

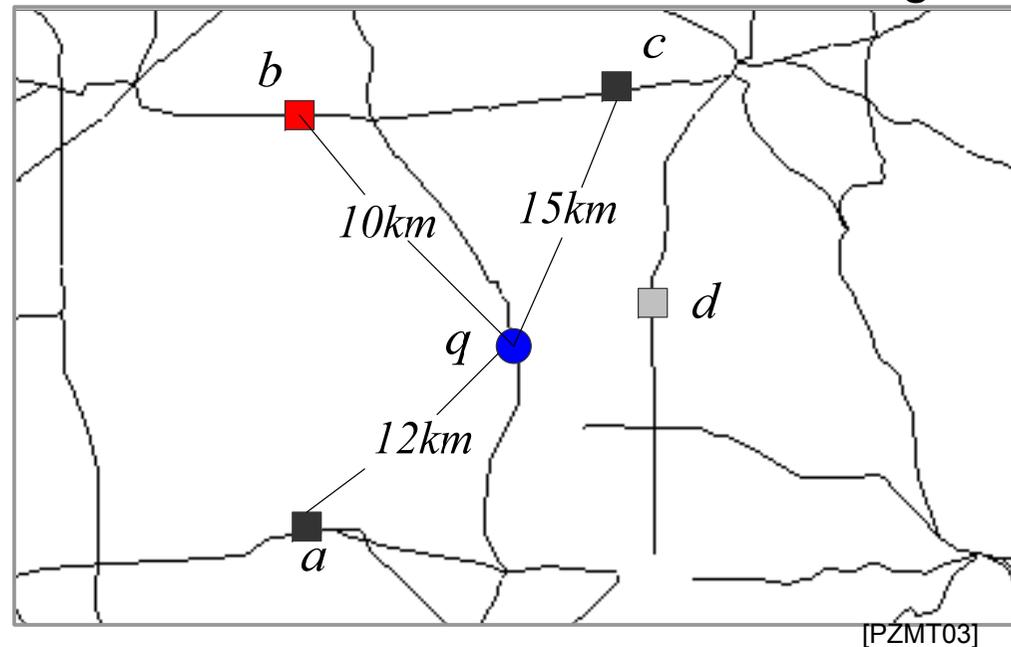
## 5.3 Räumliche Anfragen in Straßennetzwerken

### 5.3.1 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  $d_N(q, n)$  //hole erstes Segment aus der Prioritätswarteschlange

while  $d_N(q, n) \leq e$  and  $S' \neq \emptyset$  //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  $\text{check\_entity}(n_x, n, s)$  //... ob er auf dem Segment  $n_x n$  liegt

        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

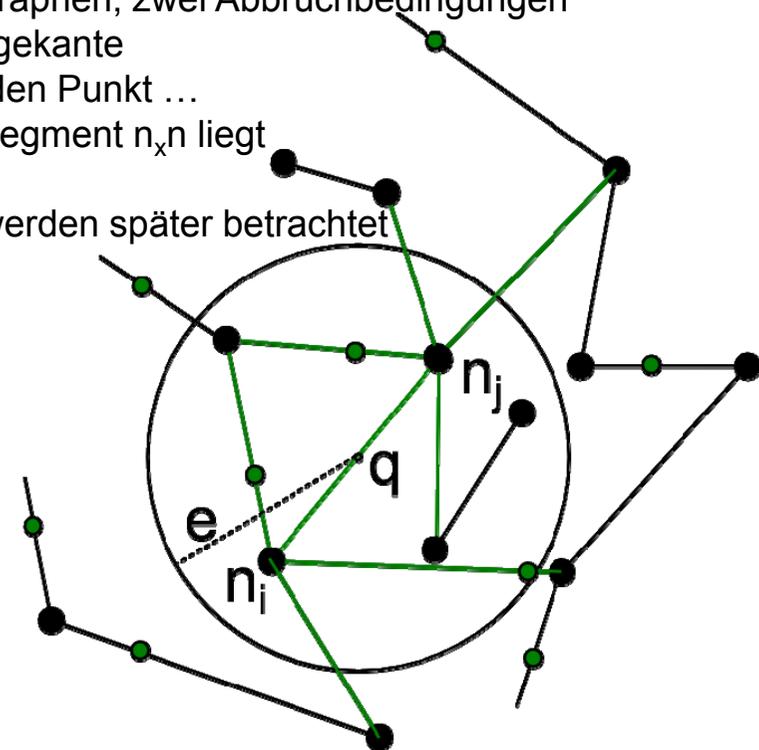
//Traversiere den Graphen, zwei Abbruchbedingungen

//Hole eine Nachfolgekante

//... und teste für jeden Punkt ...

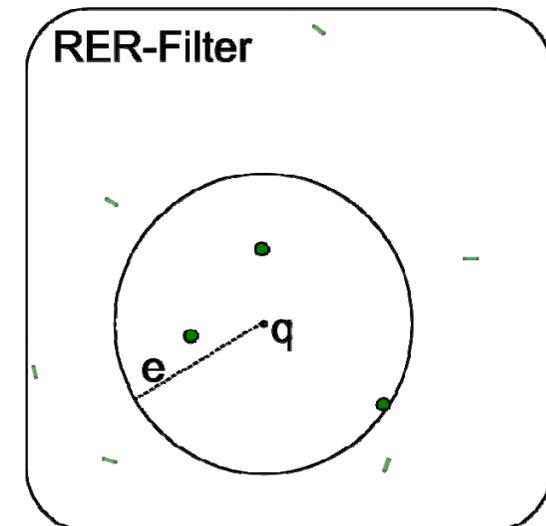
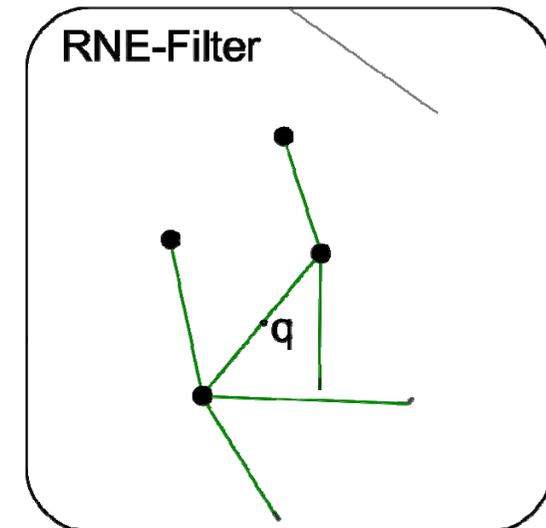
//... ob er auf dem Segment  $n_x n$  liegt

//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<sub>node\_id</sub> = plane-sweep(node\_id.entries,  $QS_i$ )

    sort result<sub>node\_id</sub> to remove duplicates

    result = result<sub>node\_id</sub>  $\cup$  result<sub>node\_id</sub>

//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

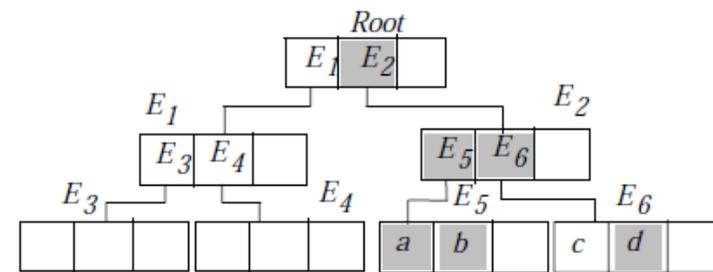
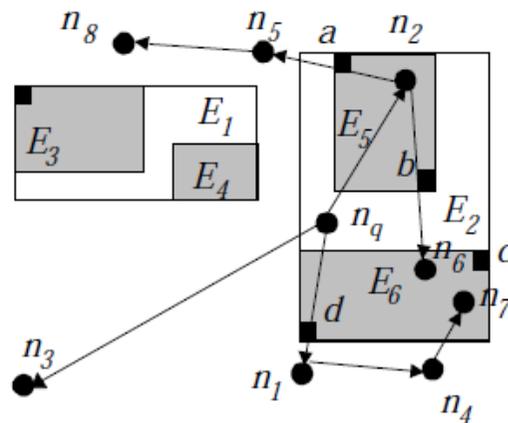
//Schnittmenge nicht leer ist

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

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

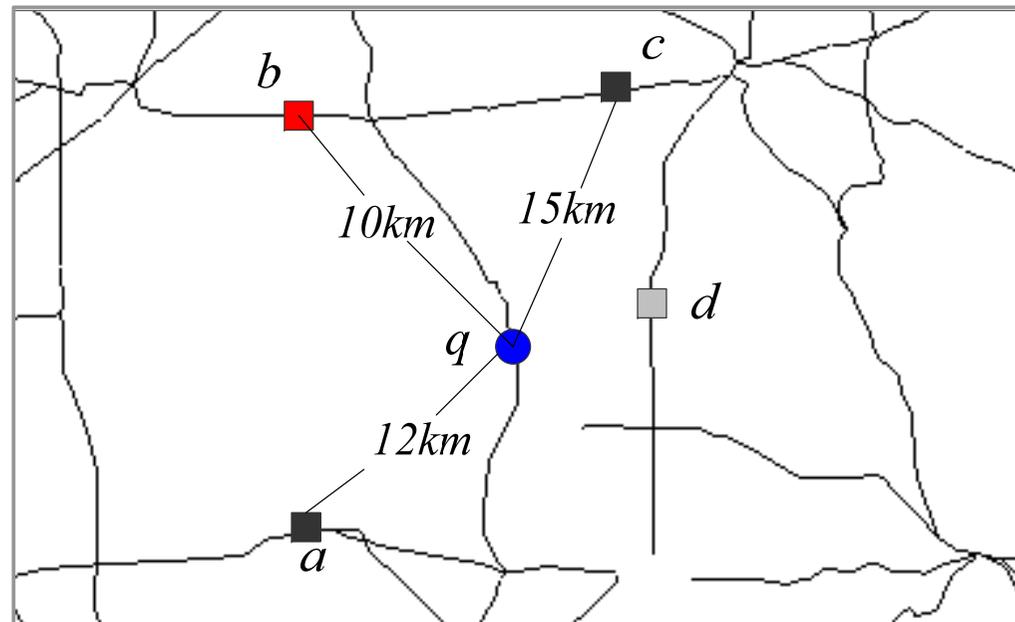
//Segment fallen

//Und entferne ggf. Duplikate



## 5.3.2 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

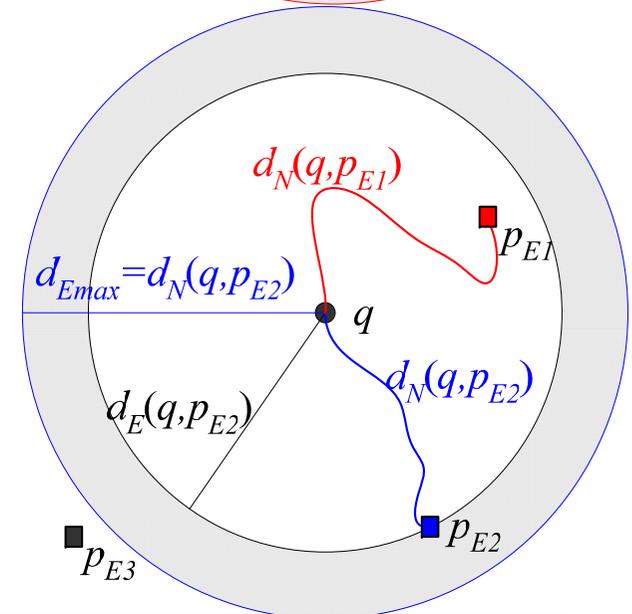
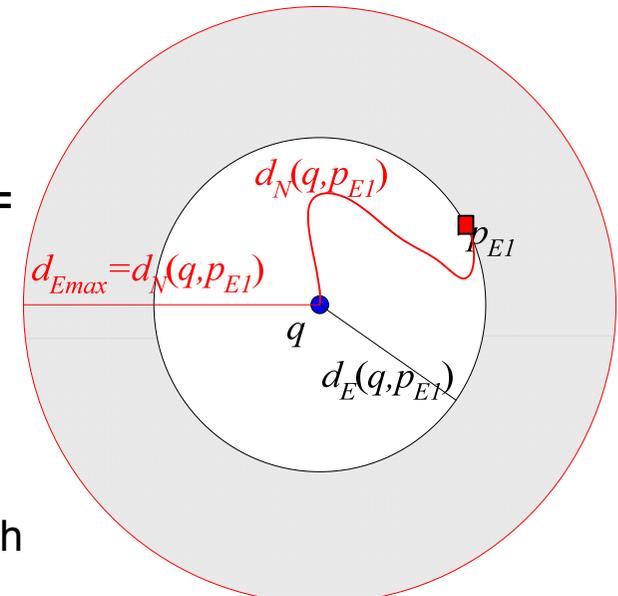
2.  $\forall p_{Ei}$  berechne Netzwerkdistanz  $\text{dist}_{\text{net}}(q, p_{Ei})$

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

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

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

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



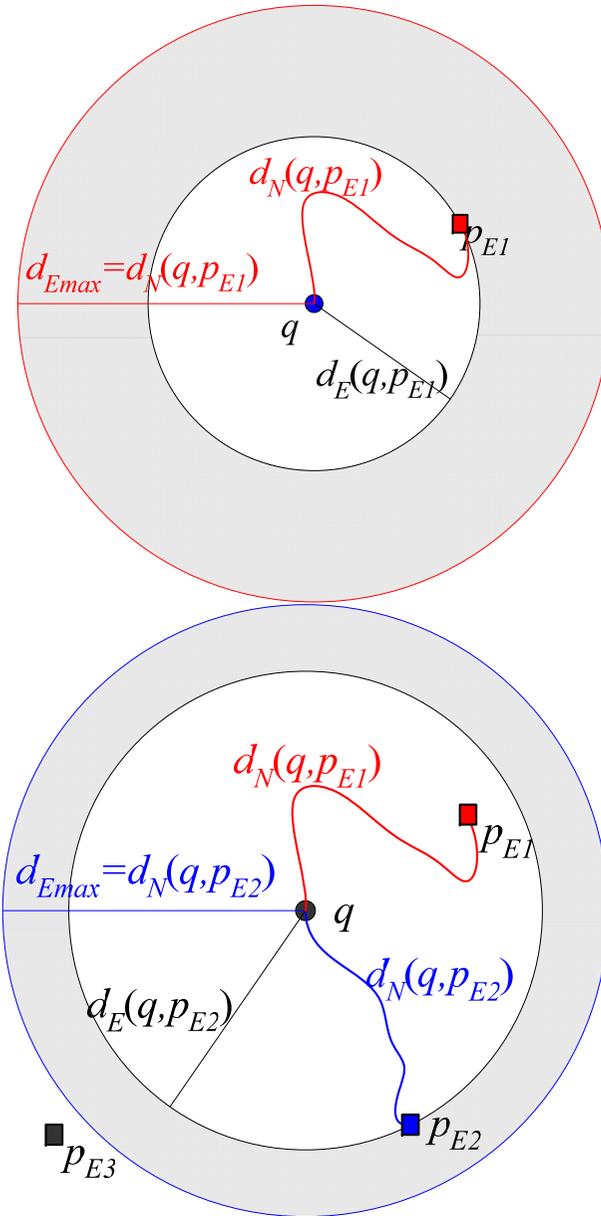
**Algorithm IER ( $q, k$ )**

/\*  $q$  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  $k^{\text{th}}$  NN
9.         insert  $p$  in  $\{p_1, \dots, p_k\}$  // remove ex- $k^{\text{th}}$  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

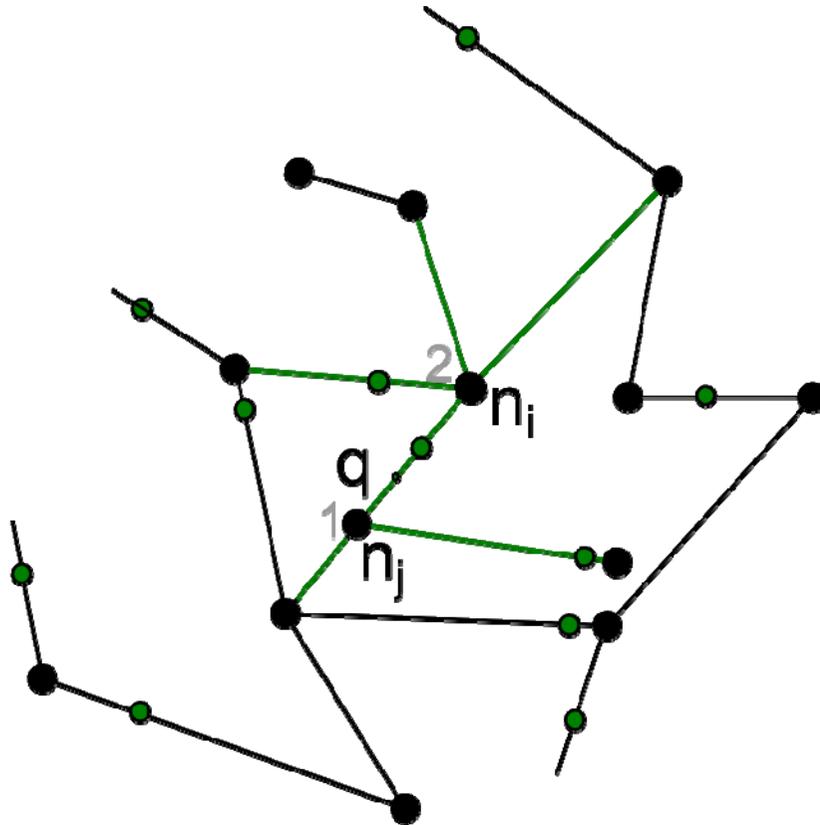


## Berechnung der k-NN-Anfrage

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<sub>[PZMT03]</sub>




---

## Algorithm INE ( $q, k$ )

1.  $n_i n_j = \text{find\_segment}(q)$
2.  $S_{cover} = \text{find\_entities}(n_i n_j)$ ; //  $S_{cover}$  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.  $d_{Nmax} = d_N(q, p_k)$  // if  $p_k = \emptyset$ ,  $d_{Nmax} = \infty$
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 ( $d_N(q, n) < d_{Nmax}$ )
8.     for each non-visited adjacent node  $n_x$  of  $n$
9.          $S_{cover} = \text{find\_entities}(n_x n)$ ;
10.         update  $\{p_1, \dots, p_k\}$  from  $\{p_1, \dots, p_k\} \cup S_{cover}$
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  $Q$

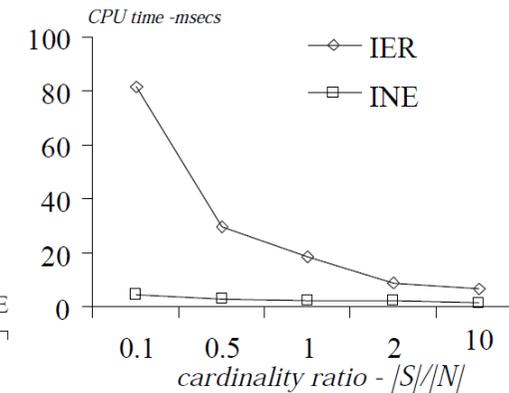
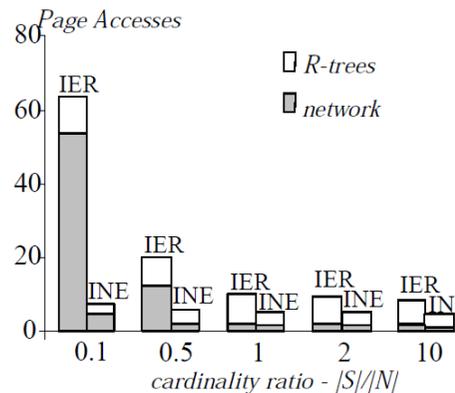
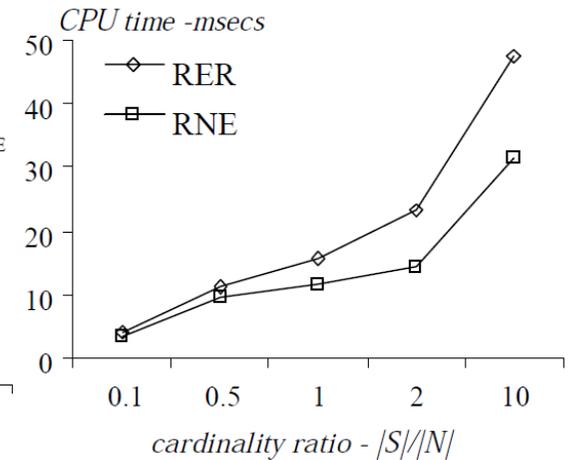
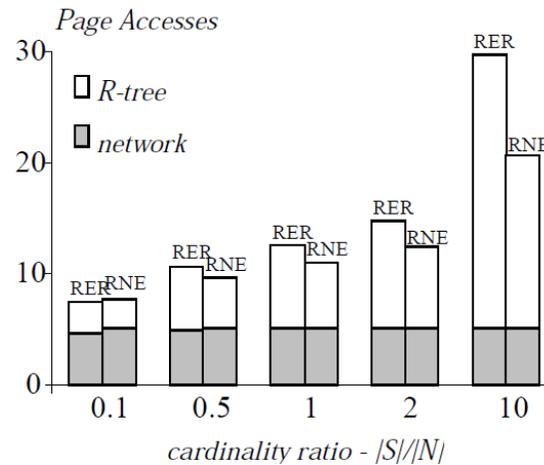
**End INE**

---

**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

## 5.4 Distanz- und Shortest-Path-Berechnung in Straßennetzen

- Von grundlegender Bedeutung für die Anfragebearbeitung auf Straßennetzwerken ist die Berechnung von Distanzen zwischen zwei Knoten
  - Dijkstra: Berechnung der kürzesten Pfade zwischen einem Startknoten und allen verbleibenden Knoten
  - A\*-Algorithmus: Berechnung des kürzesten Pfades zwischen einem Startknoten und einem Endknoten
- Die berechneten Distanzen  $\text{dist}_{\text{net}}(v_i, v_j)$  können dann wiederum zur Bearbeitung komplexerer Anfragen auf Straßennetzen verwendet werden
  - Beispiel: Bereichsanfragen und Nächste-Nachbarn-Anfragen
  - $\text{dist}_{\text{net}}(v_i, v_j)$  im Gegensatz zur euklidischen Distanz nicht unbedingt symmetrisch, d.h.  $\text{dist}_{\text{net}}(v_i, v_j) \neq \text{dist}_{\text{net}}(v_j, v_i)$  (z.B. Einbahnstraßen)

## 5.4.1 Single-Source Shortest Path – Dijkstra

### – Gegeben:

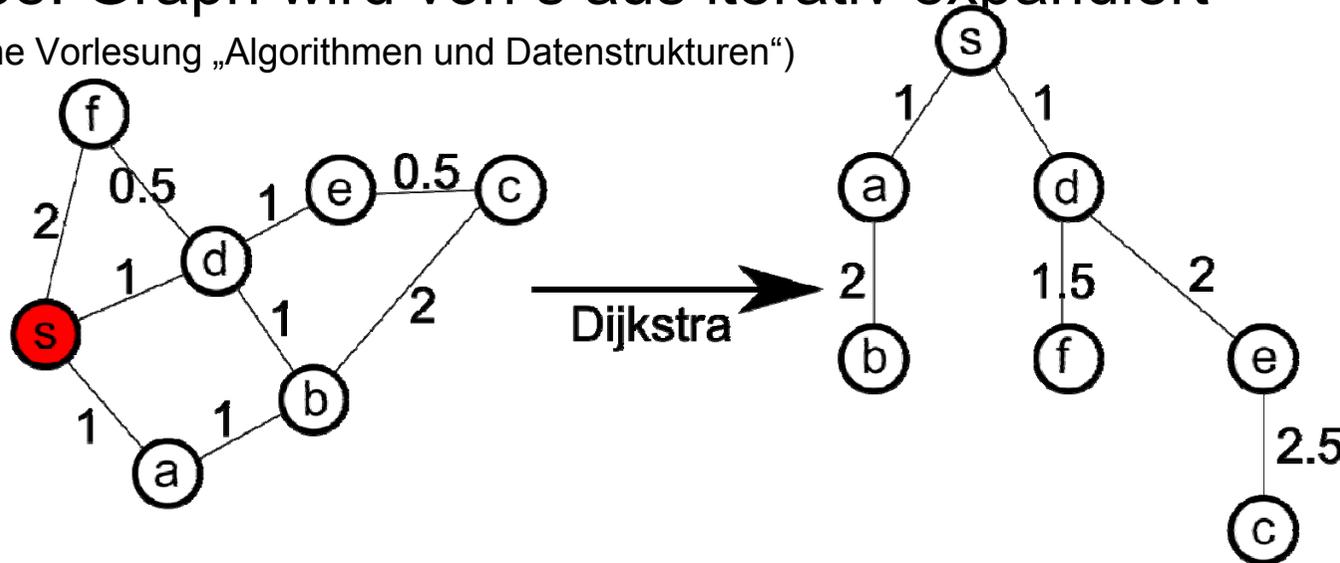
- (gerichteter) Graph  $G=(V,E)$
- Kanten haben nicht-negative, reelle Gewichte

### – Gesucht:

- Kürzester Pfad von einem Startknoten  $s$  zu allen erreichbaren verbleibenden Knoten

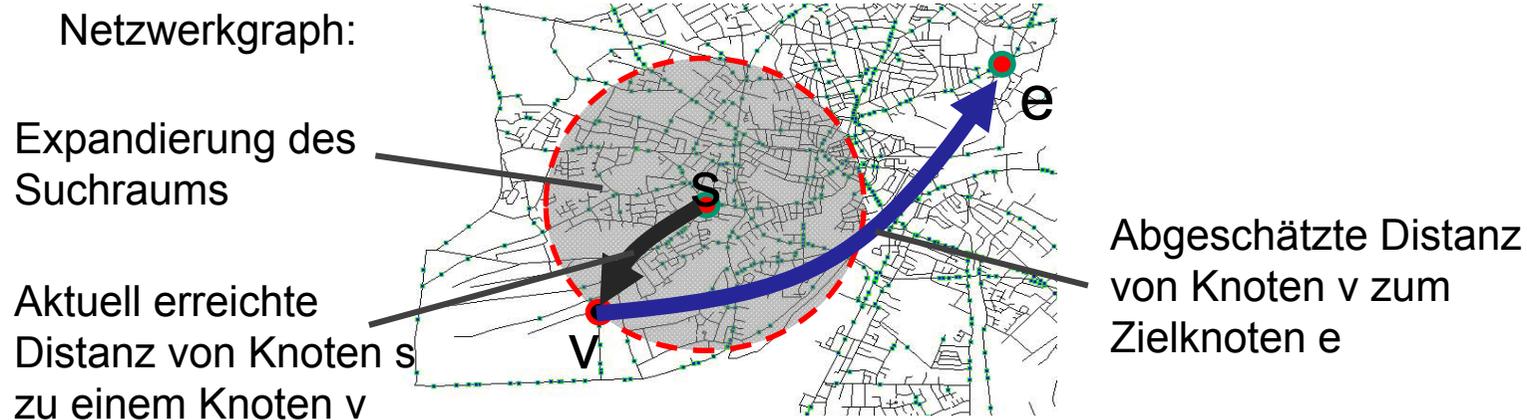
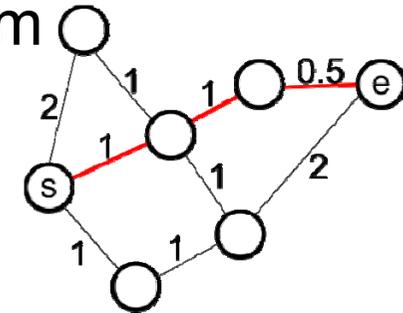
### – Idee: Graph wird von $s$ aus iterativ expandiert

(siehe Vorlesung „Algorithmen und Datenstrukturen“)



## 5.4.2 Der A\*-Algorithmus [HNR68]

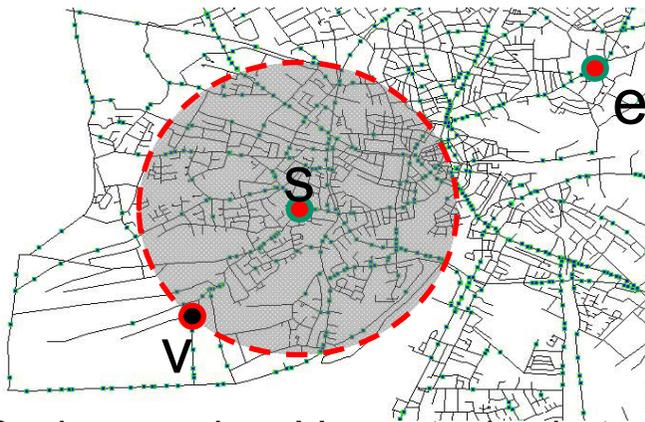
- Ziel: Bestimme den kürzesten Pfad von einem Startknoten  $s$  zu einem Zielknoten  $e$
- Nah verwandt mit dem Dijkstra-Algorithmus
- Idee:
  - Verwendung einer Heuristik zur Vorwärtsabschätzung zum Zielknoten  $e$  für die Abschätzung der gesamten Pfadkosten während der Expansion:



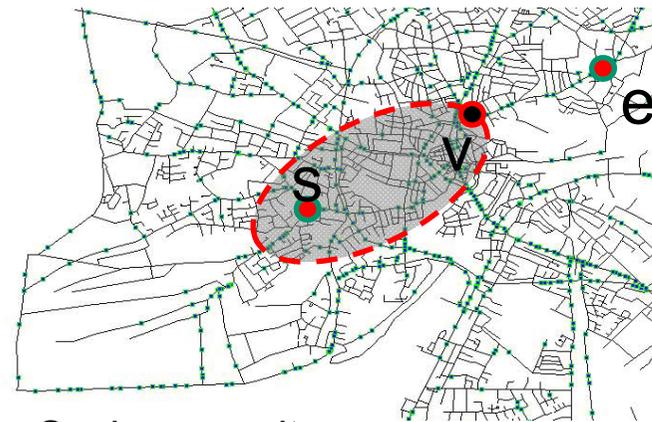
- Best-first-basierte Expansion des Graphen unter Berücksichtigung der Abgeschätzten Gesamtdistanz zwischen  $s$  und  $e$ , d.h. expandiere nur Knoten  $v$  über die der (kürzeste) Pfad von  $s$  nach  $e$  die Gesamtdistanz optimiert.

=> der Graph wird eher in Richtung des Zielknotens expandiert

⇒ Reduzierung des Suchraums durch vorzeitiges Abschneiden nicht-zielführender Pfade



Suchraum ohne Vorwärtsabschätzung  
(Dijkstra)



Suchraum mit  
Vorwärtsabschätzung ( $A^*$ )

- Korrektheit: Die  $A^*$ -Suche findet trotz Verwendung einer Heuristik immer den real kürzesten Pfad von  $s$  nach  $e$

## – A\*-Algorithmus:

- Starte bei Startknoten  $s$
- Expandiere von  $s$  aus den Graphen wie bei Dijkstra iterativ
- Verwendung eines Heaps zur Verwaltung der **relevantesten** Knoten für die weitere Expansion.
- Bei der Expansion von  $s$  werden die Nachbarknoten von  $s$  zunächst in den Heap eingefügt.
- In dem nächsten Iterationsschritt wird das erste Element aus dem Heap geholt und expandiert, dabei werden nur Pfade verfolgt (zugriff auf Nachbarknoten), die die Gesamtkosten von  $s$  nach  $e$  minimieren.
- Die Relevanz  $f(v)$  eines Knotens  $v$  ist bestimmt durch die geschätzte Länge des Weges von  $s$  nach  $e$  über  $v$ :  $f(v) = g(v) + h(v)$ 
  - $f(v)$  – Kosten von  $s$  über  $v$  nach  $e$
  - $g(v)$  – Bisher ermittelte minimale Kosten von  $s$  nach  $v$
  - $h(v)$  – Schätzung der Kosten von  $v$  nach  $e$ , z.B. L2-Distanz

**function A\*(s,e)**

```
closedset =  $\emptyset$  // Bereits betrachtete Knoten
g[s] = 0 // Kosten von s nach Knoten x über den besten bekannten Pfad
h[s] = heuristicCostEstimate(s, e)
openheap = {(s, g[s] + h[s])} // Zu betrachtende Knoten
cameFrom =  $\emptyset$  // Zur Rekonstruktion des kürzesten Pfades

while openheap  $\neq \emptyset$ 
  (x, f) = openheap.poll();
  if x == e
    rekonstruiere Pfad über cameFrom und gib Ergebnispfad zurück
  closedset.add(x);
  foreach y  $\in$  neighbors(x)
    if y  $\in$  closedset:
      continue
    gNew := g[x] + dist(x,y)
    if (gNew < g[y])
      newPathIsBetter := true
    else
      newPathIsBetter := false
    if newPathIsBetter
      cameFrom[y] := x
      g[y] := tentativeGScore
      h[y] := heuristicCostEstimate(y, goal)
      lösche ggf. altes y aus openheap und füge (y, g[y] + h[y]) in openheap ein
return null //kein Pfad gefunden
```

## – Eigenschaften der Schätzfunktion $h(v)$

- Die Korrektheit des A\*-Algorithmus kann nur garantiert werden, wenn die Schätzfunktion  $h(v)$  die Kosten für die Pfad von  $v$  zum Zielknoten unterschätzt. (WICHTIG !!!)
- Damit gilt:  $f(v) \leq \text{dist}_{\text{net}}(s,v) + \text{dist}_{\text{net}}(v,e)$ .
- Je besser die Schätzung von  $h(v)$  (d.h. je größer  $h(v)$ ), desto weniger Knoten werden besucht.
- Bei Verwendung von  $h(v) = 0, \forall v \in V$ , verhält sich A\* wie Dijkstra.

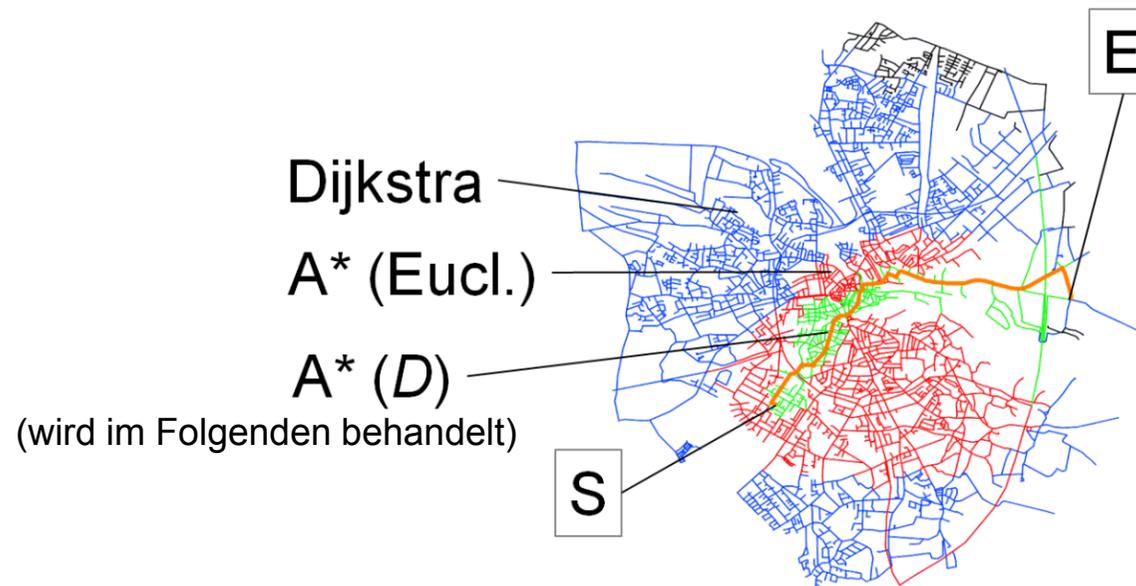
*Heuristiken für A\*: Setze  $h(v) = 0$*

- Die zurückzulegende Distanz wird immer auf 0 gesetzt.
- Dadurch wird der Heap der zu besuchenden Knoten nur anhand der bereits zurückgelegten Strecke sortiert.
- Das entspricht dem Dijkstra-Algorithmus.
- Der Suchraum wird dadurch sehr groß.

Heuristiken für A\*: Setze  $h(v) = \text{dist}_{L_2}(v, e)$

- Verwendet stets die geringste praktisch mögliche Distanz.
- Reale Distanz ist normalerweise weit größer als euklidische Distanz.
- Kleinerer Suchraum als bei Dijkstra, aber immer noch suboptimale Schätzung.

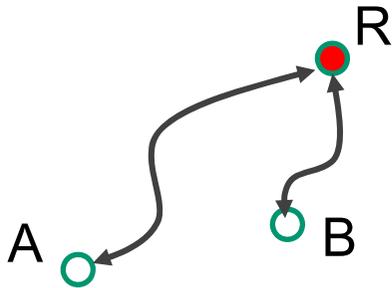
– Vergleich der Heuristiken:



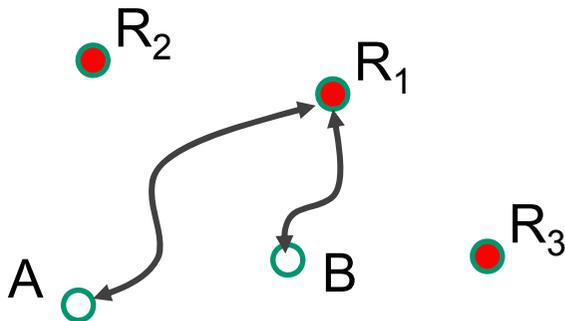
## Heuristiken für A\*: Graph Embedding (D-Distanz) [KKKRS08]

- Idee:

- Berechne Distanzen zu einem/oder mehreren Referenzknoten vor.
- Verwendung dieser Distanzen mit Referenzknoten zur Abschätzung der Distanz zweier beliebiger Knoten



$$\text{dist}_{\text{net}}(A, B) \geq |\text{dist}_{\text{net}}(A, R) - \text{dist}_{\text{net}}(B, R)|$$



Bei  $k$  Referenzknoten  $R_{1, \dots, k}$ :

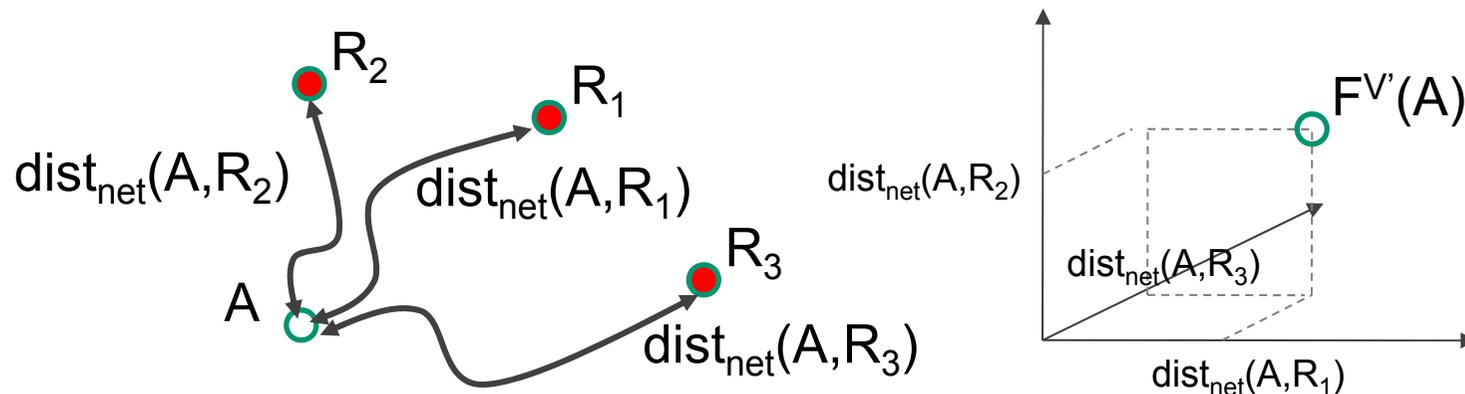
$$\text{dist}_{\text{net}}(A, B) \geq \max_{i=0..k} (|\text{dist}_{\text{net}}(A, R_i) - \text{dist}_{\text{net}}(B, R_i)|)$$

Führt zur besseren Abschätzung als mit nur einem Referenzknoten.

- Wähle eine Teilmenge von Knoten  $V' \subseteq V$ ,  $|V'| = k \geq 1$
- Definiere eine Funktion  $F^{V'} : V \rightarrow \mathbb{R}^k$
- Für das Reference Node Embedding definiere  $F^{V'}$  folgendermaßen:

$$F^{V'}(v) = (F_1^{V'}(v), \dots, F_k^{V'}(v))$$

$$F_i^{V'}(v) = \text{dist}_{\text{net}}(v, v_i) \text{ (Preprocessing!)}$$

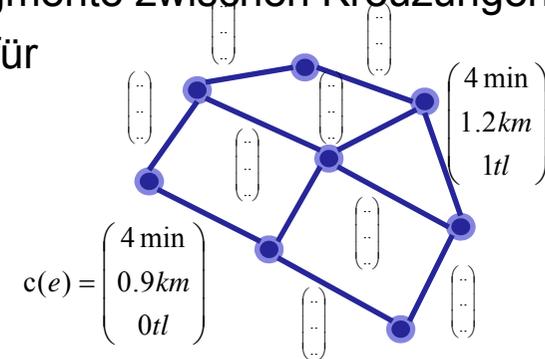


- $F^{V'}(v)$  kann direkt zur Berechnung von  $h(v)$  verwendet werden, denn es gilt  $\forall v_i, v_j \in V: \text{dist}_{L_\infty}(F^{V'}(v_i), F^{V'}(v_j)) \leq \text{dist}_{\text{net}}(v_i, v_j)$ .
- Geringerer Suchraum als bei Verwendung der euklidischen Distanz.

## 5.5 Routen-Suche in Multiattributs-Verkehrsnetzwerken

### – Gegeben:

- Verkehrsnetzwerk
- Straßensegmente haben mehrere Kosten-Attribute, wie z.B.:  
Weglänge, Fahrzeit, Anzahl der Ampeln, Benzinverbrauch (abhängig vom Fahrzeugtyp), ...
- Graph  $G(V,E,c)$ :
  - $V$ : Menge von Knoten  $\leftrightarrow$  Kreuzungen, Sackgassen
  - $E \in V \times V$ : Menge von Kanten  $\leftrightarrow$  Strassensegmente zwischen Kreuzungen
  - $c: E \rightarrow \mathbb{R}^d$ : Kostenfunktion  $\leftrightarrow$  Kostenvektor für Kanten mit  $d$  Kostenattributen
- Annahmen:
  - Keine **negativen** Kosten (**Warum?**)
  - Alle Kostenattribute sind additiv  
(Gilt für Kosten eines Pfades (Route))



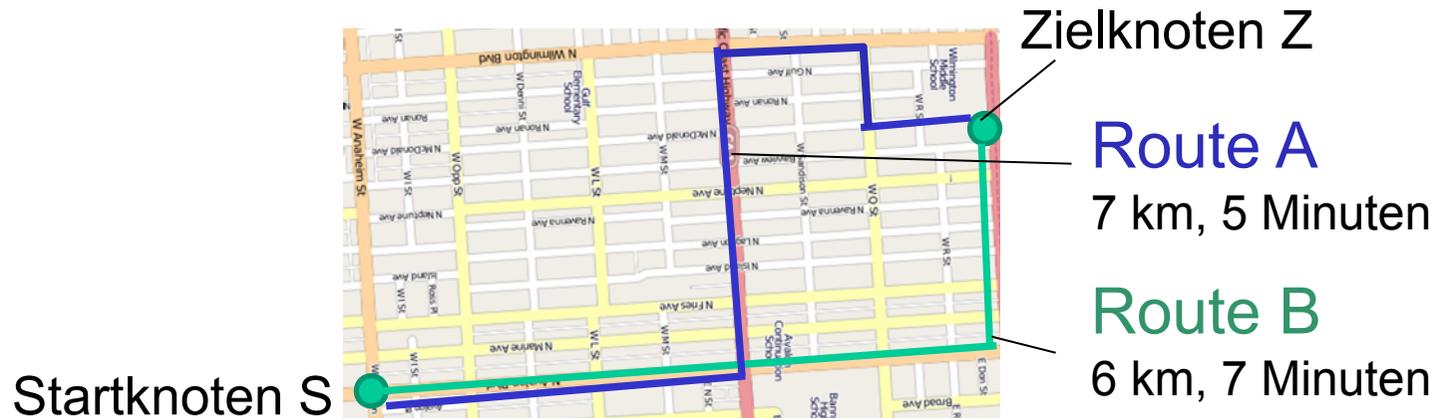
- Route:
  - $p = \langle v_1, \dots, v_n \rangle$  mit  $(v_i, v_{i+1}) \in E, \forall 1 \leq i < n$
  - ohne Zyklus, d.h.  $\forall 1 \leq i < n, \forall 1 \leq j < n, i \neq j, v_i \neq v_j$
  - Kosten einer Route:  $\text{cost}(p) = \sum_{e \in p} c(e) = \begin{pmatrix} \text{cost}(p)_1 \\ \vdots \\ \text{cost}(p)_d \end{pmatrix}$
- Gesucht:
  - “Bester” Pfad von S nach Z



**:= Pfad mit minimalen Kosten**  $\sum_{i=1}^d w_i \cdot \text{cost}(p)_i$  ?

## – Problem

- Mehrere (Pfad-)Attribute (Kostenattribute) zu berücksichtigen (i.d.R. eine Auswahl an Kostenattribute)



## Welcher Pfad ist optimal A oder B?

- Kostenattribute schwierig zu vergleichen:  
#Ampeln  $\leftrightarrow$  Strecke?
- Präferenzen sind abhängig vom Benutzer und Anwendung:  
Geschäftlich oder privat unterwegs?
- Geeignete Gewichtung der Attribute schwer zu ermitteln

– Ansatz:

Ausgabe aller **pareto-optimaler** Pfade → **Route Skyline**

– Anfrage:

- Finde die Menge RS alle Pfade zwischen S und Z, sodaß für jeden Pfad  $p \in RS$  kein Pfad  $p'$  zwischen S und Z existiert mit der folgenden Eigenschaft:

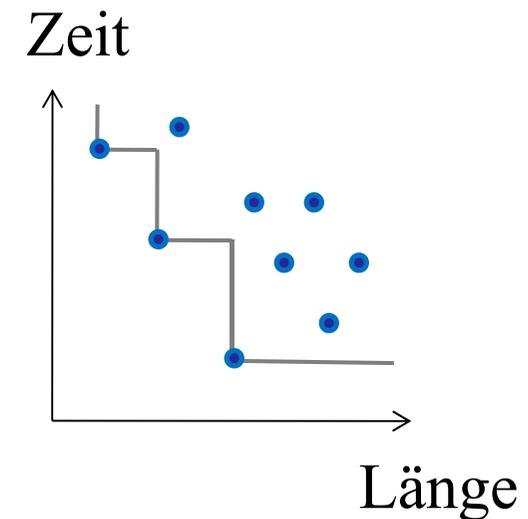
$$\forall 1 \leq i \leq d : \text{cost}(p')_i \geq \text{cost}(p)_i$$

$$\exists 1 \leq i \leq d : \text{cost}(p')_i > \text{cost}(p)_i$$

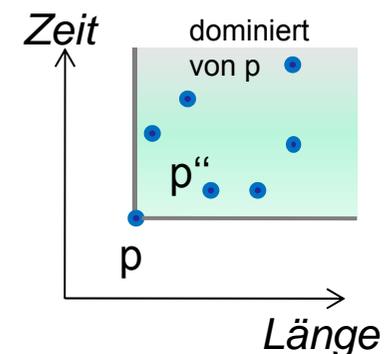
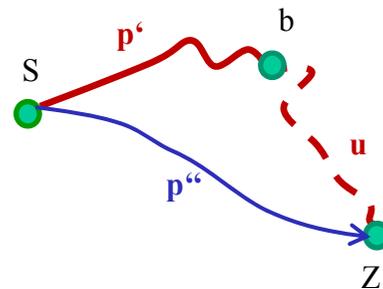
- Man spricht:  $p$  wird **nicht** von  $p'$  dominiert

– Probleme:

- Anzahl potentieller Pfade extrem hoch
  - Materialisierung der Kosten aller Pfade extrem teuer
- => Skylineanfragemethoden für Vektorobjekte nicht (effizient) anwendbar

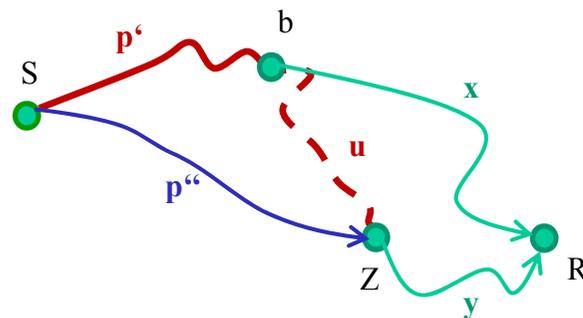


- Effiziente Route-Skyline Berechnung: [KRS 10]
- Idee:
  - Graphdurchlauf bei Startknoten S starten analog zu Dijkstra/A\*-Suche
  - Erweitere eine Route  $p'$  falls  $p'$  Teilroute einer Skylineroute  $p$  sein kann, d.h. erweitere  $p'$  nicht wenn abgeschätzt werden kann dass  $p$  von einer anderen Route  $p''$  dominiert wird
  - Stoppe falls keine Route mehr erweitert werden kann
- Gegeben eine Teilroute  $p' = \langle S, \dots, b \rangle$ ,



- Wie kann man abschätzen ob  $p'$  Teil einer Skylineroute ist? => **Pruningkriterien**

- Pruningkriterium I: Pruning mittels Vorwärtsabschätzung
  - Basiert auf Kostenabschätzung mittels Referenzknoten (siehe Kap. zu Graph Embedding)
  - Schätze Gesamtkosten (Kostenvektor)  $\text{cost}(p)$  einer potentiellen Route (über einen Referenzknoten R) ab
  - Falls bereits ein Pfad  $p''$  existiert, der den Vektor  $\text{cost}(p)$  dominiert, dann kann  $p'$  nicht zu einer Skylineroute erweitert werden



$$\text{cost}(u)_i \geq |\text{cost}(x)_i - \text{cost}(y)_i|$$

- Funktioniert solange die Kostenabschätzung die untere Schrankeneigenschaft erfüllt

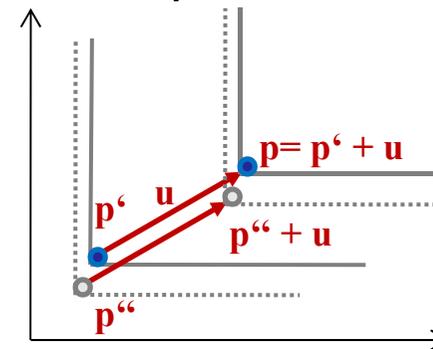
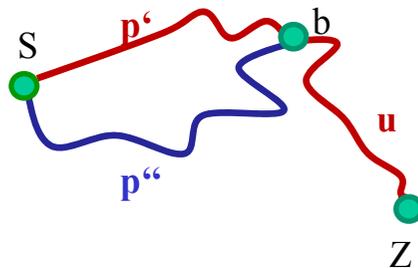
– Pruningkriterium II: Pruning unter Verwendung der Monotonieeigenschaft der Dominanzbeziehung

Theorem:

Gegeben sei eine Skyline Route  $p = \langle S, \dots, b, \dots, Z \rangle$ , d.h.  $p$  wird von keiner anderen Route zwischen  $S$  und  $Z$  dominiert. Dann wird jede Teilroute  $p' = \langle S, \dots, b \rangle$  von  $p$  von keiner anderen Teilroute  $p'' = \langle S, \dots, b \rangle$  dominiert.

Beweis:

Annahme es gibt eine Route  $p'' = \langle S, \dots, b \rangle$ , die die Teilroute  $p' = \langle S, \dots, b \rangle \neq p''$  von  $p$  dominiert. Dann würde die Erweiterung von  $p''$  über die Teilroute  $u = \langle b, \dots, Z \rangle$  von  $p$  zu einer Route führen, die die Route  $p$  dominiert  $\Rightarrow$  Widerspruch zur Annahme daß  $p$  Skylineroute ist.



## – Skyline-Route-Algorithmus:

- Idee: wie A\*-Suche, aber verwalte bei jedem Knoten n eine Skyline (Skyline bzgl. aller Pfade  $p' = \langle S, \dots, n \rangle$ )

- Algorithmus:

Input: Start S, Ziel Z, Graph(V,E,L) (mit Embedding)

Output: alle Skyline-Routen zwischen S und D

initialisiere Routen-Heap (queue);

queue.insert(S);

while (queue is not empty)

    aktRoute = queue.top();

*// erweitere alle Skyline Routen von Knoten aktNode*

    cand = extend(aktRoute.last\_node.SkR);

**for all**  $c \in \text{cand}$

**if**  $\exists p \in Z.SkR$ , p dominiert  $\text{cost}_{\text{est}}(c)$ , **then** lösche c aus cand;

**else**

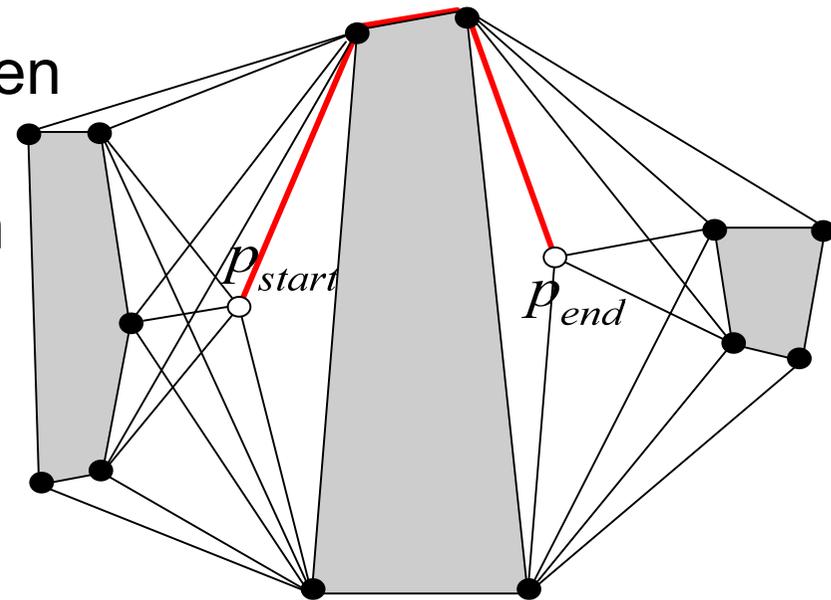
            update(c.last\_node.SkylineRoutes,c);

            queue.insert(c);

report(Z.SkR);

## 5.6 Anfragen im Euklidischen Raum mit Hindernissen [ZPMZ04]

- Ziel: Anfragebearbeitung (z.B. NN-Anfragen) im euklidischen Raum bei Anwesenheit von Hindernissen (Häuser, Seen, ...)
- Mögliche Pfade (mit kürzesten Distanzen) können mittels Sichtbarkeitsgraphen modelliert werden.
- Für die Ermittlung der kürzesten Distanz zwischen zwei Punkten können Graphalgorithmen (z.B. Dijkstra) verwendet werden.



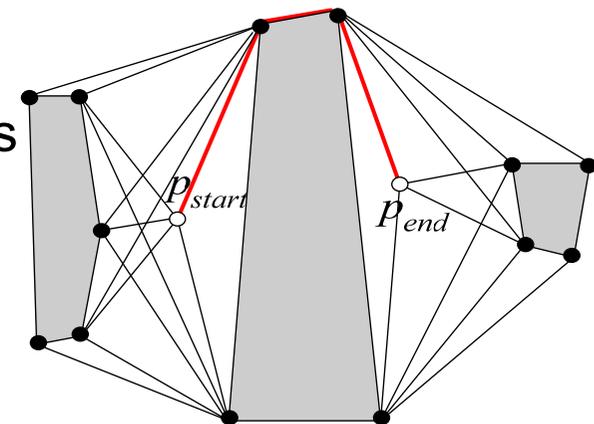
## 5.6.1 Sichtbarkeitsgraph

- Dient der Modellierung der kürzesten Pfade zwischen zwei beliebigen Punkten im Raum mit Hindernissen
- Hindernisse werden als Polygone modelliert
- Knoten: Eckpunkte von Hindernissen, Objekte ( $p_i$ )
- Gilt nur im 2-dimensionalen Raum !!!
- Kanten: Verbinden Knoten die gegenseitig „sichtbar“ sind (d.h. dürfen kein Hindernis kreuzen)
- Ermittlung des Sichtbarkeitsgraphen:

- Suche für jeden Eckpunkt eines Polygons die Eckpunkte anderer Polygone die direkt „sichtbar“ sind.

(siehe Computational Geometry [BKOS97])

- Naive Suche:  $O(n^3)$
- Rotational plane-sweep:  $O(n^2 \log n)$  [SS84],  $O(n^2)$  [W85, AGH186]

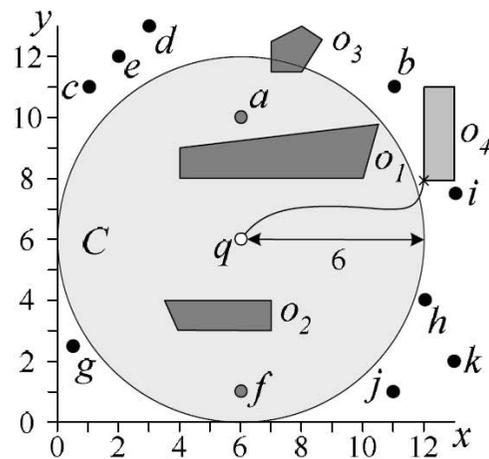


### 5.6.2 Idee bei Nachbarschaftsanfragen in Räumen mit Hindernis

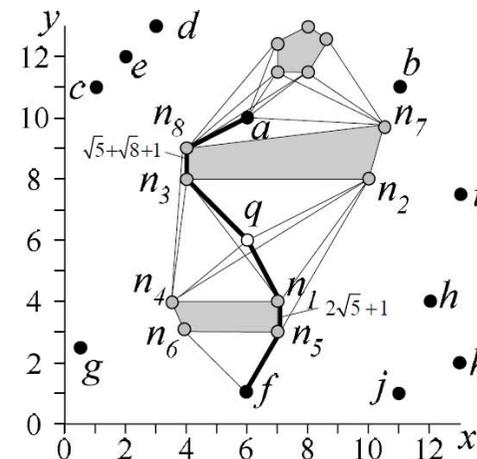
- Berechnung des gesamten Sichtbarkeitsgraphen zu aufwändig.
- Deshalb Reduktion auf relevante Teilgraphen, bei denen nur relevante Hindernisse betrachtet werden.
- Welche Hindernisse sind für die Anfragebearbeitung relevant?
- Euklidische Distanz zwischen zwei Punkten bildet eine untere Schranke für die Distanz in Räumen mit Hindernissen.
- Ermittlung der relevanten Hindernisse über die Verwendung der Euklidischen Distanz (Filterschritt)

## 5.6.2 Bereichsanfrage mit Hindernissen

1. Euklidische Bereichsanfrage auf Ergebniskandidaten  $P$  von Anfragepunkt  $q$  mit Distanz  $e$  (filter)
2. Suche von für die Anfrage relevanten Hindernissen  $O$ . Diese schneiden die Fläche der Bereichsanfrage.
3. Lokalen Sichtbarkeitsgraph über  $P$  und  $O$  aufbauen
4. False Hits aus  $P$  über Sichtbarkeitsgraph entfernen



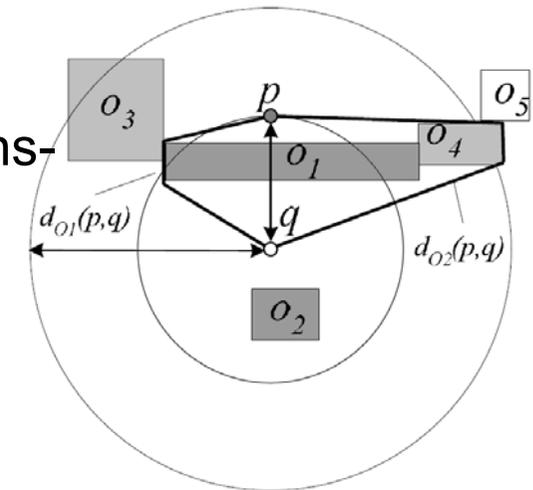
(a) Obstacle range query



(b) Local visibility graph

## 5.6.2 NN-Anfrage mit Hindernissen

- Entspricht dem IER-Algorithmus, wobei jede Runde der entsprechende Sichtbarkeitsgraph erzeugt werden muss, um  $\text{dist}_{\text{NET}}(p,q)$  zu berechnen.
- Initialisiere NN-Rankinganfrage über die Euklid. Dist.
- Berechnung der Netzwerkdistanz  $\text{dist}_{\text{NET}}(p,q)$  für jeden NN-Kandidaten  $p$  über den Sichtbarkeitsgraphen:
  1. Berechne Hindernisse  $O$ , die  $[p,q]$  schneiden
  2. Berechne den Sichtbarkeitsgraph über  $O \cup \{p,q\}$  und daraus  $\text{dist}'_{\text{NET}}(p,q)$
  3. Frage alle Hindernisse  $O$  an, die die Bereichsanfrage von  $q$  bzgl.  $\text{dist}'_{\text{NET}}(p,q)$  schneiden
  4. Führe 3. und 4. so lange durch, bis  $O$  oder  $\text{dist}'_{\text{NET}}(p,q)$  sich nicht mehr verändert.  
Es gilt:  $\text{dist}_{\text{NET}}(p,q) = \text{dist}'_{\text{NET}}(p,q)$



[HNR68]: Hart, Nilsson, Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In IEEE Transactions of Systems Science and Cybernetics, 1968.

[KRS10]: Kriegel, Renz, Schubert: Route Skyline Queries: A Multi-Preference Path Planning Approach. In Proc. of ICDE, 2010.

[KKRS08]: Kriegel, Kröger, Kunath, Renz, Schmidt: Efficient Query Processing in Large Traffic Networks. In Proc. of ICDE, 2008.

[PZMT03]: Papadias, Zhang, Mamoulis, Tao. Query Processing in Spatial Network Databases. In Proc. of VLDB, 2003.

[BKOS97] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. Computational Geometry. pp. 305-315, Springer, 1997.

[SS84] Sharir, M., Schorr, A. On Shortest Paths in Polyhedral Spaces. STOC, 1984.

[W85] Welzl, E. Constructing the Visibility Graph for  $n$  Line Segments in  $O(n^2)$  Time, Information Processing Letters 20, 167-171, 1985.

[AGHI86] Asano, T., Guibas, L., Hershberger, J., Imai, H. Visibility of Disjoint Polygons. Algorithmica 1, 49-63, 1986.

[ZPMZ04]: Zhang, Papadias, Mouratidis, Zhou. Spatial Queries in the Presence of Obstacles. In Proc. of EDBT, 2004.