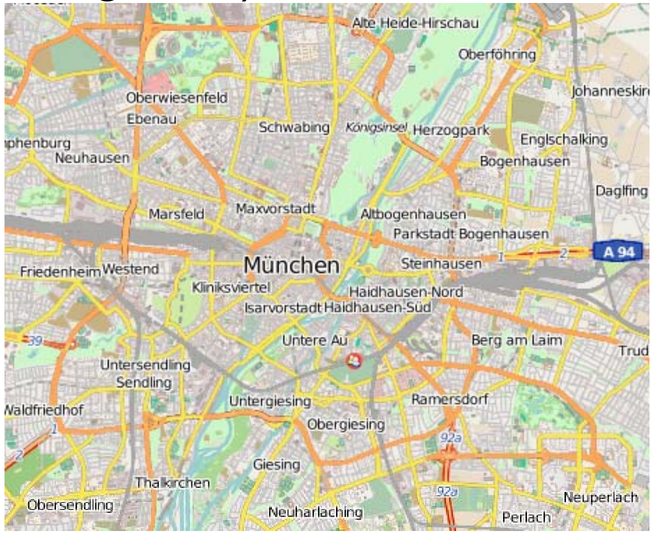
Kapitel 4

Anfragemethoden auf Verkehrsnetzwerke

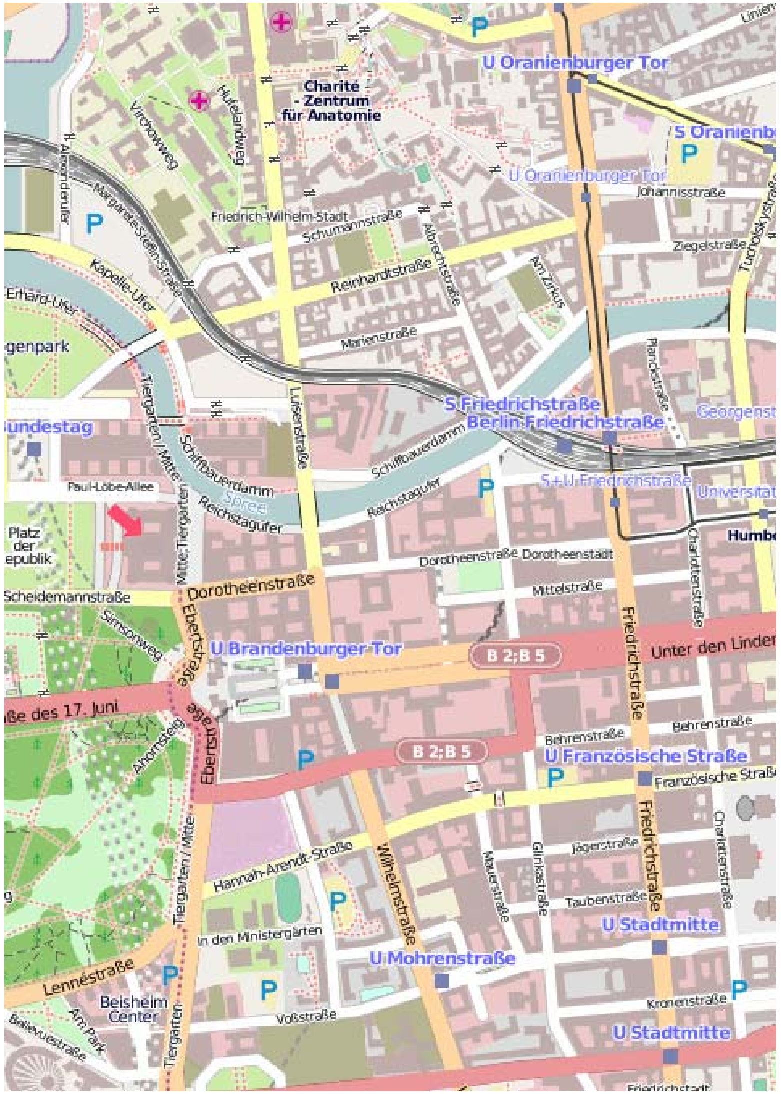
Skript zur Vorlesung: Spatial, Temporal, and Multimedia Databases II
Wintersemester 2011/12, LMU München

4.1 Motivation

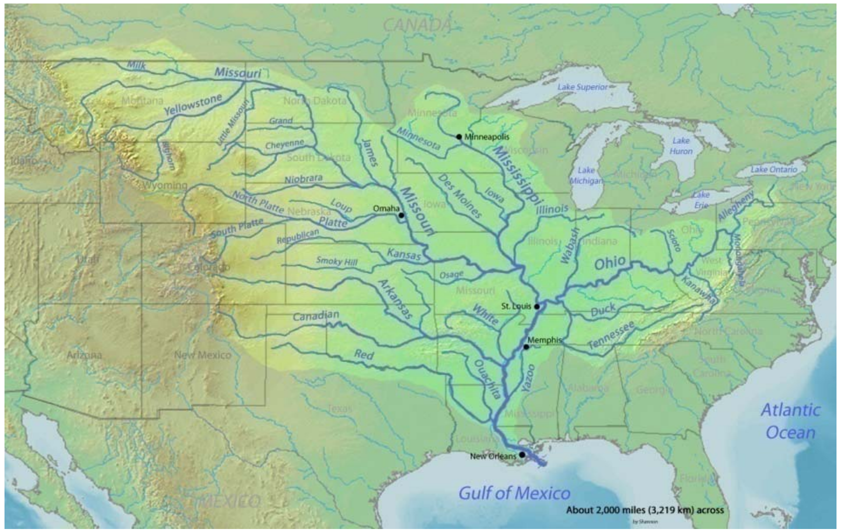
- Bisher: Distanz zwischen zwei Punkten über einfacher Distanzmaße wie z.B. die Euklidische Distanz berechnet.
- In vielen Anwendungen, z.B. im Straßenverkehr, existieren Einschränkungen bzgl. möglicher Bewegungsrichtungen.
- In Verkehrsnetzwerken wird als Distanz zwischen zwei Punkten i.d.R. die Länge des kürzesten Pfades (Netzwerkdistanz) 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 Schienennetzerke



Maximilian Dörrbecker



– 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 näheste Computergeschäft nach zu laufender Strecke.”
  - “Finde den kürzesten Pfad, um mehrere Shops zu beliefern.”
  - “Verweise Kunden auf den nähesten Kundendienst.”

4.2 Modellierung von Verkehrsnetzwerken

Drei-Ebenen-Modell:

1. Konzeptuelles Datenmodell: Graphen

2. Logisches Datenmodell
   - Datentypen
     - Graph, Vertex, Edge, Path, …
   - Operationen (Queries)
     - is connected(.,.), shortest-path(), weitere Nachbarschaftsanfragen, ...

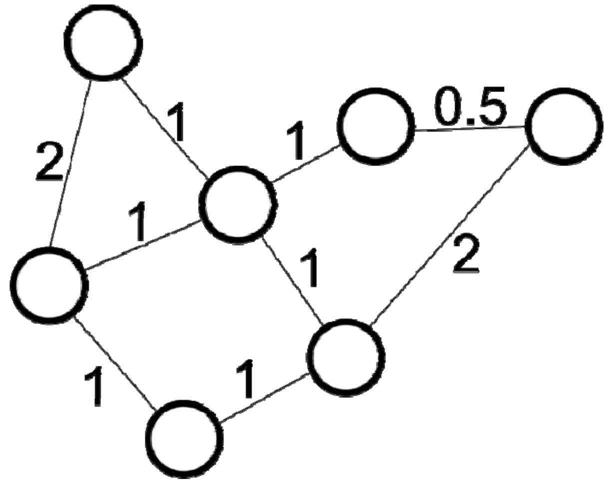
3. Physisches Datenmodell
   - Hauptspeicherbasierte Darstellung
   - Festplattenbasierte Darstellung

4.2.1 Konzeptuelles Modell

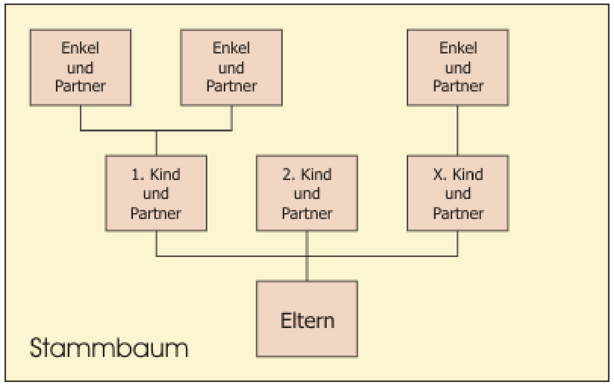
- Straßennetze können als Graph G=(V,E), bestehend aus einer Menge von Knoten V sowie einer Menge von (gerichteten) Kanten $E \subset \{(v_i, v_j) \mid v_i, v_j \in V\}$ modelliert werden

- 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*



- Kanten haben Gewichte (Kosten), z. B.
  - reellwertig
    - Entfernung zwischen zwei Knoten
    - Fahrtdauer
    - Benzinverbrauch
    - gewichtete Kombination mehrerer Merkmale
  - vektorielle Kombination mehrerer reellwertiger Gewichte [KRS10]
- Klassifikation von Graphen
  - Repräsentieren Knoten räumliche Punkte? Räumlicher Graph ↔ Abstrakter Graph
  - Gerichteter Graph ↔ Ungerichteter Graph



4.2.2 Logisches Datenmodell – Datentypen

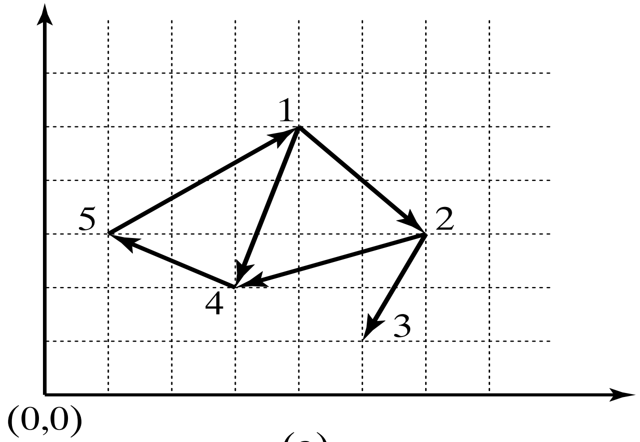
– Beschreibt das Schema einer Räumlichen Netzwerk Datenbank (Spatial Network Database SNDB)

- 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. edge(vertex A, 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

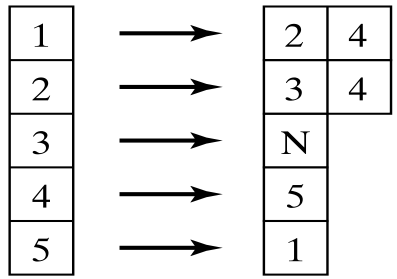


(a)

Destination

source	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



(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

4.3 Basisalgorithmen auf 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

–Floyd: Berechnung des kürzesten Pfades von jedem Knoten zu jedem anderen Knoten

• Die berechneten Distanzen $dist_{net}(v_i, v_j)$ können dann wiederum zur Bearbeitung komplexerer Anfragen auf Straßennetzen verwendet werden

– Bereichsanfragen

– Nächste-Nachbarn-Anfragen

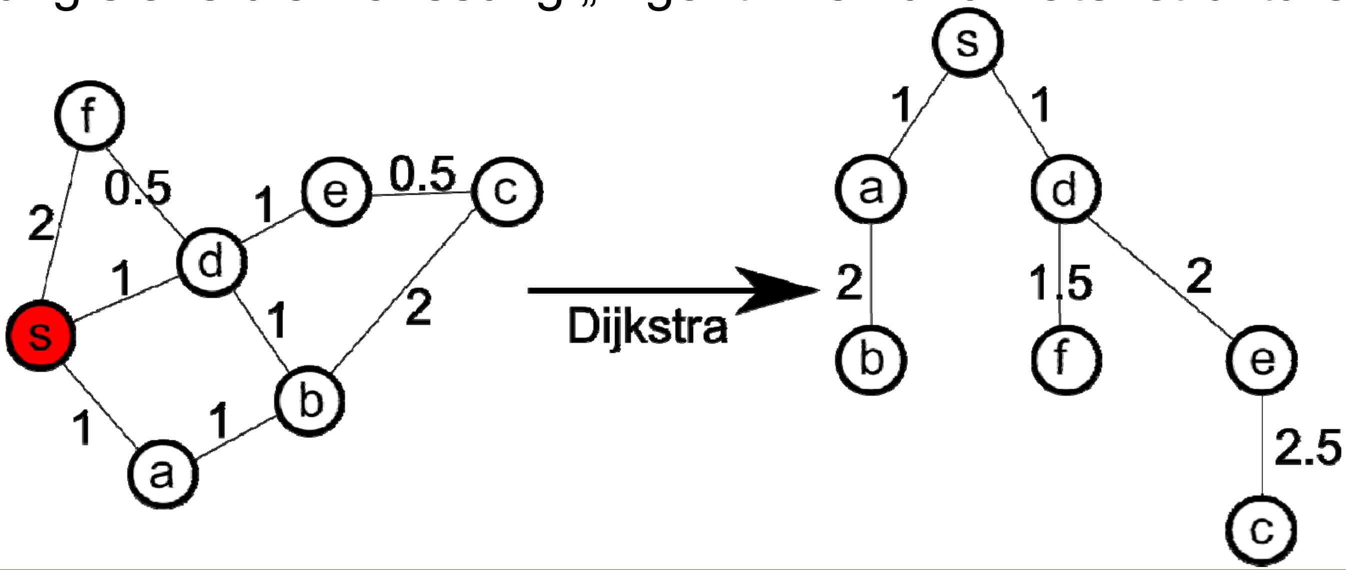
– Skyline-Queries

• $dist_{net}(v_i, v_j)$ im Gegensatz zur euklidischen Distanz nicht unbedingt symmetrisch (z.B. Einbahnstraßen): $dist_{net}(v_i, v_j) \neq dist_{net}(v_j, v_i)$

4.3.1 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

- Hier nur eine kurze Wiederholung des Algorithmus, für eine genaue Erläuterung siehe die Vorlesung „Algorithmen und Datenstrukturen“



4.3.2 Der A*-Algorithmus[HNR68]

•Ziel: Bestimme den kürzesten Pfad von einem Startknoten *s* zu einem Zielknoten *e*

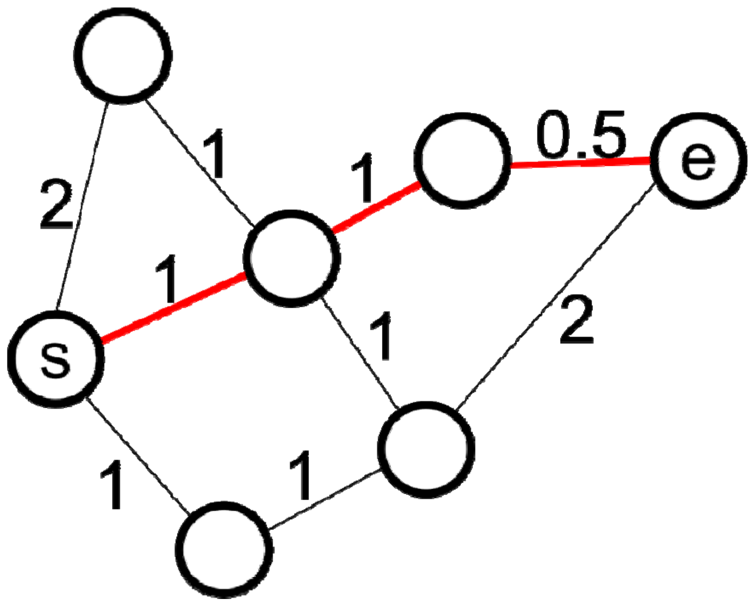
•Nah verwandt mit dem Dijkstra-Algorithmus

•Heuristisches Verfahren: A* schätzt Entfernungen zwischen Knoten, dadurch Reduktion des Suchaufwands

•Best-First-Search mit Hilfe der gewählten Heuristik

•Eigenschaften:

–Korrekt: Findet trotz Verwendung einer Heuristik Immer den kürzesten Pfad von s nach e



Heuristik des A-Algorithmus:*

– Knoten $v_i \in V$ werden nicht wie bei Dijkstra in zufälliger Reihenfolge, sondern anhand ihrer erwarteten Relevanz in Best-First-Manier betrachtet

– Die Relevanz eines Knotens f(v_i) ist bestimmt durch die geschätzte Länge des Weges von s nach e über v_i: $f(v_i) = g(v_i) + h(v_i)$

- *$f(v_i)$ – Kosten von s über v_i nach e*
- *$g(v_i)$ – bisher ermittelte minimale Kosten von s nach v_i*
- *$h(v_i)$ – Schätzung der Kosten von v_i nach e, z.B. L_2-Distanz*
    - *– Wichtig: Lower-Bounding- Eigenschaft von h(n).*
    - *– Je besser die Schätzung von $h(v_i)$, desto weniger Knoten müssen besucht werden*
    - *– Bei Verwendung von $h(v_i) = 0$ verhält sich A* wie Dijkstra*

```
function A*(s,e)
    closedset = Ø    // 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 = Ø    // Zur Rekonstruktion des kürzesten Pfades

    while openheap≠Ø
        (x, f) = openheap.poll();
        if x == e
            rekonstruiere Pfad über cameFrom und gib Ergebnispfad zurück
        closedset.add(x);
        foreach y ∊ neighbors(x)
            if y ∊ closedset:
                continue
            gNew := g[x] + dist(x,y)
            if (y ∉ openheap) □ (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
```

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 extrem groß

Heuristiken für A: Setzte* $h(v) = dist_{L_2}(v,e)$

– Verwendet stets die geringste praktisch mögliche Distanz

– Reale Distanz ist normalerweise weit größer als euklidische Distanz

– Kleinerer Suchraum als bei Dijkstra, aber immer noch sub-optimale Schätzung

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

– Wähle eine Teilmenge von Knoten $V' \subseteq V$, $|V'| = k \geq 1$

– Definiere eine Funktion $F^{V'} : V \rightarrow \mathbb{R}^k$

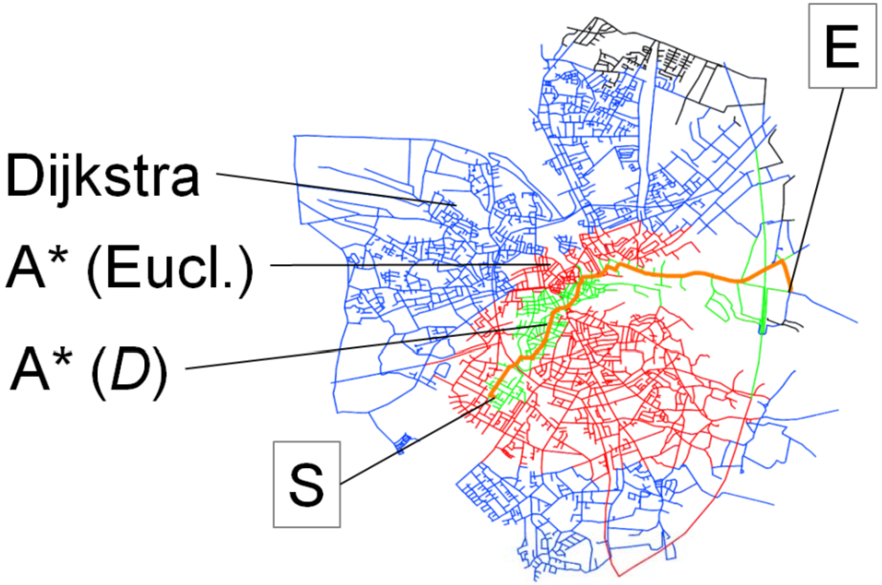
– Für das Reference Node Embedding definiere $F^{V'}$ folgendermaßen:

$$F^{V'}(v_i) = (F_1^{V'}(v_i), \ldots, F_k^{V'}(v_i))$$
$$F_j^{V'}(v_i) = dist_{net}(v_i, v_j) \text{ (Preprocessing!)}$$

– $F^{V'}(v)$ kann direkt zur Berechnung von $h(v)$ verwendet werden, denn es gilt $\forall\, v_i, v_j \in V$: $dist_{L_\infty}(F^{V'}(v_i), F^{V'}(v_j)) \leq dist_{net}(v_i, v_j)$

– Beweis

Die Netzwerkdistanz $dist_{net}(v_i, v_j)$ ist transitiv, deshalb gilt die folgende Aussage:
$dist_{L_\infty}(F^{V'}(v_i), F^{V'}(v_j)) =$
$= \max_{l=1..k} | F_l^{V'}(v_i) - F_l^{V'}(v_j) |$
$\leq dist_{net}(v_i, v_j)$



– Geringerer Suchraum als bei Verwendung der euklidischen Distanz

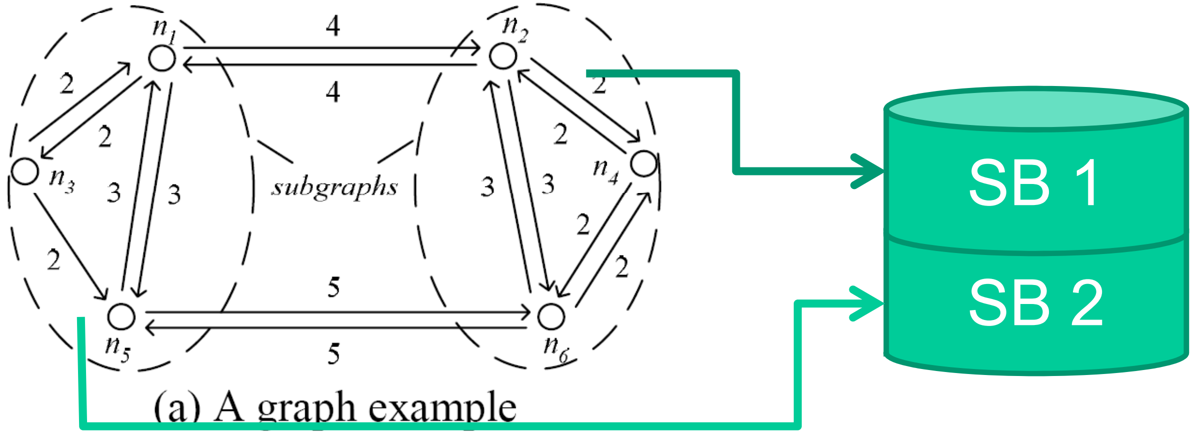
4.3.3 Effiziente Organisation des Straßen-Netzwerkgraphen

– 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

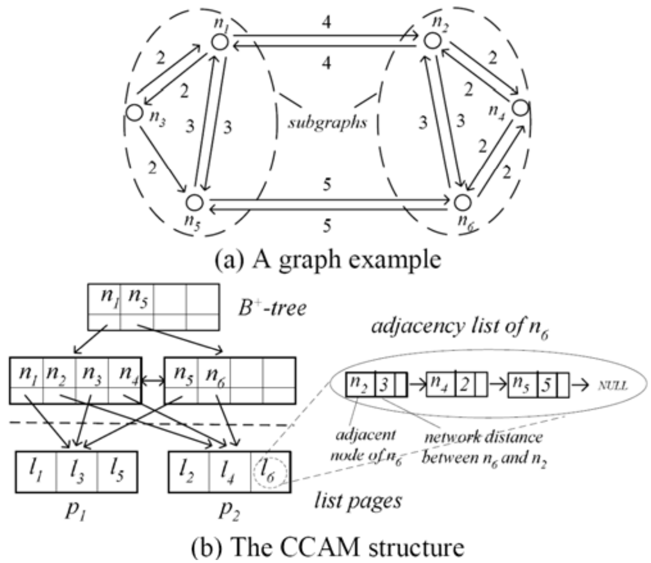
– Idee: Gruppiere Listen von adjazenten Knoten in gleichen Speicherblöcken (Seiten)



(a) A graph example

– 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

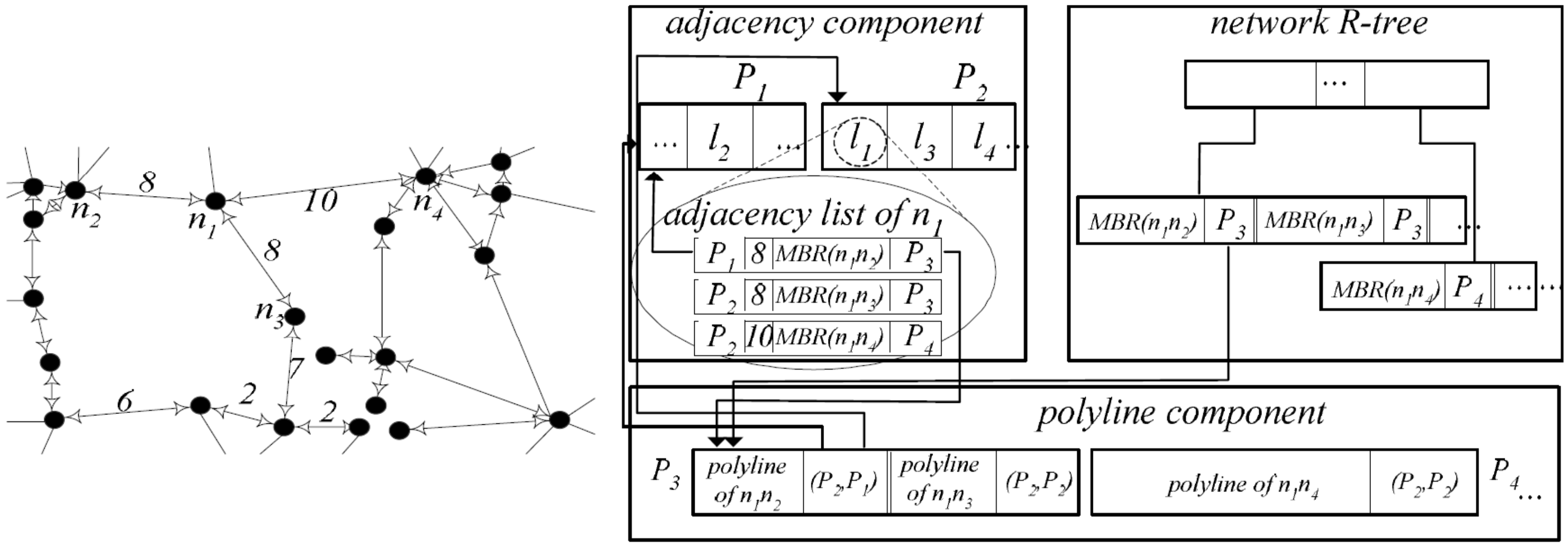


(a) A graph example

(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*

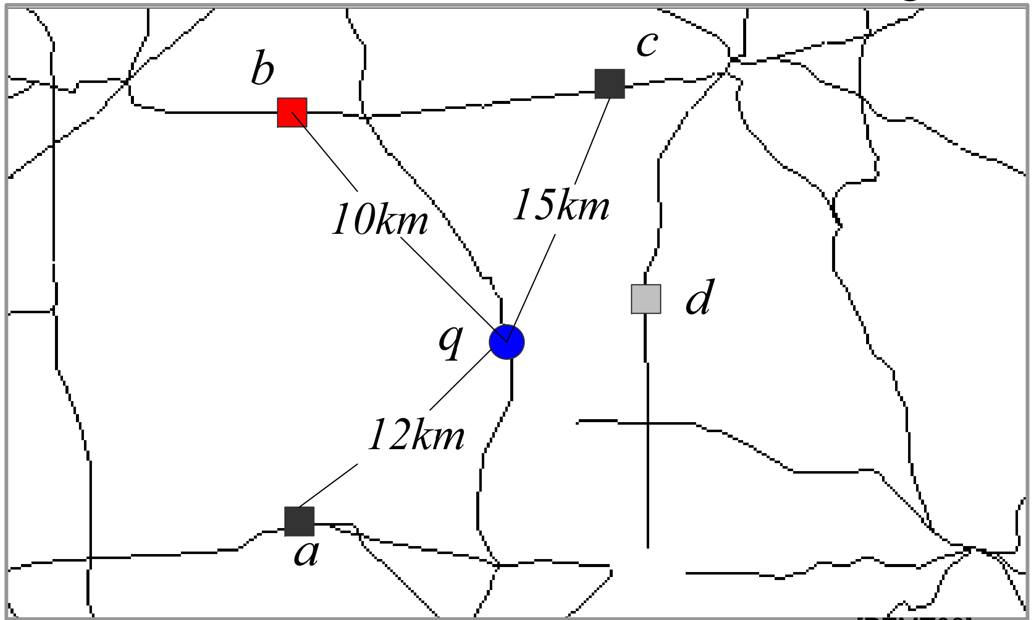
4.4 Räumliche Anfragen in Straßennetzwerken

4.4.1 Bereichsanfragen

– Ziel: Bestimme alle Objekte in einem bestimmten Umkreis unter Beachtung des unterliegenden 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.

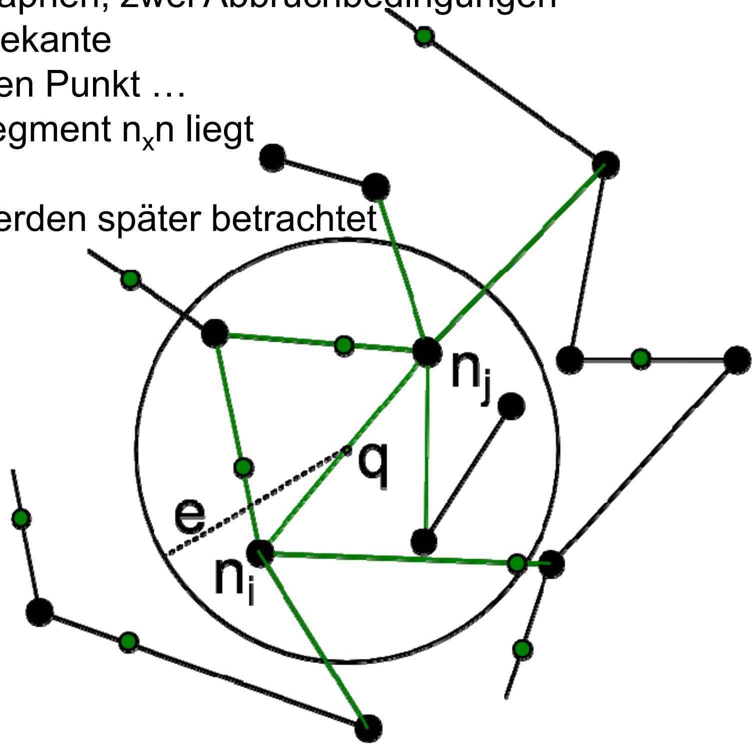
   Filter: p liegt in MBR(k), Beschleunigung über Sortierung von S‘ bzgl. einer Dimemsion

   Refinement: p liegt auf k

– Durch dieses Vorgehen muss der Netzwerkgraph nur einmal durchlaufen werden

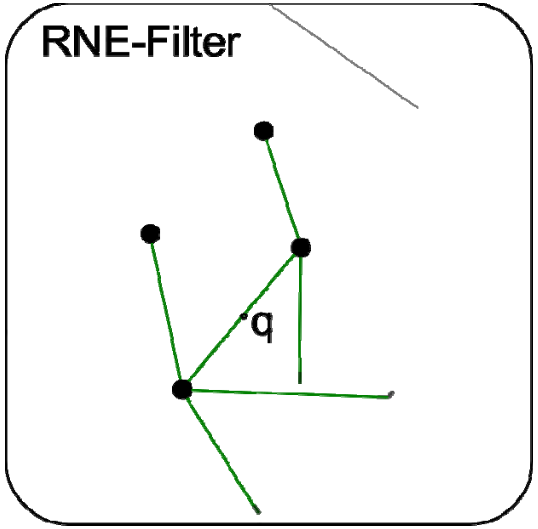
- Algorithmus

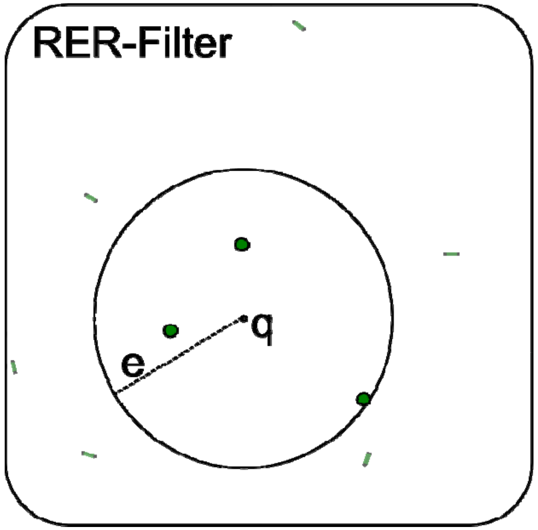
RER(q, e):
result = Ø
S‘ =Euclidean-range(q,e) //S‘ gibt alle Kandidaten mit euklidischer Distanz e zu q zurück
$n_i n_j$=find_segment(q) //Hole das Segment des Netzwerkgraphen, auf dem q liegt
Q=<(n_i,$d_N(q,n_i)$), (n_j,$d_N(q,n_j)$)> //Q: Prioritätswarteschlange, sortiert nach Netzwerkdistanz zu q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

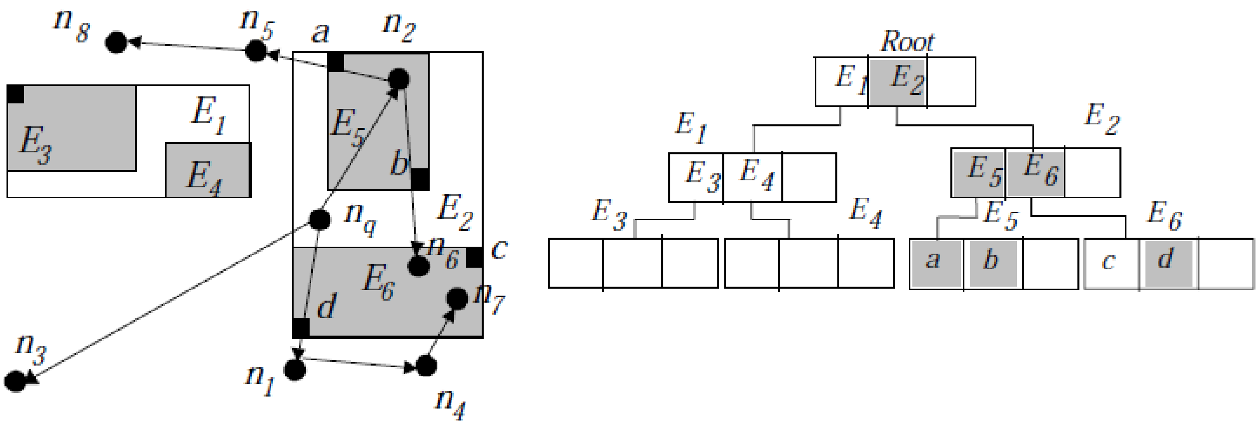
RNE(E_i.node_id, QS_i, result)

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 ∪ $result_{node\_id}$

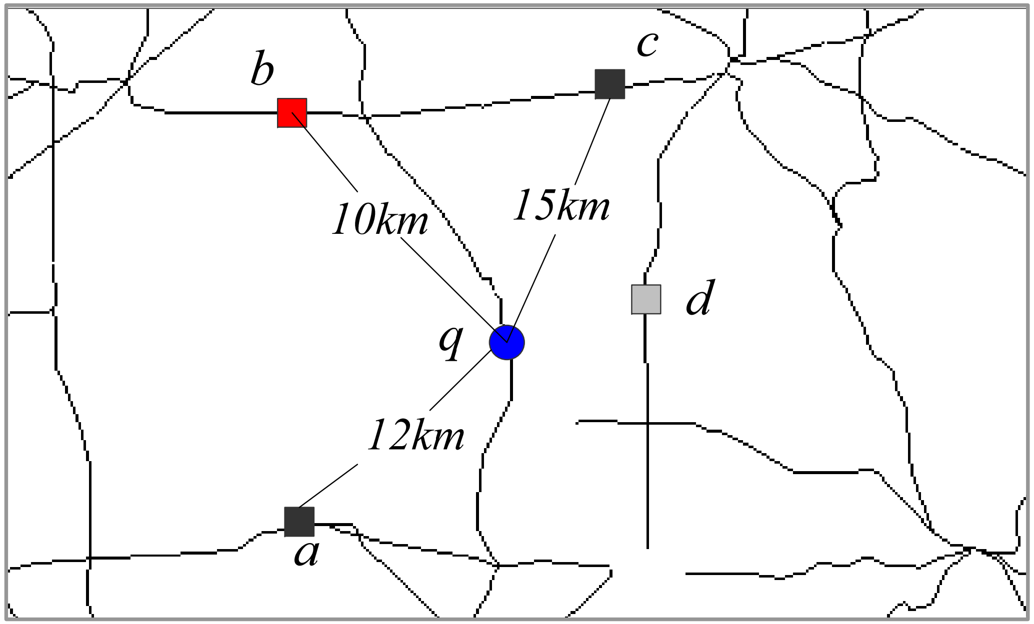


4.4.2 Nächste-Nachbarn-Anfragen

• Ziel: Bestimme den nächsten Nachbarn bzgl. der zurückzulegenden Strecke.

• Beispiel: „Finde das mir am nächsten gelegene Hotel“

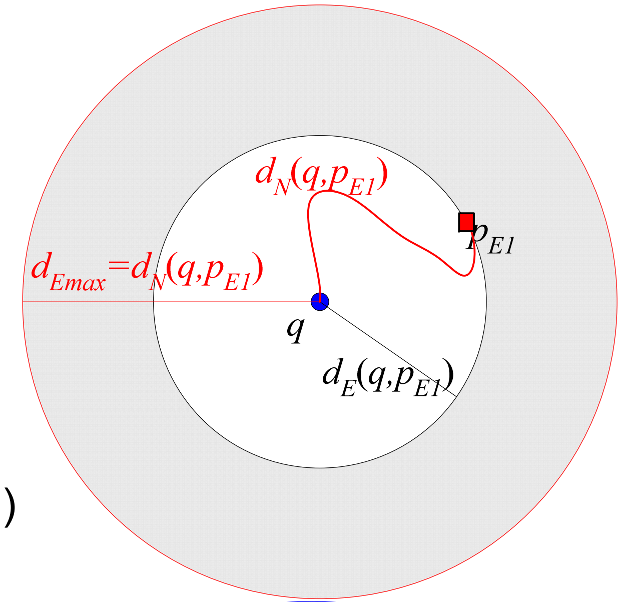
• Entspricht nicht unbedingt der euklidischen Distanz - im Beispiel hat *d* die kürzeste euklidische Distanz, aber eine sehr hohe Netzwerkdistanz.

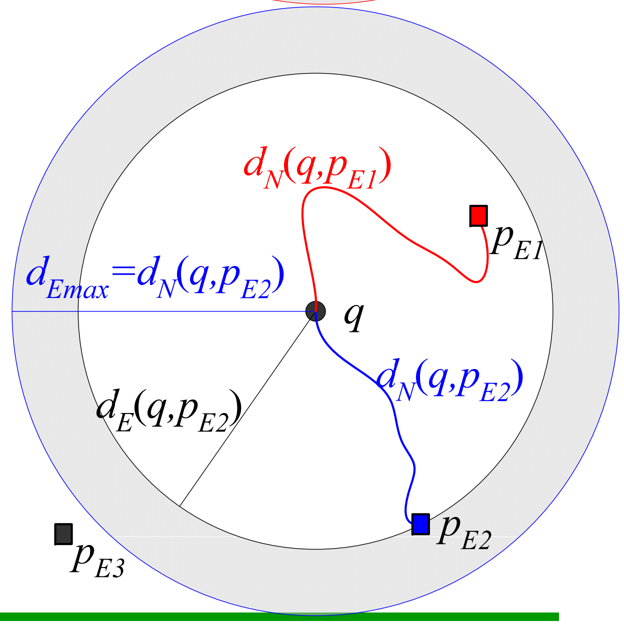


[PZMT03]

NN über Incremental Euclidean Restriction [PZMT03]

1. Berechne 1-NN p_{E1} bzgl. euklidischer Distanz über Standardalgorithmus
2. Berechne Netzwerkdistanz $dist_{net}(q, p_{E1})$
   - pruningdist = $dist_{net}(q, p_{E1})$
   - resultCandidate = p_{E1}
   - Da $dist_{net}(q, p) \geq dist_{L_2}(q,p)$ haben alle möglichen Kandidaten c eine Distanz $dist_{L_2}(q,c) < dist_{net}(q, p_{E1})$ (Pruningkriterium)
3. Solange für i-NN $dist_{L_2}(q,p_{Ei}) \leq$ pruningdist:
   - Suche $p_{E(i+1)}$
   - Falls $dist_{net}(q, p_{Ei}) < dist_{net}(q, \text{resultCandidate})$
     resultCandidate = p_{Ei}
     pruningdist = $dist_{net}(q, p_{Ei})$





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

Algorithm IER (q, k)

/* q is the query point */

$\{p_1,\ldots,p_k\}$=Euclidean_NN(q,k);

for each entity p_i

  $d_N(q,p_i)=compute\_ND(q,p_i)$

sort $\{p_1,\ldots,p_k\}$ in ascending order of $d_N(q,p_i)$

$d_{Emax}= d_N(q, p_k)$

repeat

  $(p,d_E(q, p))$=next_Euclidean_NN(q);

  if ($d_N(q,p)<d_N(q,p_k)$) // p closer than the k^{th} NN

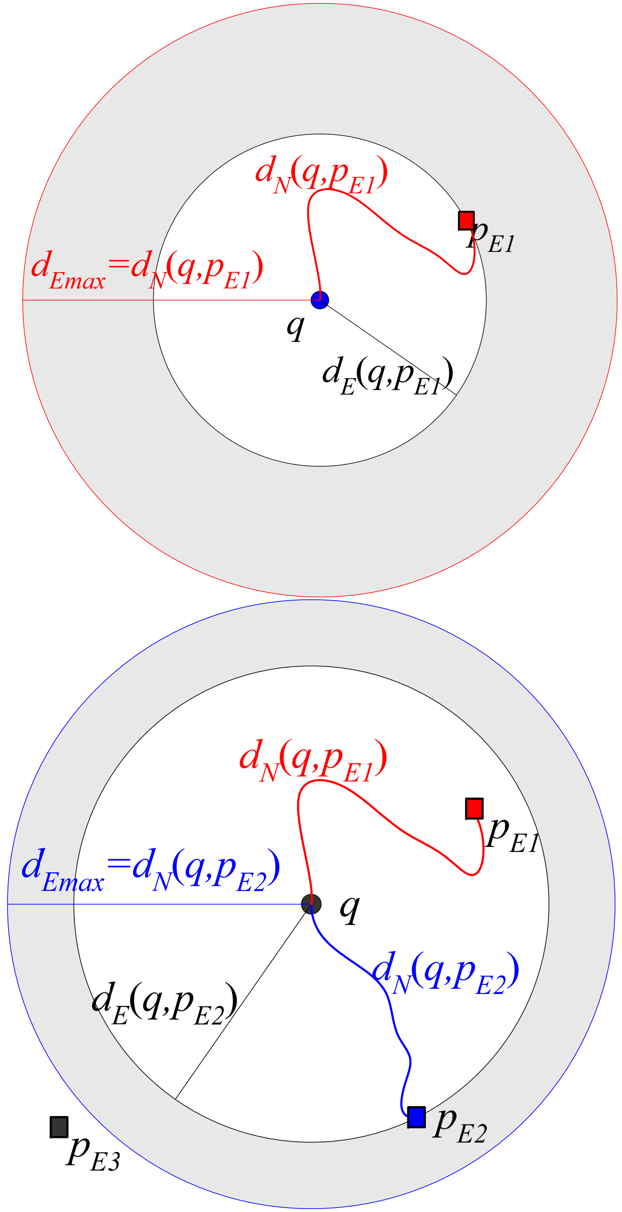
    insert p in $\{p_1,\ldots,p_k\}$ // remove ex-k^{th} NN

    $d_{Emax} = d_N(q, p_k)$

until $d_E(q,p)>d_{Emax}$

End IER

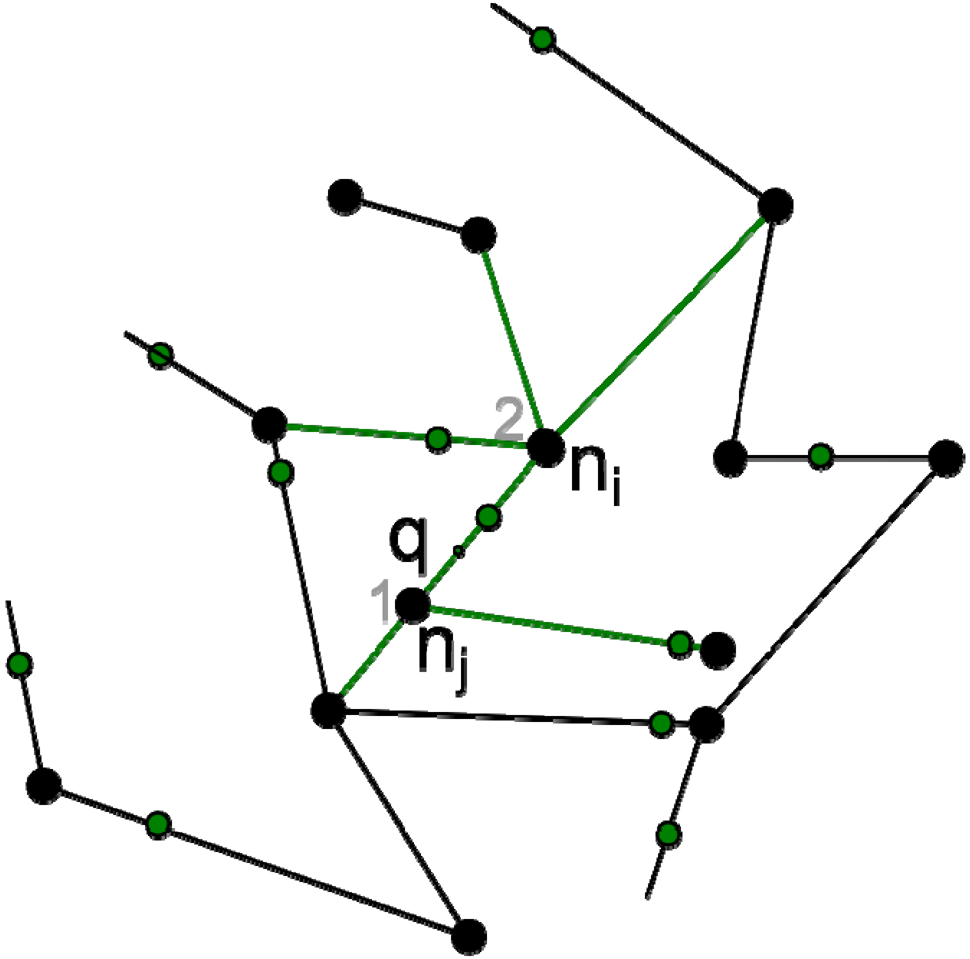
Figure 4.2: Incremental Euclidean Restriction



NN über Incremental Network Expansion [PZMT03]

– Sehr ähnlich zum Dijkstra-Algorithmus
– Das Netzwerk wird entsprechend des Dijkstra-Algorithmus durchsucht, bis der erste NN-Kandidat gefunden wird. Eine Variable speichert die aktuelle NN-Distanz.
– Werden weitere NN-Kandidaten gefunden, wird sowohl die NN-Distanz als auch der aktuelle 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.

k-NN-Suche über IER und INE[PZMT03]



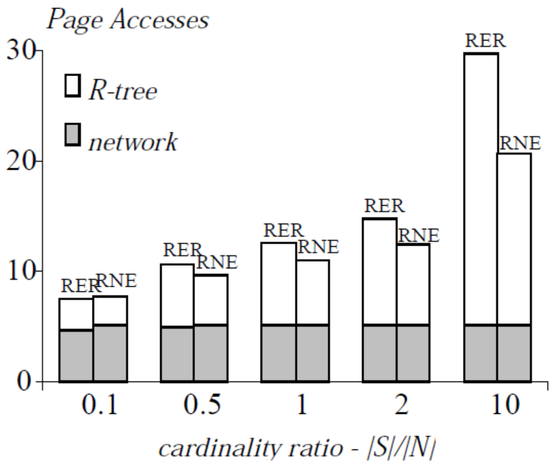
Algorithm INE (q, k)

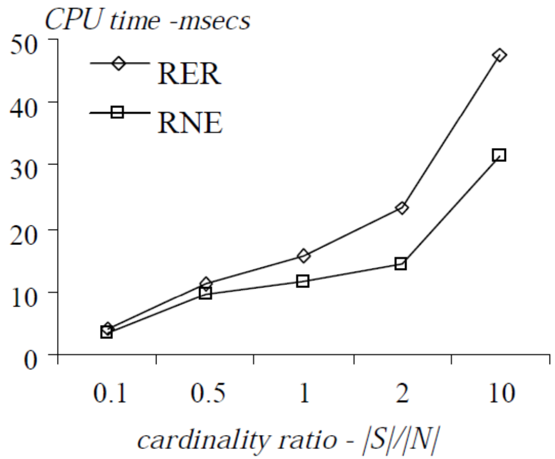
n_in_j=*find_segment*(q)
S_{cover}=*find_entities*(n_in_j); // S_{cover} is the set of entities covered by n_in_j
$\{p_1,\ldots,p_k\}$=the k (network) nearest entities in S_{cover} sorted in ascending order of their network distance (p_m, $p_{m+1}\ldots,p_k$ may be $\varnothing$ if S_{cover} contains $< k$ points)
$d_{Nmax}=d_N(q,p_k)$ // if $p_k=\varnothing$, $d_{Nmax}=\infty$
$Q=<(n_i, d_N(q,n_i)), (n_j, d_N(q,n_j))>$ //sorted on d_N
de-queue the node n in Q with the smallest $d_N(q,n)$
while ($d_N(q,n)<d_{Nmax}$)
  for each non-visited adjacent node n_x of n
    S_{cover}=*find_entities*(n_xn);
    update $\{p_1,\ldots,p_k\}$ from $\{p_1,\ldots,p_k\}\cup S_{cover}$
    $d_{Nmax}=d_N(q,p_k)$
    en-queue ($n_x,d_N(q,n_x)$)
  de-queue the next node n in Q
End INE

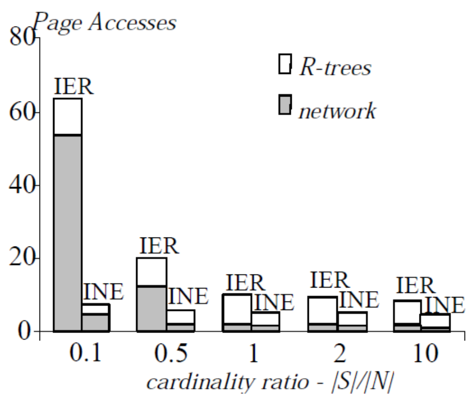
Figure 4.4: Incremental Network Expansion

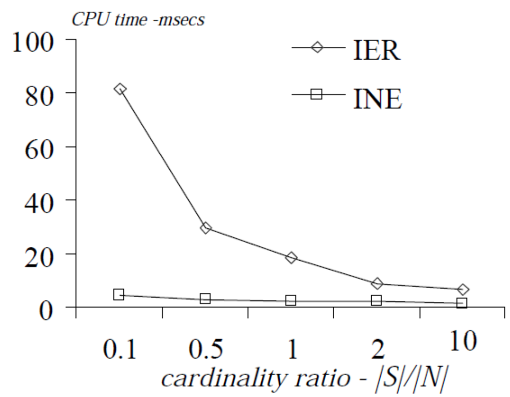
• Performanz von RER, RNE, IER, INE

– Geprüft wurden
  - Seitenzugriffe (HDD)
  - CPU-Zeit









– Network Expansion meist besser als Euclidean Restriction

4.4.3 Routen-Suche in Multiattributs-Verkehrsnetzwerken

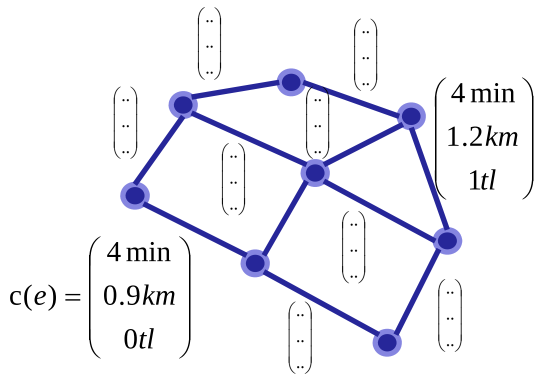
– Gegeben:

- 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 <-> Kreuzungen, Sackgassen
  – $E \in V \times V$: Menge von Kanten <-> Strassensegmente zwischen Kreuzungen
  – c: $E \rightarrow IR^d$: Kostenfunktion <-> Kostenvektor für

  Kanten mit d Kostenattributen
- Annahmen:
  – Keine negativen Kosten (Warum?)
  – Alle Kostenattribute sind additiv

  (Gilt für Kosten eines Pfades (Route))



- Route:
  - p = <$v_1,\ldots,v_n$> 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

Zielknoten Z

Startknoten S

:= 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)

Zielknoten Z

Startknoten S

Route A
7 km, 5 Minuten

Route B
6 km, 7 Minuten

Welcher Pfad ist optimal A oder B?

- Kostenattribute schwierig zu vergleichen:
  #Ampeln ←→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

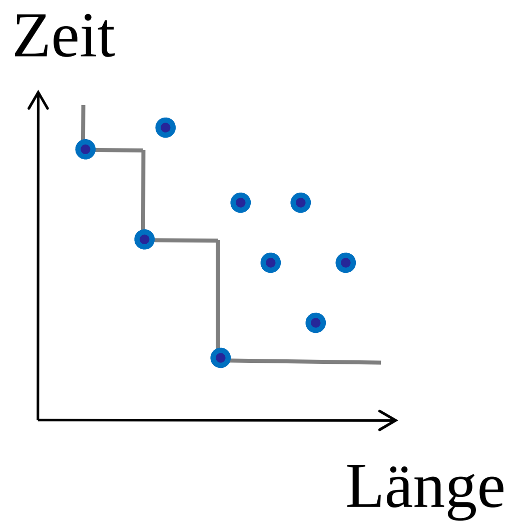
– Anfrage:

- Finde die Menge RS alle Pfade zwischen S und Z, sodaß für jeden Pfad p∈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

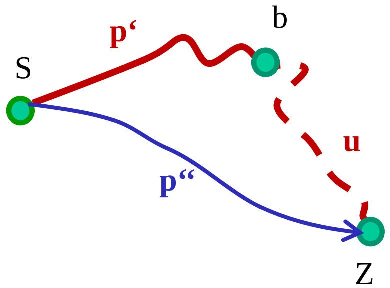


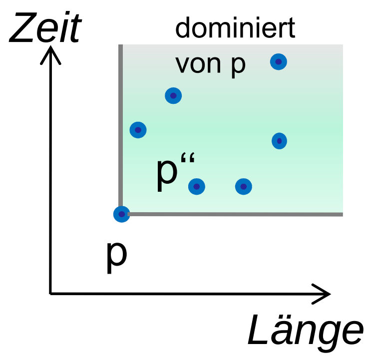
– Probleme:

- 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'=<S,…,b>,

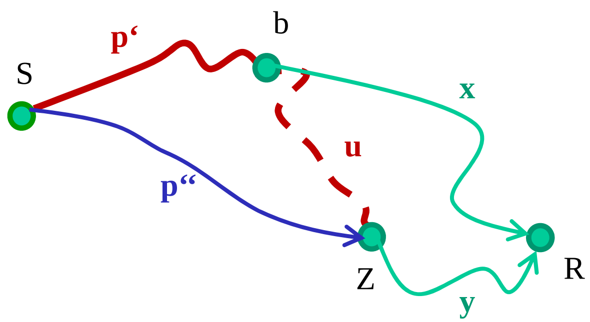




- Wie kann man abschätzen ob p' Teil einer Skylineroute ist? => Pruningkriterien

– Pruningkriterium I: Pruning mittels Vorwärtsabschätzung

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



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

- Funktioniert solange die Kostenabschätzung die untere Schrankeneigenschaft erfüllt

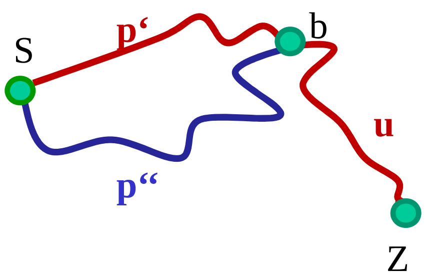
– Pruningkriterium II: Pruning unter Verwendung der Monotonieeigenschaft der Dominanzbeziehung

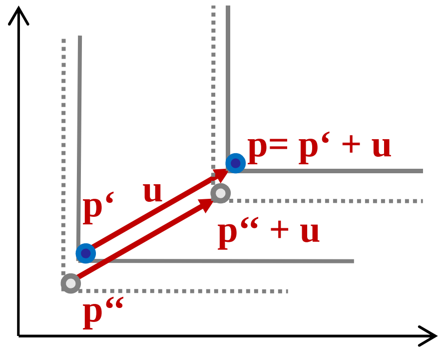
Theorem:

Gegeben sei eine Skyline Route p=<S,…,b,…,Z>, d.h. p wird von keiner anderen Route zwischen S umd Z dominiert. Dann wird jede Teilroute p’=<S,…,b> von p von keiner anderen Teilroute p’’=<S,…,b> dominiert.

Beweis:

Annahme es gibt eine Route p’’=<S,…,b>, die die Teilroute p’=<S,…,b>≠p’’ von p dominiert. Dann würde die Erweiterung von p’’ über die Teilroute u=<b,…,Z> von p zu einer Route führen, die die Route p dominiert => Wiederspruch zur Annahme daß p Skylineroute ist.





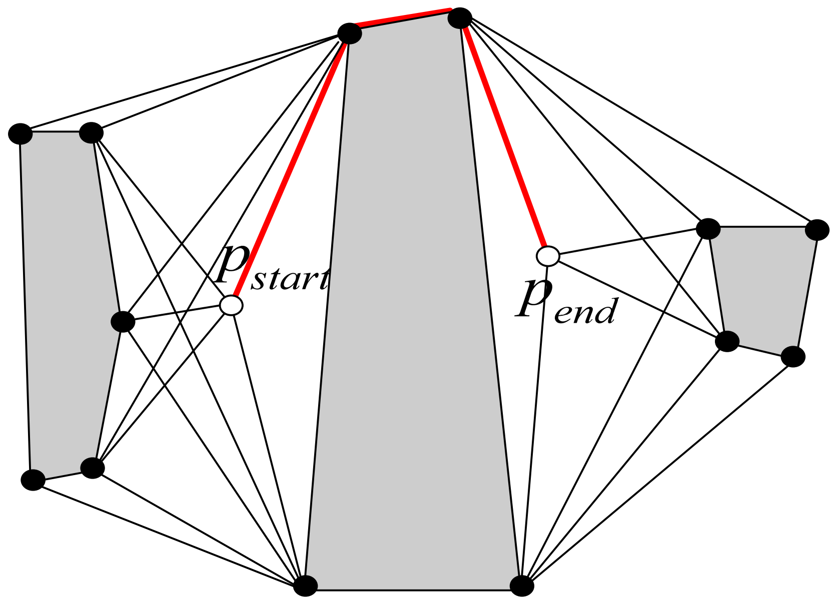
– Skyline-Route-Algorithmus:

- Idee: wie A*-Suche, aber verwalte bei jedem Knoten n eine Skyline (Skyline bzgl. aller Pfade p'=<S,…,n>)
- 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$\in$cand
   if $\exists$p$\in$Z.SkR, p dominiert cost_{est}(c), **then** lösche c aus cand;
   else
      update(c.last_node.SkylineRoutes,c);
      queue.insert(c);
report(Z.SkR);

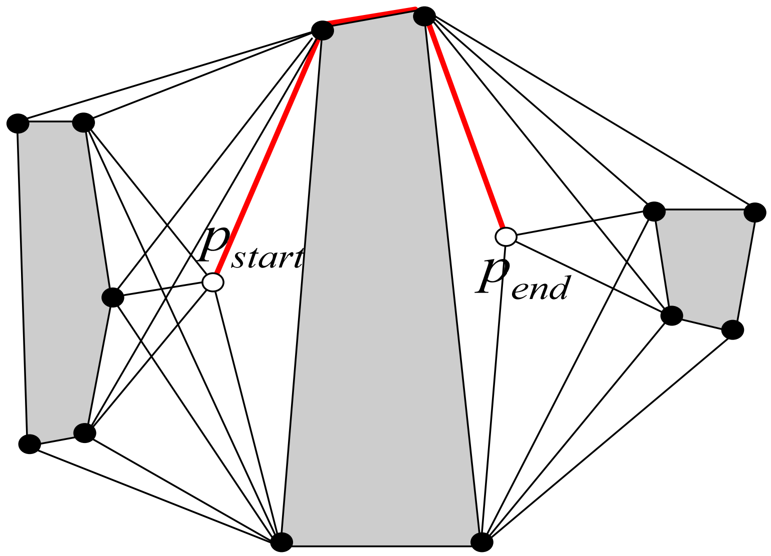
4.4.4 Pfadsuche im euklidischen Raum mit Hindernissen[ZPMZ04]

- Ziel: Anfragebearbeitung (z.B. NN-Anfragen) im euklidischen Raum bei Anwesenheit von Hinernissen (Häuser, Seen, …)
- Die zurückgelegte Strecke erhöht sich dabei wie in Graphen
- Zur Anfragebearbeitung können Graphalgorithmen verwendet werden



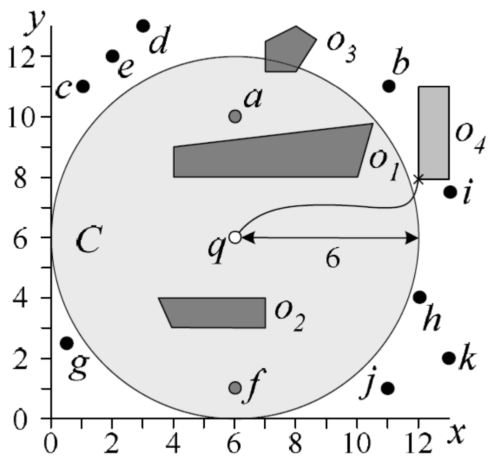
– Sichtbarkeitsgraph

- 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)
- Kanten: Verbinden Knoten, dürfen kein Hindernis kreuzen
- Berechnung des gesamten Sichtbarkeitsgraphen zu aufwändig
- Deshalb Reduktion auf relevante Teilgraphen, bei denen nur relevante Hindernisse betrachtet werden.

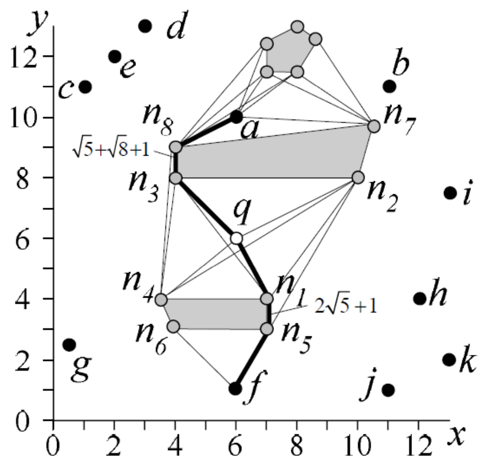


• Bereichsanfrage mit Hindernissen

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



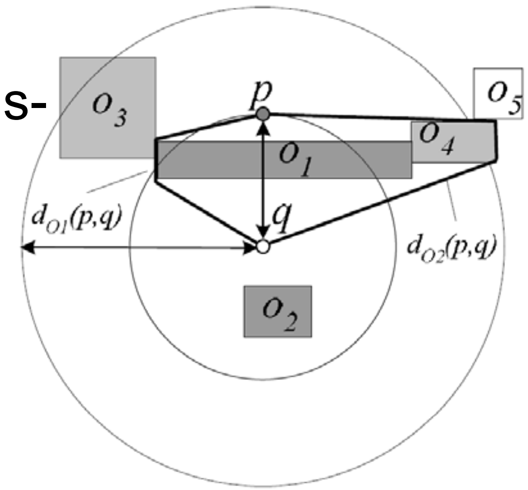
(a) Obstacle range query



(b) Local visibility graph

• NN-Anfrage mit Hindernissen

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



[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.

[KKKRS08]: 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.

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