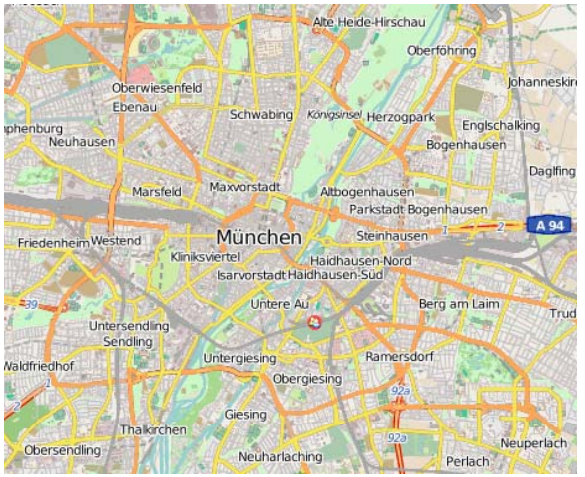
Kapitel 4
Anfragemethoden auf Verkehrsnetzwerke

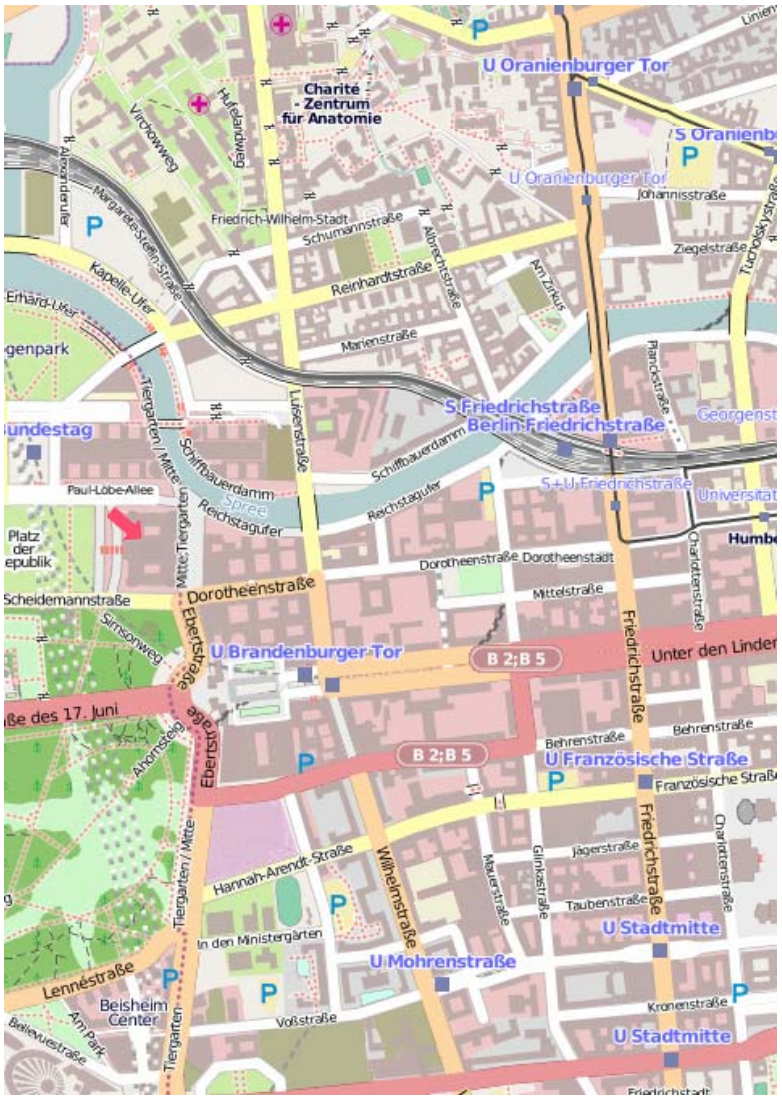
Skript zur Vorlesung: Neue Trends zur Suche in modernen Datenbanksystemen
Wintersemester 2012/13, LMU München

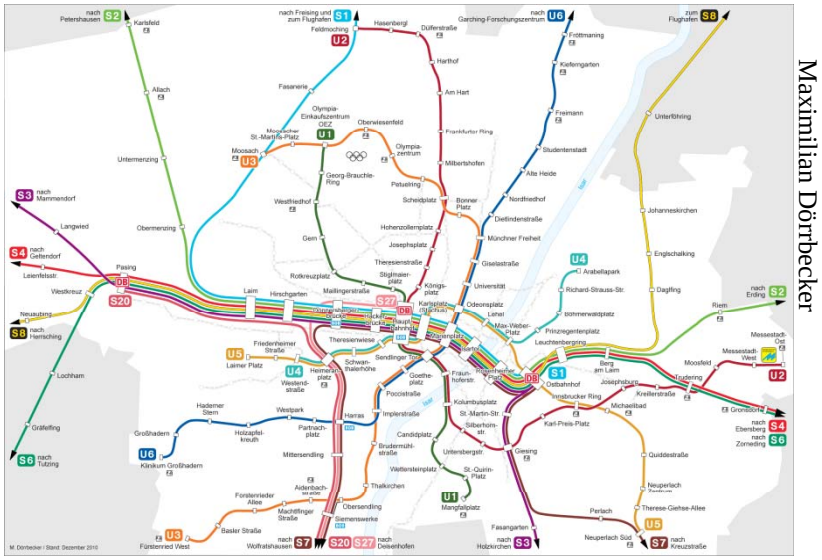
4.1 Motivation

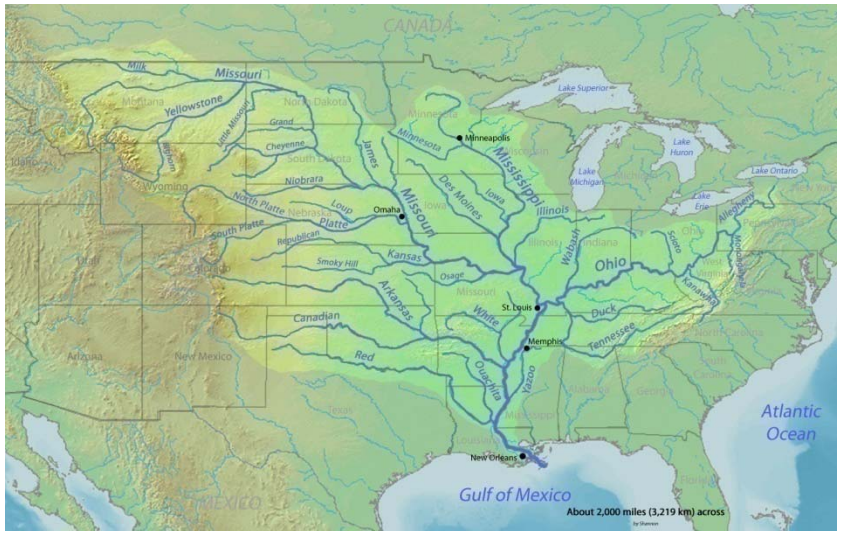
- Bisher: Distanz zwischen zwei Punkten über einfacher Distanzmaße wie z.B. die Euklidische Distanz berechnet.
- In vielen Anwendungen, z.B. im Straßenverkehr, existieren Einschränkungen bzgl. möglicher Bewegungsrichtungen.
- In Verkehrsnetzwerken wird als Distanz zwischen zwei Punkten i.d.R. die Länge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

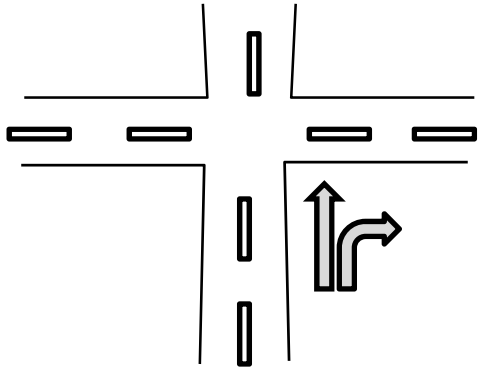
4.2.1 Konzeptuelles Modell

- 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*

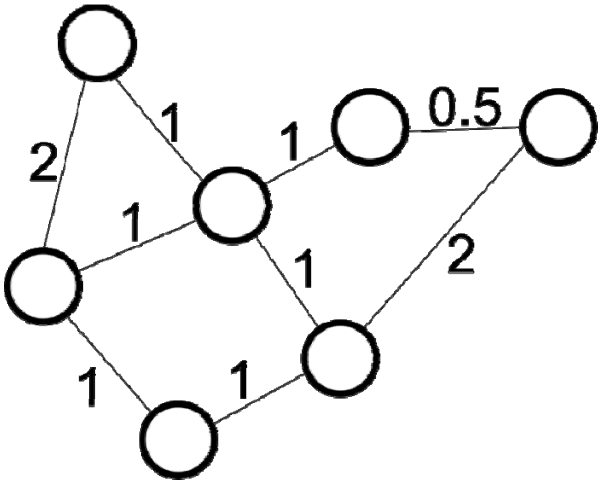
- *Repräsentation 2:*
  - *Knoten $v_i \in V$: Straßen*
  - *Kanten $e_i \in E$: Kreuzungen zwischen Straßen*

Wann sinnvoll?

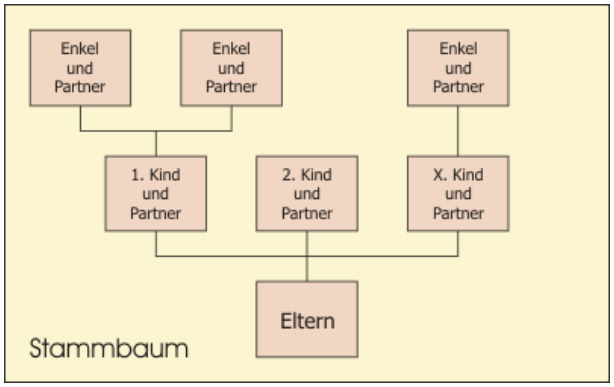


Mit Rep. 1 möglich?

- Kanten haben Gewichte (Kosten), z. B.
  - Reellwertig
    - Entfernung zwischen zwei Knoten
    - Fahrtdauer
    - Benzinverbrauch
    - gewichtete Kombination mehrerer Merkmale
  - Vektorielle Kombination mehrerer reellwertiger Gewichte [KRS10]



- 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)

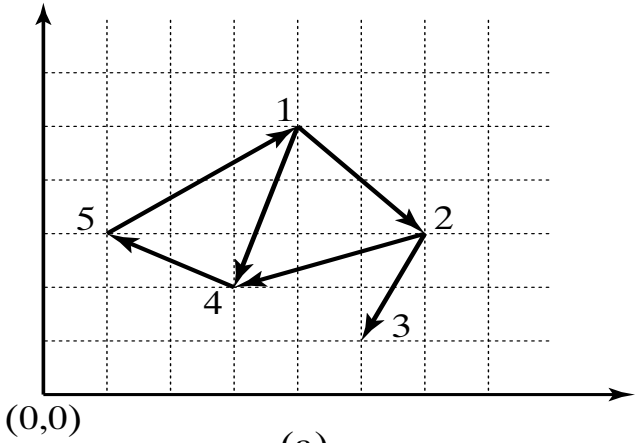
- Vertex, Attribute:
  - label
  - isVisited
  - location (räumliche Graphen)
- DirectedEdge, Attribute:
  - startNode
  - endNode
  - label
- Graph, Attribute:
  - Set<Vertex>
  - Set<DirectedEdge>
- Path: Attribute
  - Sequence<Vertex>

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

– Beispiel



(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

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

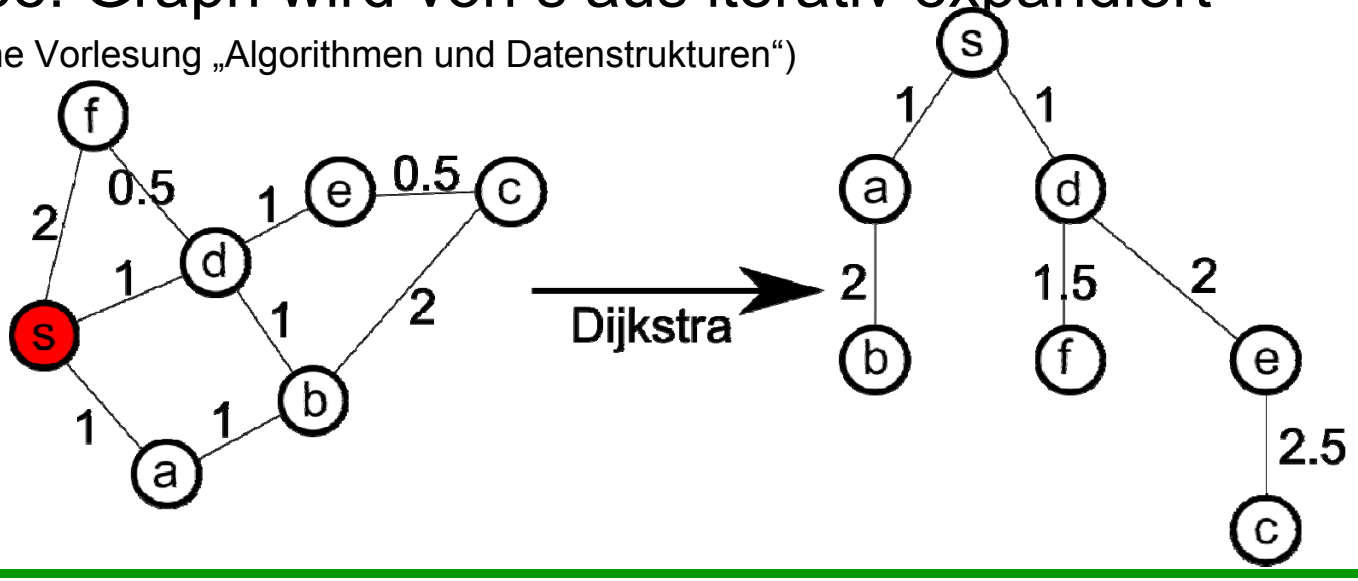
4.3 Basisalgorithmen auf Straßennetzen

- 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
- Die berechneten Distanzen $dist_{net}(v_i, v_j)$ können dann wiederum zur Bearbeitung komplexerer Anfragen auf Straßennetzen verwendet werden
  - Beispiel: Bereichsanfragen und Nächste-Nachbarn-Anfragen
  - $dist_{net}(v_i, v_j)$ im Gegensatz zur euklidischen Distanz nicht unbedingt symmetrisch, d.h. $dist_{net}(v_i, v_j) \neq dist_{net}(v_j, v_i)$ (z.B. Einbahnstraßen)

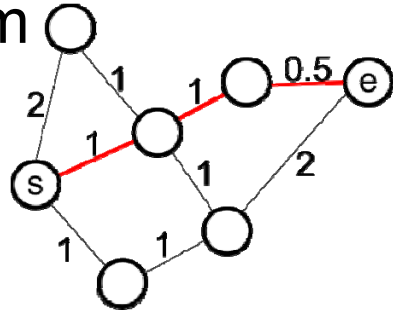
4.3.1 Single-Source Shortest Path – Dijkstra

- Gegeben:
  - (gerichteter) Graph G=(V,E)
  - Kanten haben nicht-negative, reelle Gewichte
- Gesucht:
  - Kürzester Pfad von einem Startknoten s zu allen erreichbaren verbleibenden Knoten
- Idee: Graph wird von s aus iterativ expandiert

(siehe Vorlesung „Algorithmen und Datenstrukturen“)



4.3.2 Der A*-Algorithmus [HNR68]

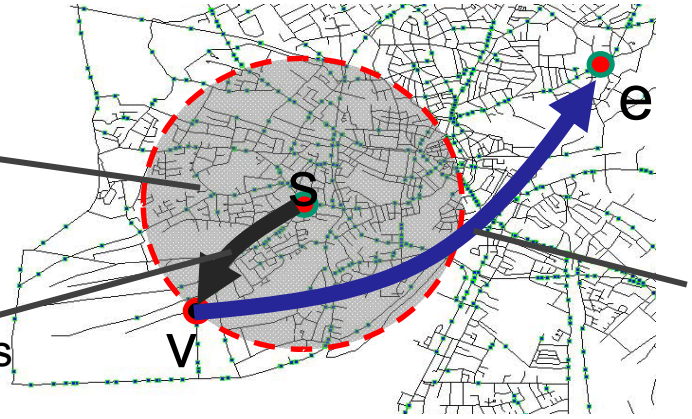


- Ziel: Bestimme den kürzesten Pfad von einem Startknoten s zu einem Zielknoten e
- Nah verwandt mit dem Dijkstra-Algorithmus
- Idee:
  - Verwendung einer Heuristik zur Vorwärtsabschätzung zum Zielknoten e für die Abschätzung der gesamten Pfadkosten während der Expansion:

Netzwerkgraph:

Expandierung des Suchraums

Aktuell erreichte Distanz von Knoten s zu einem Knoten v

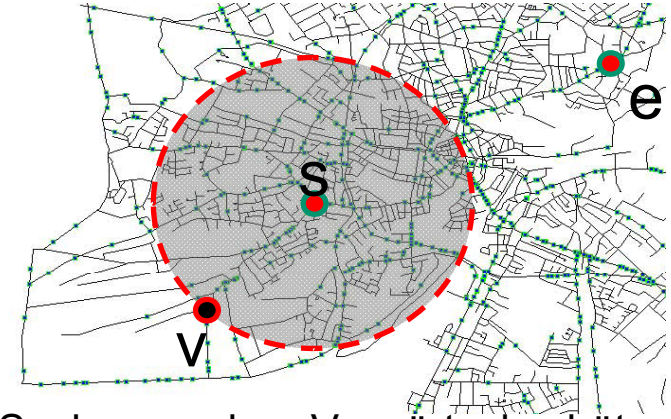


Abgeschätzte Distanz von Knoten v zum Zielknoten e

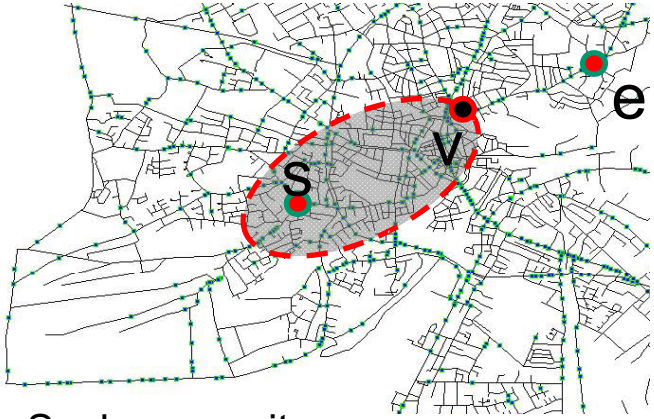
- Best-first-basierte Expansion des Graphen unter Berücksichtigung der Abgeschätzten Gesamtdistanz zwischen s und e, d.h. expandiere nur Knoten v über die der (kürzeste) Pfad von s nach e die Gesamtdistanz optimiert.

=> der Graph wird eher in Richtung des Zielknotens expandiert

$\Rightarrow$ 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 = 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 ϵ neighbors(x)
            if y ϵ 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: $f(v) \leq dist_{net}(s,v)+dist_{net}(v,e)$.
- Je besser die Schätzung von h(v) (d.h. je größer h(v)), desto weniger Knote
- Bei Verwendung von $h(v) = 0$, $\forall v \in V$, verhält sich A* wie Dijkstra.

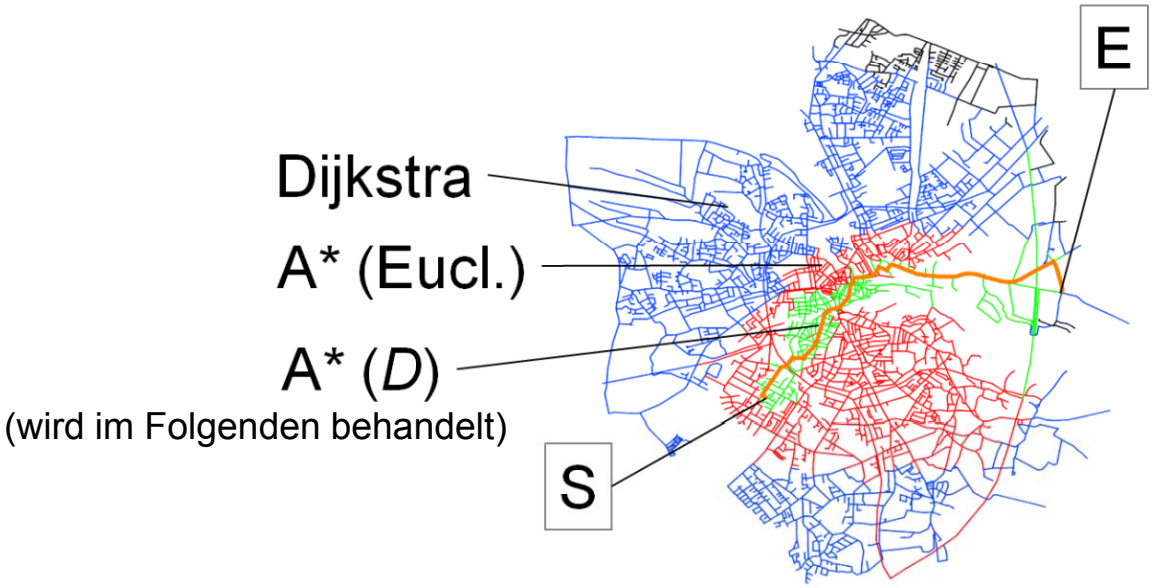
Heuristiken für A: Setze h(v) = 0*

- 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 extrem groß.

Heuristiken für A: Setzte $h(v) = dist_{L2}(v,e)$*

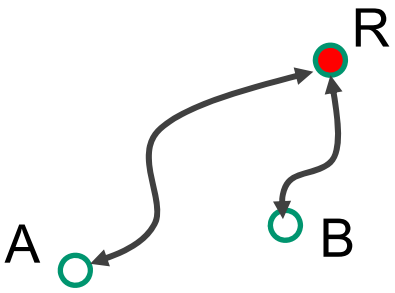
- Verwendet stets die geringste praktisch mögliche Distanz.
- Reale Distanz ist normalerweise weit größer als euklidische Distanz.
- Kleinerer Suchraum als bei Dijkstra, aber immer noch sub-optimale Schätzung.

– Vergleich der Heuristiken:

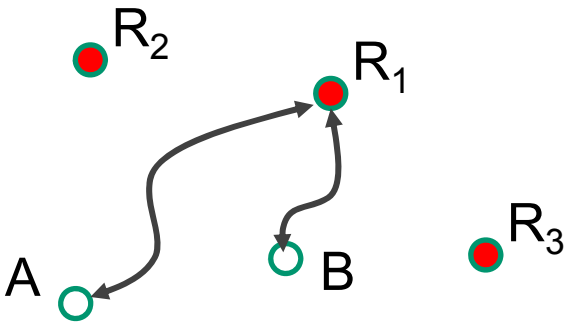


Heuristiken für A: Graph Embedding (D-Distanz)* [KKKRS08]

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



$$dist_{net}(A,B) \geq |dist_{net}(A,R) - dist_{net}(B,R)|$$

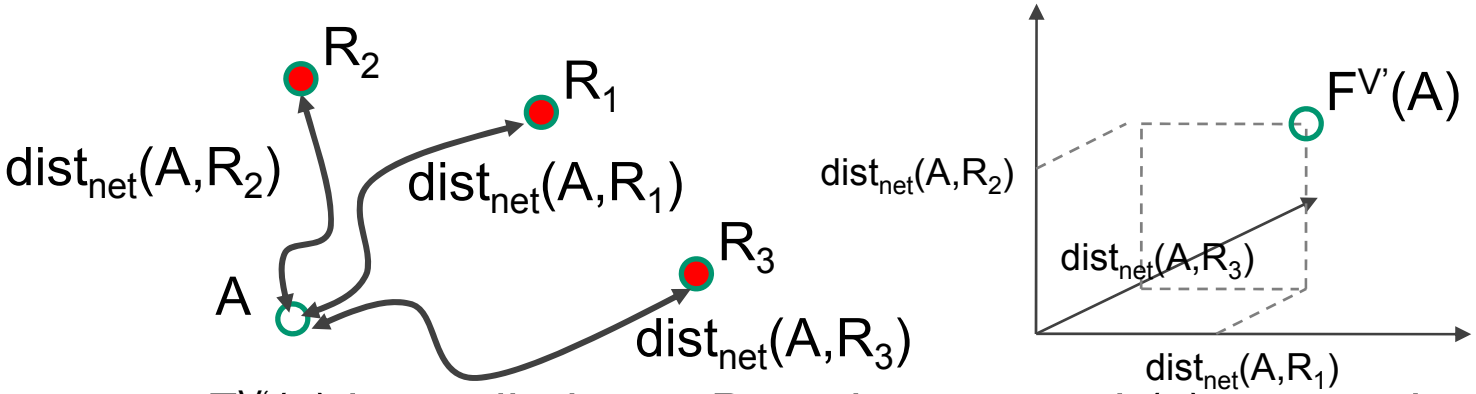


Bei k Refernzknoten $R_{1,..,k}$:
$dist_{net}(A,B) \geq max_{i=0..k}(|dist_{net}(A,R_i) - dist_{net}(B,R_i)|)$

Führt zur besseren Abschätzung als mit nur einem Refernzknoten.

- 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:

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



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