leq e$ and S‘ ≠ Ø //Traversiere den Graphen, zwei Abbruchbedingungen
    for each non-visited adjacent node n_x of n //Hole eine Nachfolgekante
        for each point s in S‘ //… und teste für jeden Punkt …
            if check_entity(n_xn,s) //… ob er auf dem Segment n_xn liegt
                result = result ∪ {s}; S‘=S‘-{s}
        en-queue (n_x,$d_N(q,n_x)$) //die Kinder von n_x werden später betrachtet
    de-queue the next node n in Q
end while



– *Range Network Expansion* [PZMT03]

- Berechne Kandidaten**segmente** QS mit Entfernung e zur Anfrage q.
- 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): //QS: Ergebnis einer Bereichsanfrage auf die //Segmente mit Radius e

if (node_id is an intermediate node) //Falls aktueller Knoten des Punkt-Index ein innerer //Knoten ist …

compute QS_i for each entry E_i in node_id //join //… berechne für jedes Kind die Segmente, die das Kind //schneiden …

for each entry E_i in node_id //… und rufe die Funktion rekursiv auf, wenn die //Schnittmenge nicht leer ist

if($QS_i \neq \emptyset$)

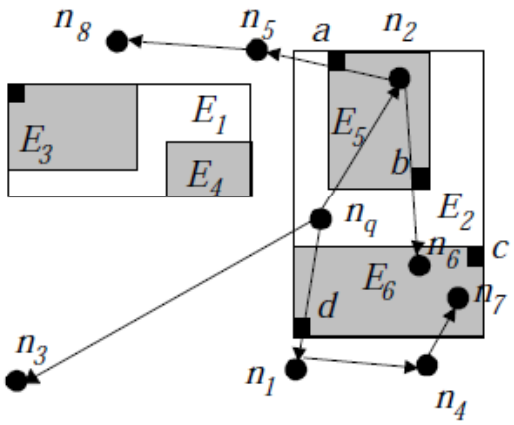
RNE(E_i.node_id, QS_i, result)

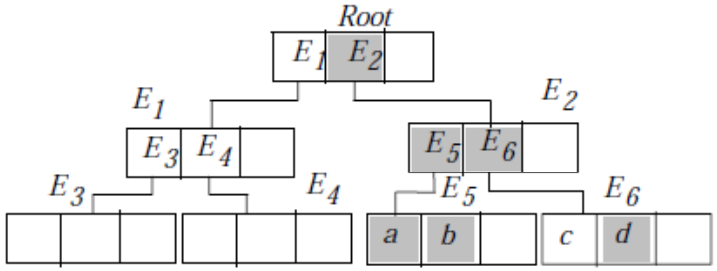
Else //Falls der Knoten ein Blatt ist, berechne über einen

$result_{node\_id}$=plane-sweep(node_id.entries, QS_i) // Plane-Sweep-Algorithmus alle Punkte, die auf das //Segment fallen

sort $result_{node\_id}$ to remove duplicates //Und entferne ggf. Duplikate

result = result ∪ $result_{node\_id}$



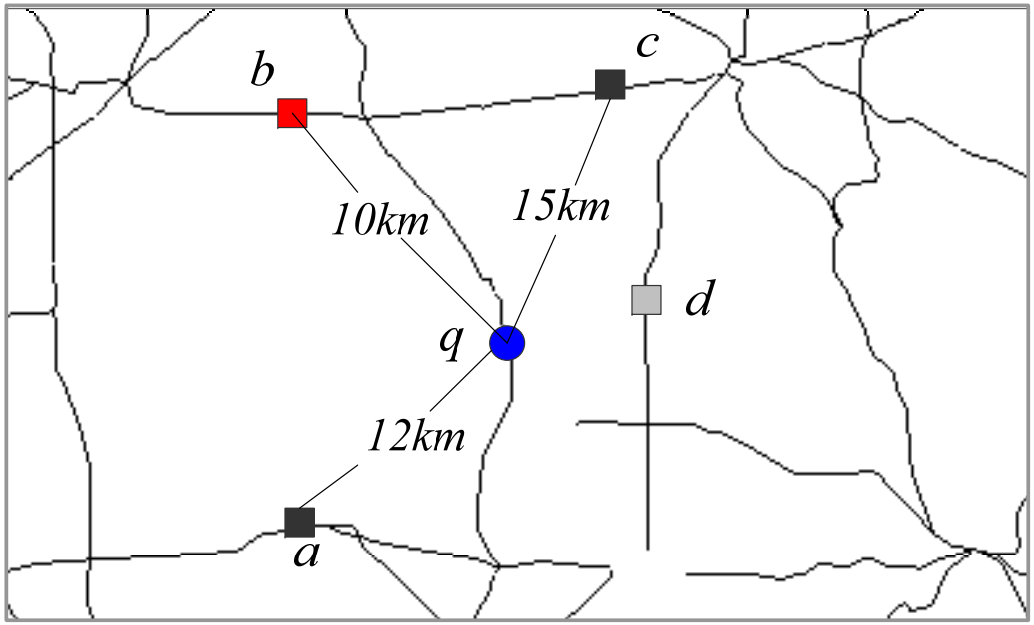


4.4.2 Nächste-Nachbarn-Anfragen

• Ziel: Bestimme den 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.

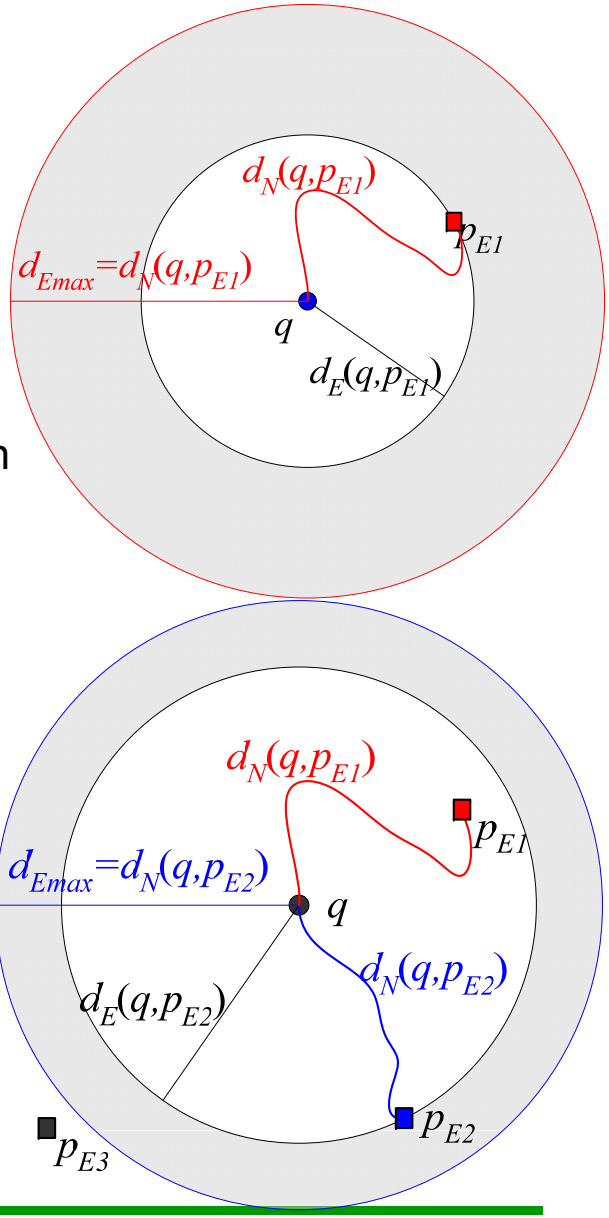


[PZMT03]

NN über Incremental Euclidean Restriction [PZMT03]

1. Berechne k-NN Kandidaten resultCandidates = $\{p_{E1},..,p_{Ek}\}$ bzgl. euklidischer Distanz d_E über Standardalgorithmus
2. $\forall p_{Ei}$ berechne Netzwerkdistanz $dist_{net}(q, p_{Ei})$
   - $d_{Emax} = max_{i=1..k}\ dist_{net}(q, p_{Ei})$
   - Sortiere Kand. in resultCandidates aufsteigend nach $dist_{net}$
   - Da $dist_{net}(q, p) \geq dist_{L_2}(q,p)$ haben alle möglichen Kandidaten c eine Distanz $dist_{L_2}(q,c) \leq d_{Emax}$ (Pruningkriterium)
3. Suche nächsten Kand. p_{Ei} bzgl. d_E: $\leq d_{Emax}$:
   - Falls $dist_{net}(q, p_{Ei}) < d_{Emax}$
     ersetze p_{Ek} mit p_{Ei} in resultCandidates
     $d_{Emax} = dist_{net}(q, p_{Ei})$
   - Wiederhole Schritt 3 bis $d_E > d_{Emax}$

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



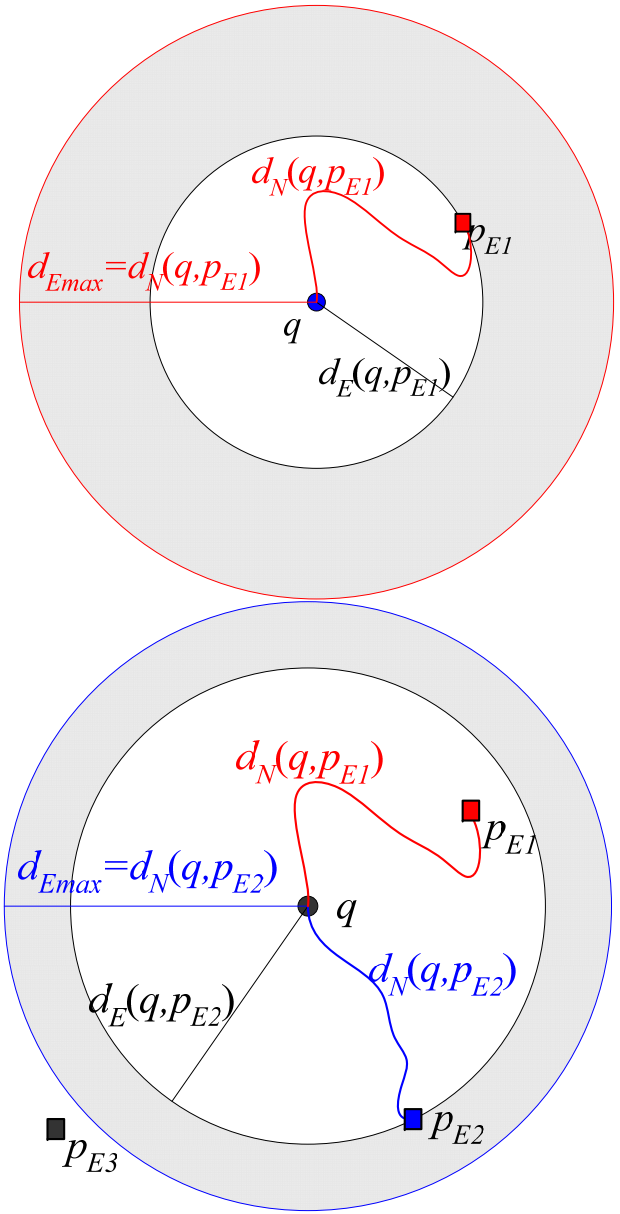
Algorithm IER (q, k)

/* q is the query point */

1. $\{p_1,...,p_k\}$=Euclidean_NN(q,k);
2. for each entity p_i
3. $d_N(q,p_i)$=*compute_ND*(q,p_i)
4. sort $\{p_1,...,p_k\}$ in ascending order of $d_N(q,p_i)$
5. d_{Emax}= $d_N(q, p_k)$
6. repeat
7. $(p,d_E(q, p))$=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 $\{p_1,...,p_k\}$ // remove ex-k^{th} NN
10. d_{Emax} = $d_N(q, p_k)$
11. until $d_E(q,p)>d_{Emax}$

End IER

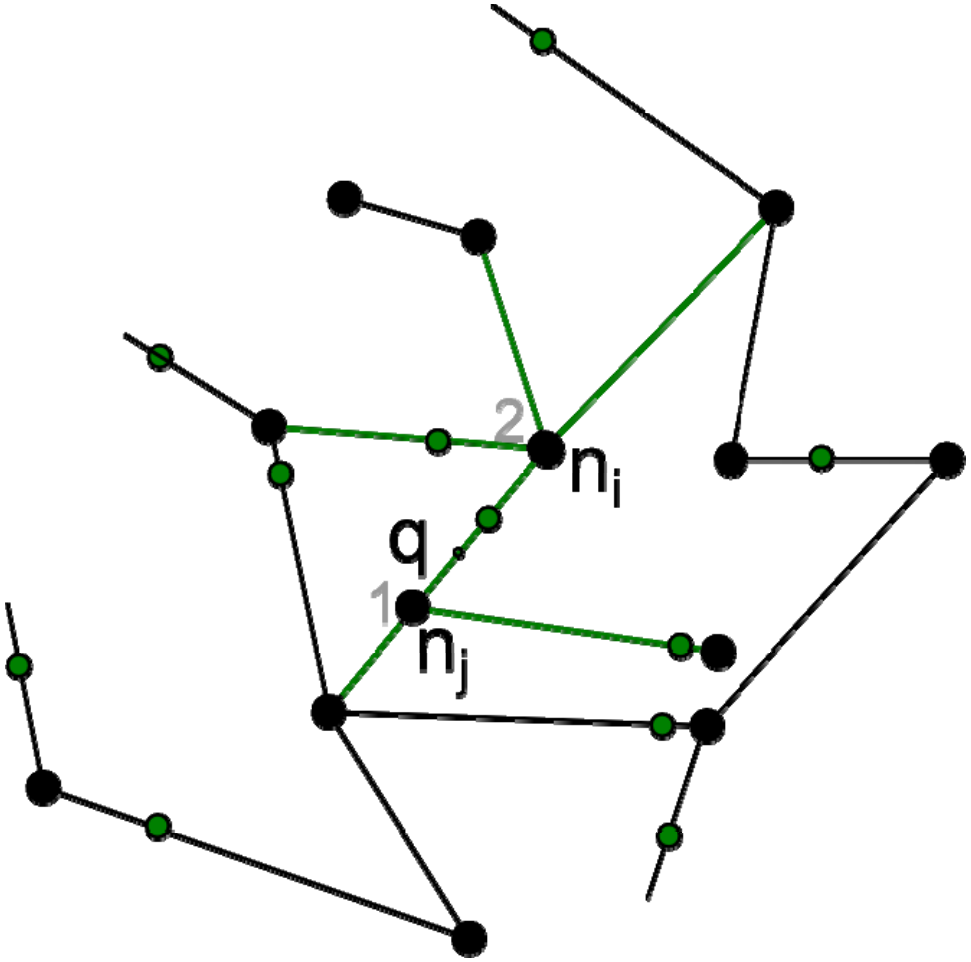
Figure 4.2: Incremental Euclidean Restriction



kNN über Incremental Network Expansion [PZMT03]

- Sehr ähnlich zum Dijkstra-Algorithmus
- Das Netzwerk wird entsprechend des Dijkstra-Algorithmus durchsucht, bis die ersten kNN-Kandidaten gefunden werden. Eine Variable speichert die aktuelle kNN-Distanz.
- Werden weitere kNN-Kandidaten gefunden, wird sowohl die kNN-Distanz als auch der aktuelle kNN-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)

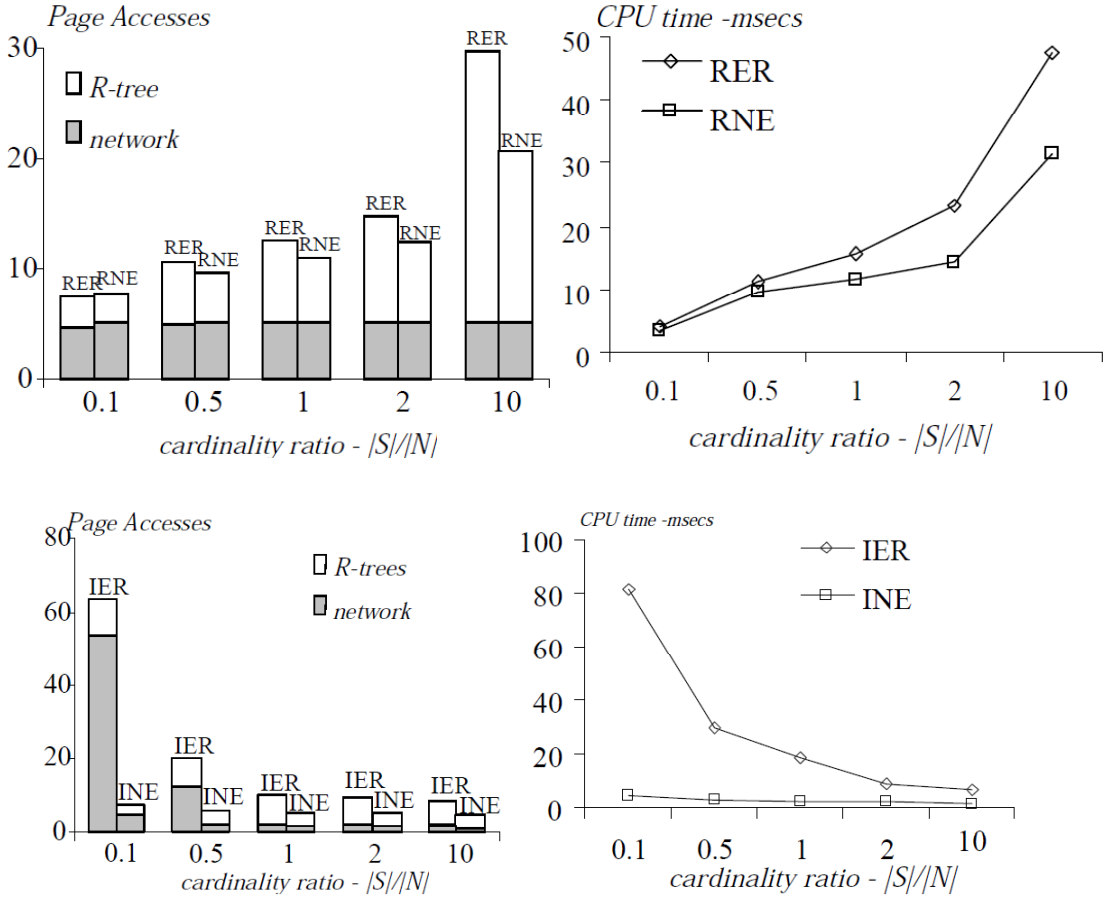
1. $n_i n_j$=*find_segment*(q)
2. S_{cover}=*find_entities*($n_i n_j$); // S_{cover} is the set of entities covered by $n_i n_j$
3. $\{p_1,...,p_k\}$=the k (network) nearest entities in S_{cover} sorted in ascending order of their network distance (p_m, p_{m+1}...,p_k may be ∅ if S_{cover} contains 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