Karlsruher Institut für Technologie

# Algorithmen für Routenplanung

16. Vorlesung, Sommersemester 2016

Valentin Buchhold | 20. Juni 2016

Institut für Theoretische Informatik · Algorithmik · Prof. Dr. Dorothea Wagner

# Prüfungstermine

19. August

16. September

7. Oktober

Anmeldung per E-Mail ans Sekretariat
(lilian.beckert@kit.edu)

# Anfrageszenarien

## Punkt-zu-Punkt

- zwei Punkte $\rightarrow$ kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken
- HubLabels 10Mx schneller

# Anfrageszenarien

## Punkt-zu-Punkt

- zwei Punkte $\rightarrow$ kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken
- HubLabels 10Mx schneller

## One-to-All

- ein Knoten $\rightarrow$ Distanzen zu allen Knoten
- wird für Vorberechnung benutzt
- PHAST 500x schneller (auf GPU)
- nutzt Hardware aus

# Anfrageszenarien

## Punkt-zu-Punkt

- zwei Punkte $\rightarrow$ kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken
- HubLabels 10Mx schneller

## One-to-Many

- ein (variierender Knoten) und eine (feste) Menge $\rightarrow$ Distanz zu allen Knoten in der Menge
- wichtig für POI

## One-to-All

- ein Knoten $\rightarrow$ Distanzen zu allen Knoten
- wird für Vorberechnung benutzt
- PHAST 500x schneller (auf GPU)
- nutzt Hardware aus

# Anfrageszenarien

## Punkt-zu-Punkt

- zwei Punkte $\rightarrow$ kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken
- HubLabels 10Mx schneller

## One-to-All

- ein Knoten $\rightarrow$ Distanzen zu allen Knoten
- wird für Vorberechnung benutzt
- PHAST 500x schneller (auf GPU)
- nutzt Hardware aus

## One-to-Many

- ein (variierender Knoten) und eine (feste) Menge $\rightarrow$ Distanz zu allen Knoten in der Menge
- wichtig für POI

## Many-to-Many

- zwei Mengen $\rightarrow$ Distanztabelle
- wichtig für Vehicle Routing

# Das One-to-All Problem

**Problem Definition:**

- Eingabe: ein Graph $G = (V, E)$ und ein Knoten $s$
- Ausgabe: Distanz von $s$ zu allen $v \in V$
- Annahme: viele Anfragen, Vorberechnung zahlt sich aus

# Das One-to-All Problem

**Problem Definition:**

- Eingabe: ein Graph $G = (V, E)$ und ein Knoten $s$
- Ausgabe: Distanz von $s$ zu allen $v \in V$
- Annahme: viele Anfragen, Vorberechnung zahlt sich aus

**Lösungen:**

- Dijkstras Algorithmus (ohne Stoppkriterium)
  $\Rightarrow$ Performance nicht praktikabel auf großen Instanzen

# Das One-to-All Problem

**Problem Definition:**

- Eingabe: ein Graph $G = (V, E)$ und ein Knoten $s$
- Ausgabe: Distanz von $s$ zu allen $v \in V$
- Annahme: viele Anfragen, Vorberechnung zahlt sich aus

**Lösungen:**

- Dijkstras Algorithmus (ohne Stoppkriterium)
  $\Rightarrow$ Performance nicht praktikabel auf großen Instanzen
- Bisher: erweitere Contraction Hierarchies
  $\Rightarrow$ PHAST-Algorithmus

# Das One-to-All Problem

**Problem Definition:**

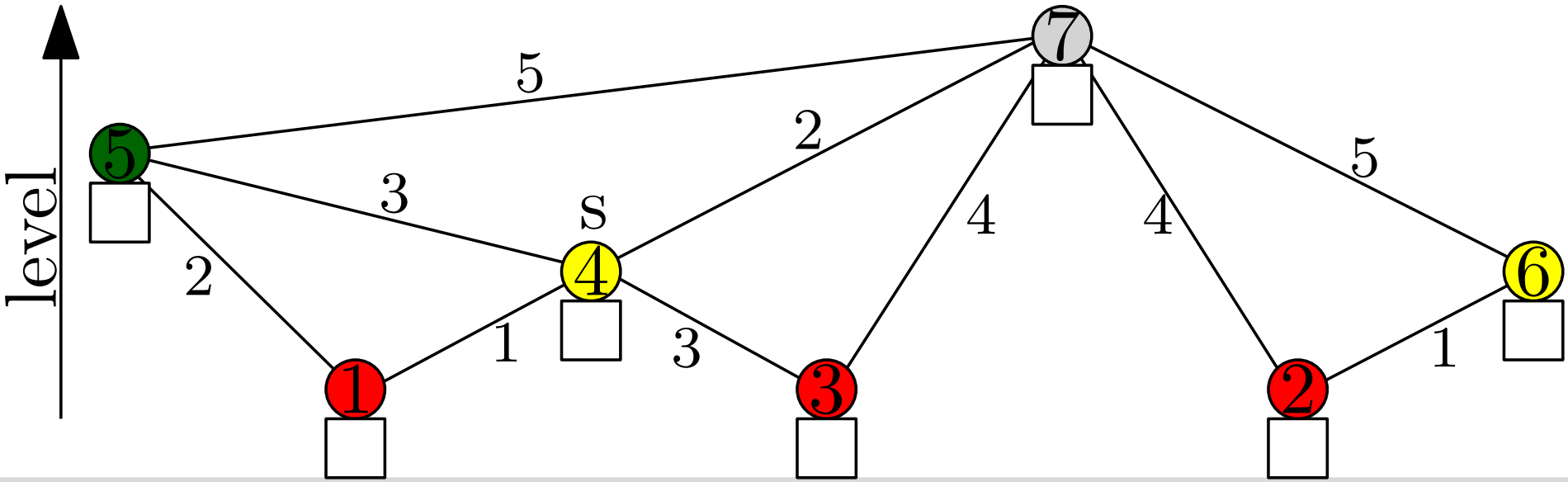
- Eingabe: ein Graph $G = (V, E)$ und ein Knoten $s$
- Ausgabe: Distanz von $s$ zu allen $v \in V$
- Annahme: viele Anfragen, Vorberechnung zahlt sich aus

**Lösungen:**

- Dijkstras Algorithmus (ohne Stoppkriterium)
  $\Rightarrow$ Performance nicht praktikabel auf großen Instanzen
- Bisher: erweitere Contraction Hierarchies
  $\Rightarrow$ PHAST-Algorithmus
- Heute: erweitere Customizable Route Planning
  $\Rightarrow$ GRASP-Algorithmus

# Wiederholung: PHAST

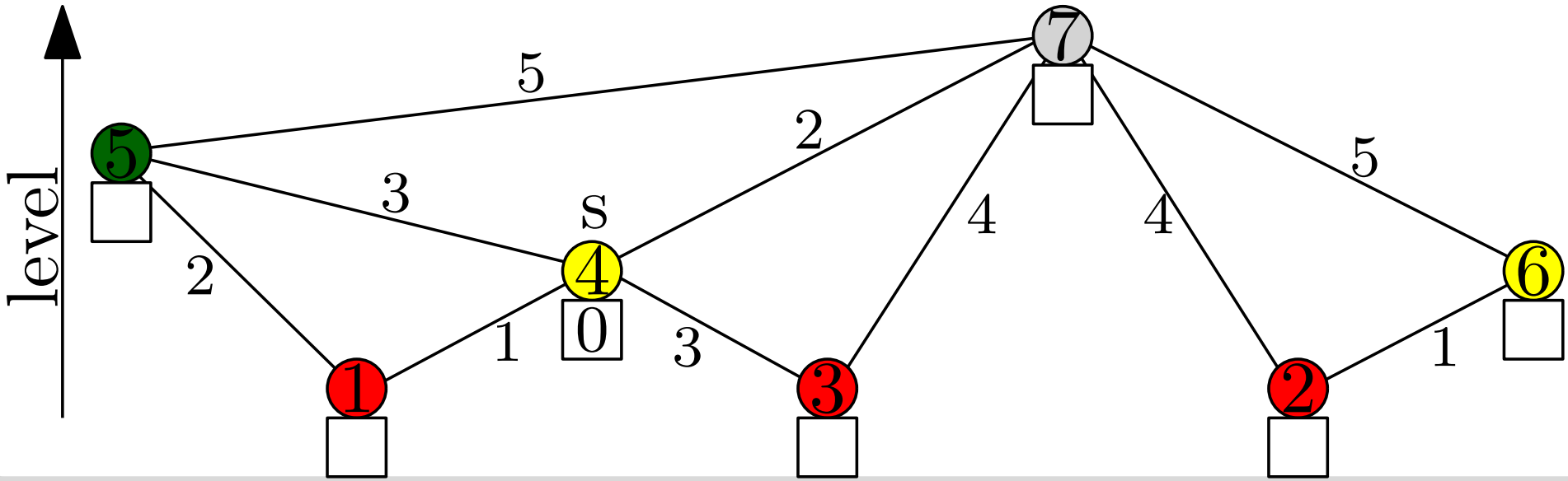
**one-to-all Suche von $s$:**



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

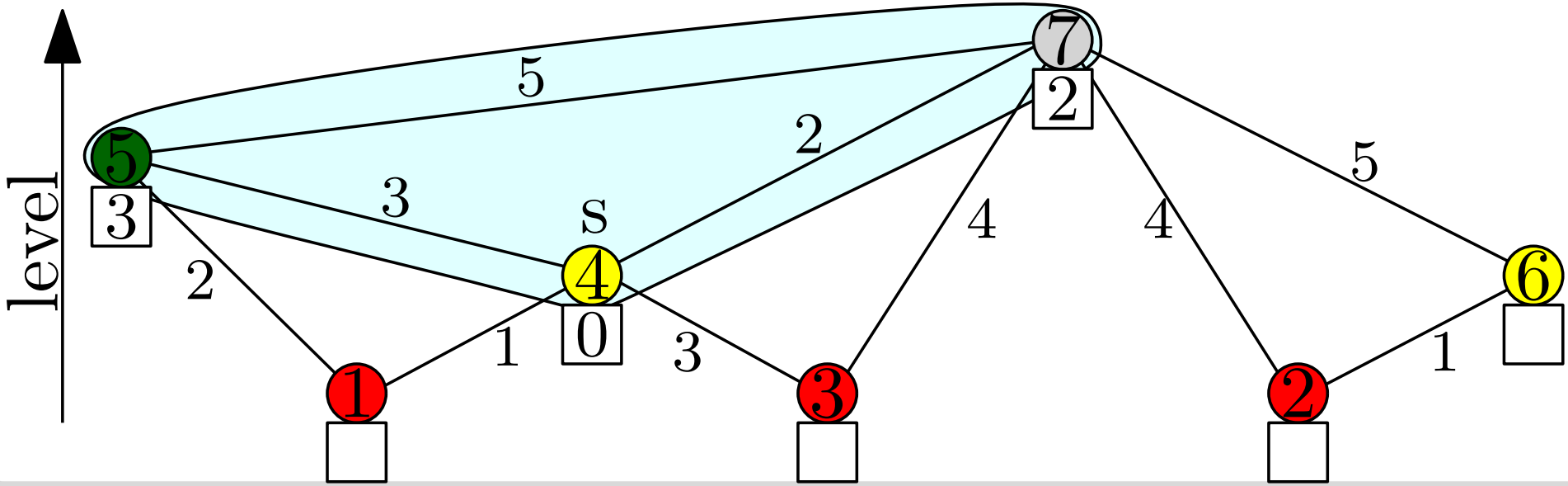
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)



# Wiederholung: PHAST

**one-to-all Suche von *s*:**

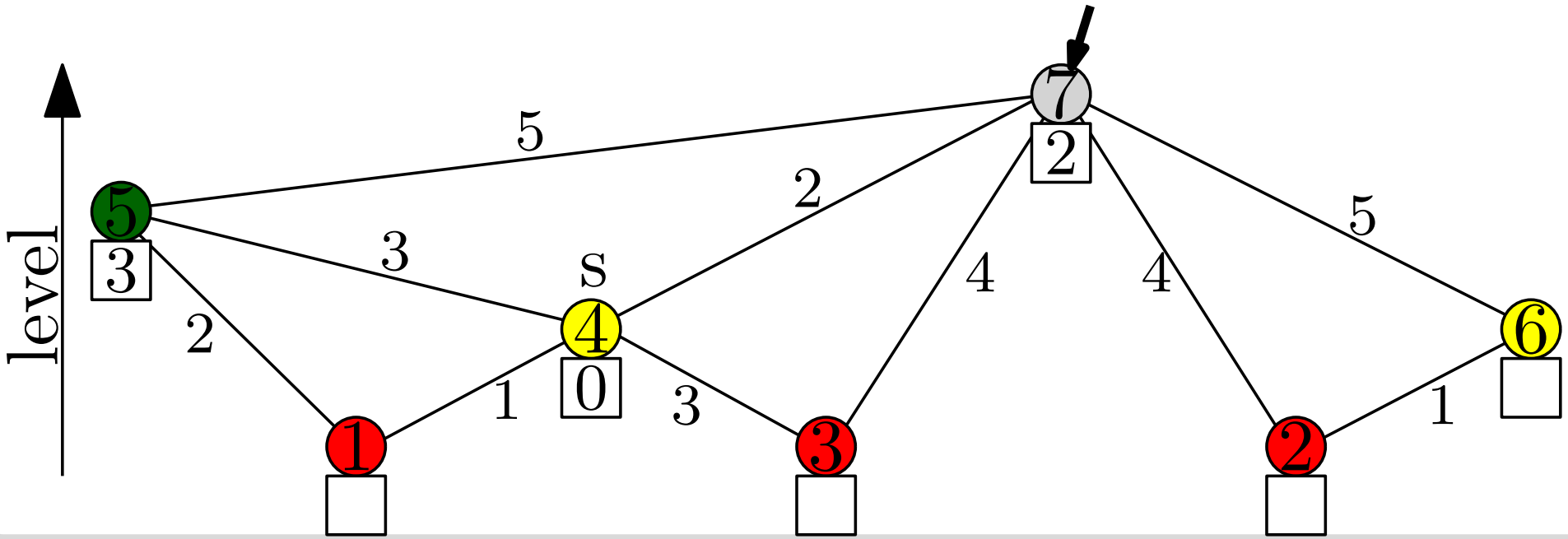
- vorwärts CH Suche von *s* ($\approx$ 0.05 ms)
- setze Distanzen *d* für alle erreichten Knoten



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

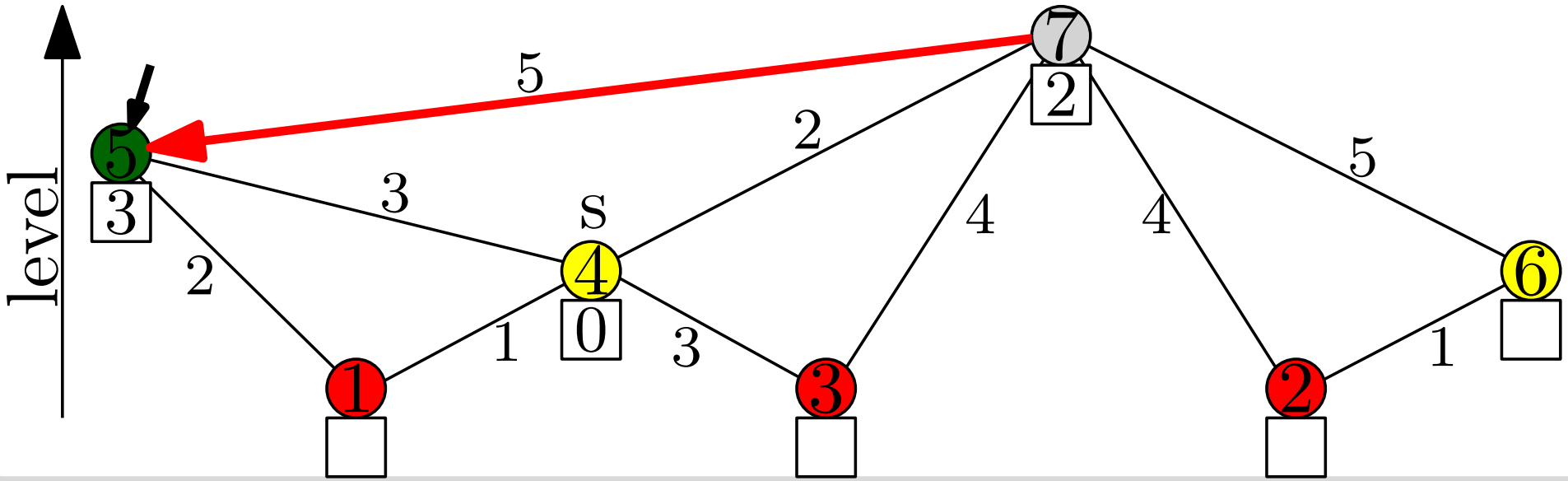
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

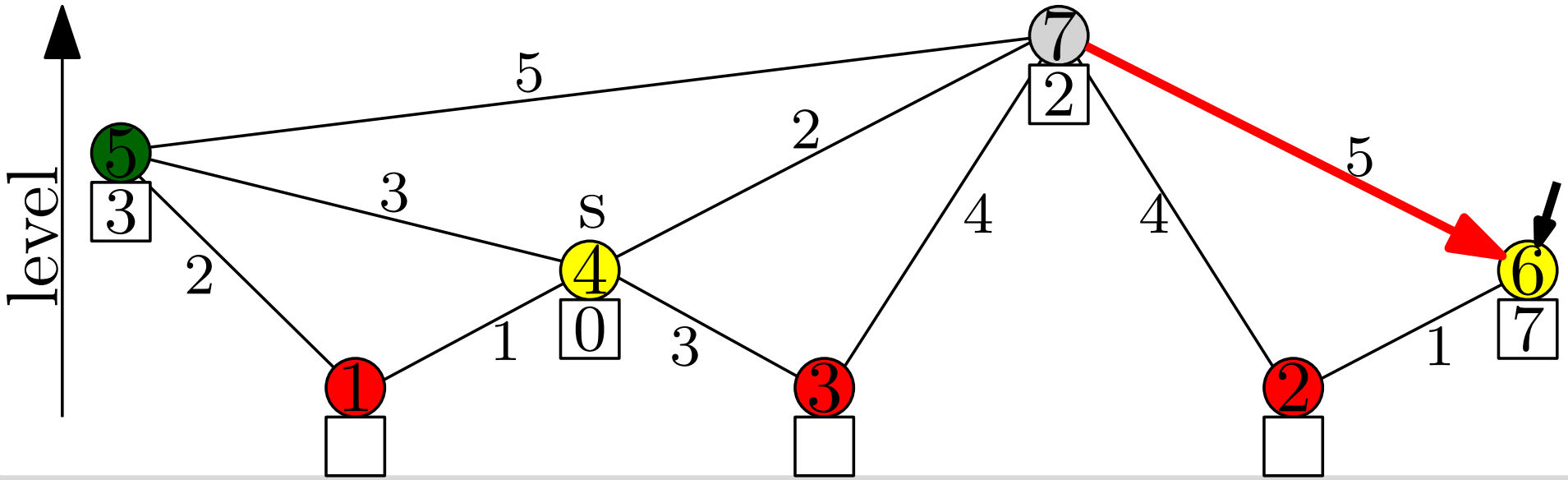
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

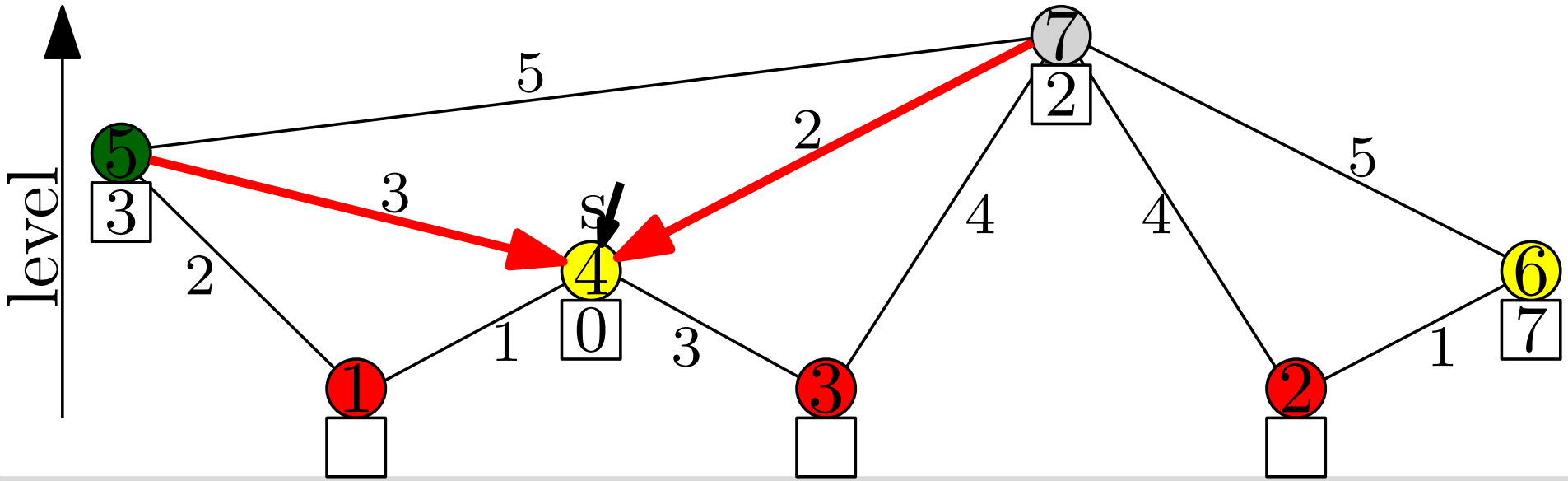
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

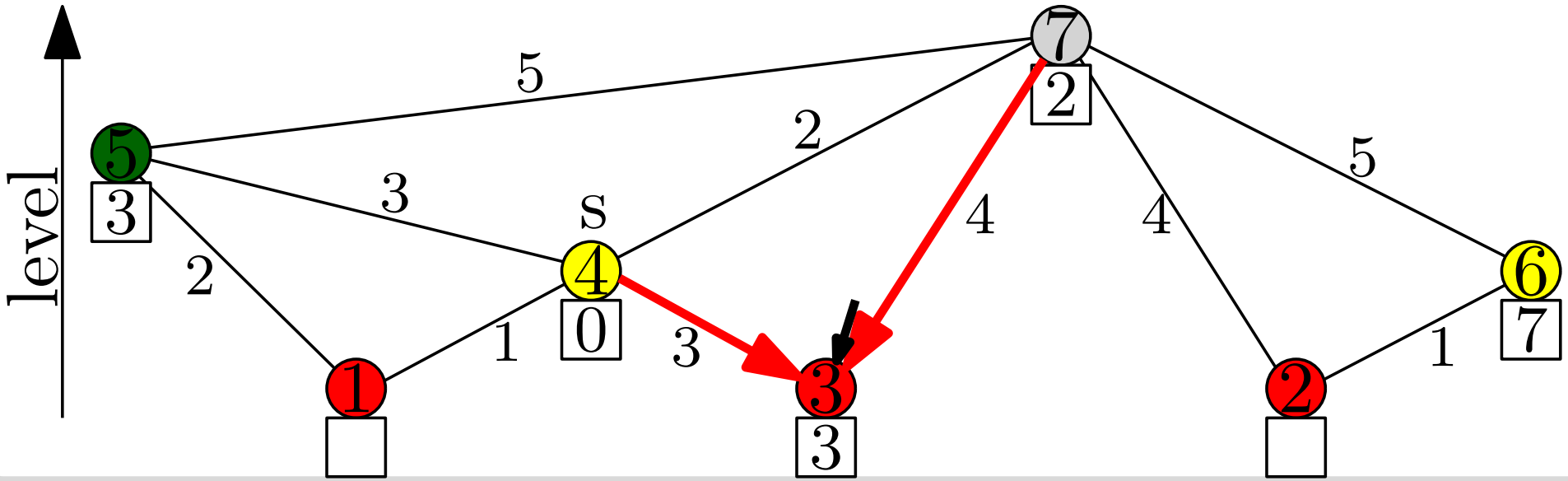
- vorwärts CH Suche von $s$ ($\approx 0.05$ ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

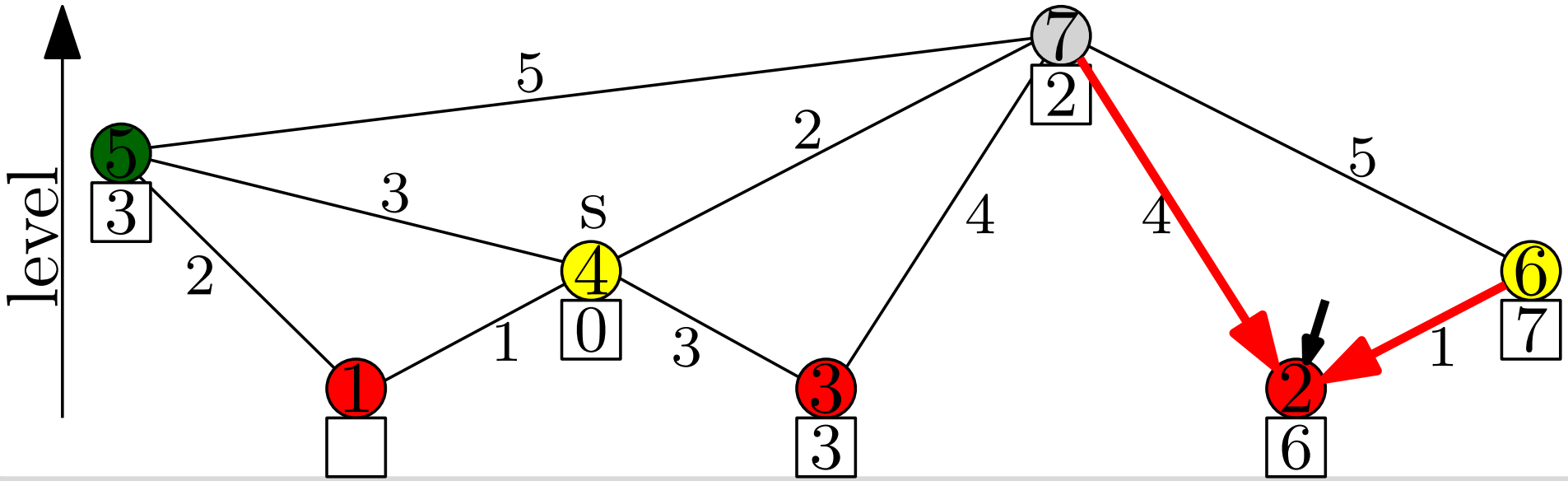
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

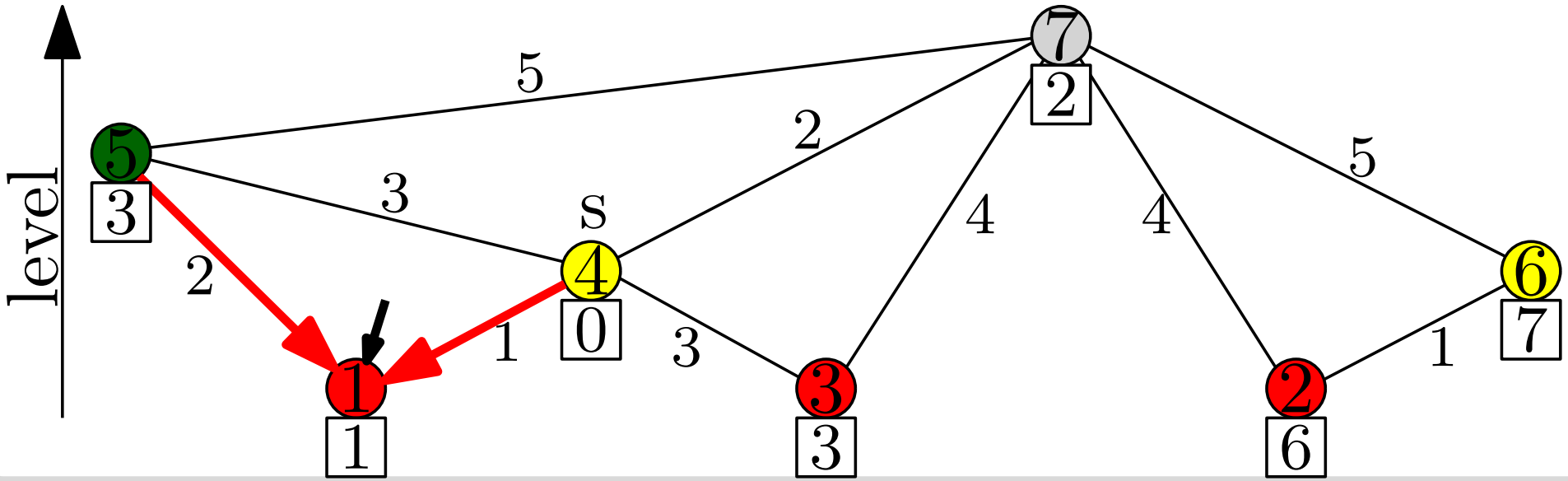
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

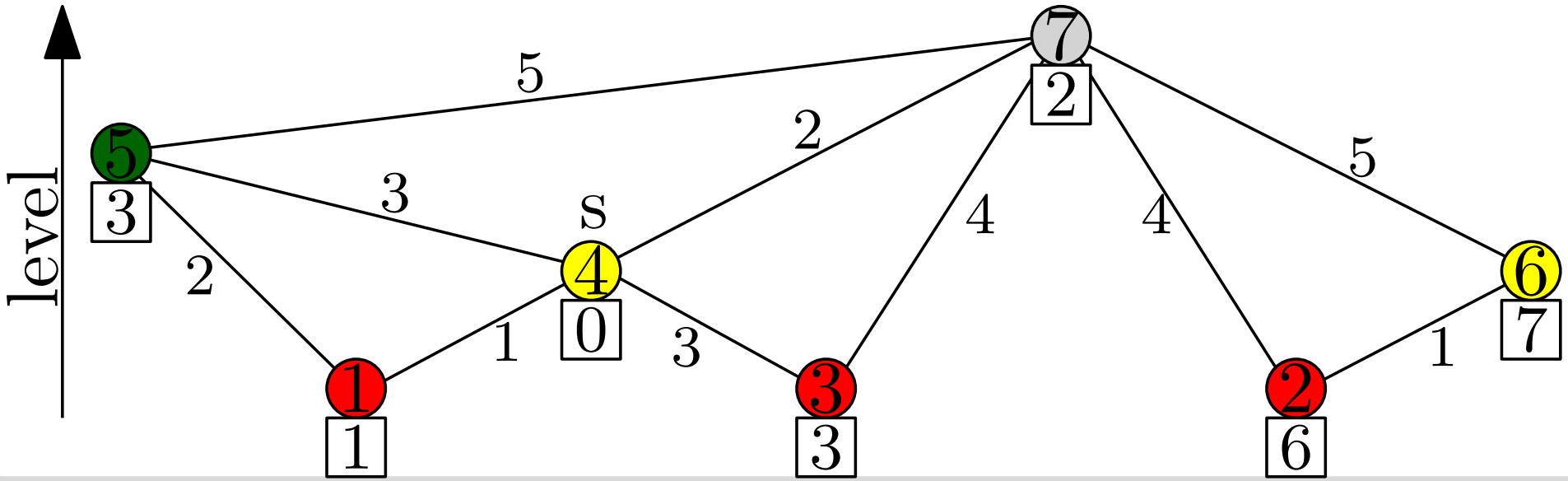
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

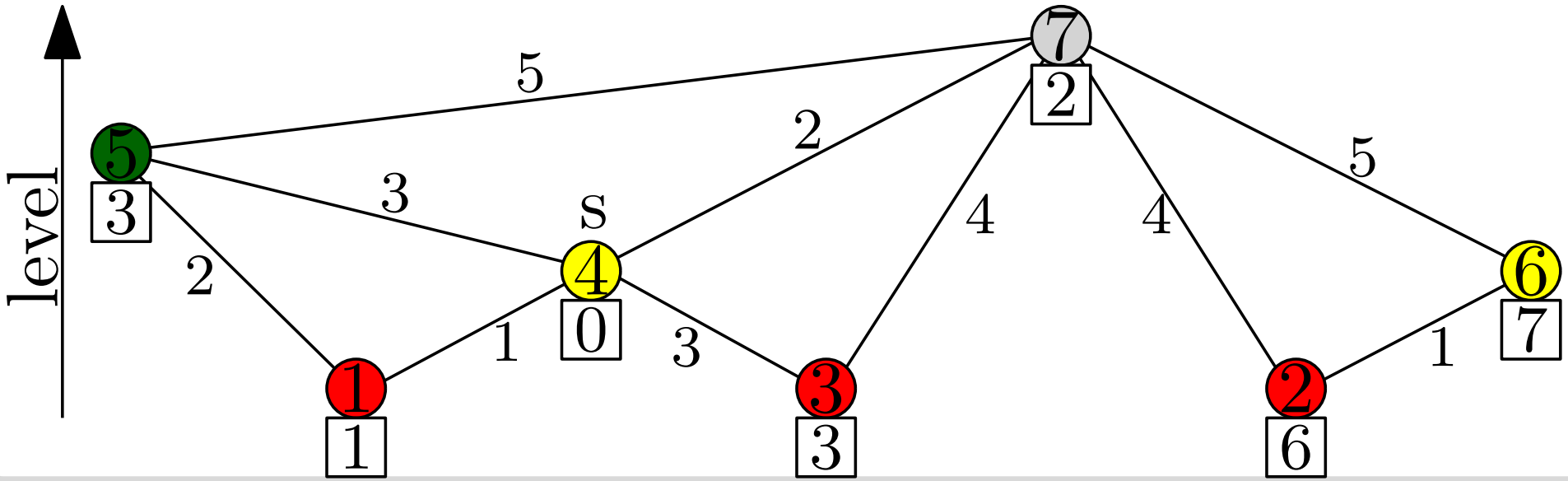
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$
- top-down Bearbeitung ohne Priority Queue (ca. 2.0 s)



# Wiederholung: PHAST

**one-to-all Suche von $s$:**

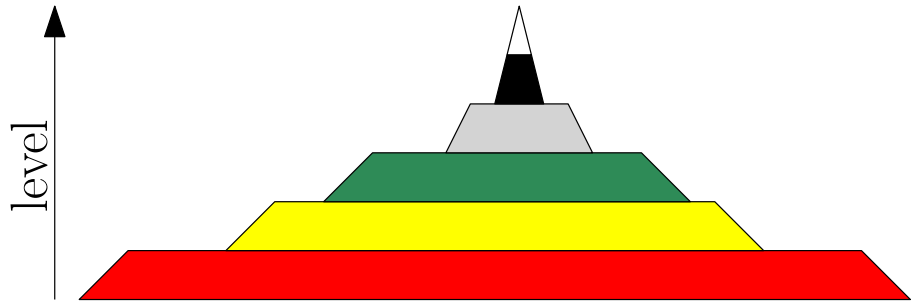
- vorwärts CH Suche von $s$ ($\approx$ 0.05 ms)
- setze Distanzen $d$ für alle erreichten Knoten
- bearbeite alle Knoten $u$ in inverser Levelordnung:
  - checke eingehende Kanten $(v, u)$ mit $lev(v) > lev(u)$
  - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$
- top-down Bearbeitung ohne Priority Queue (ca. 2.0 s)
- genauso schnell wie BFS. Warum das Ganze?



# Wiederholung: PHAST

**Beobachtung:**

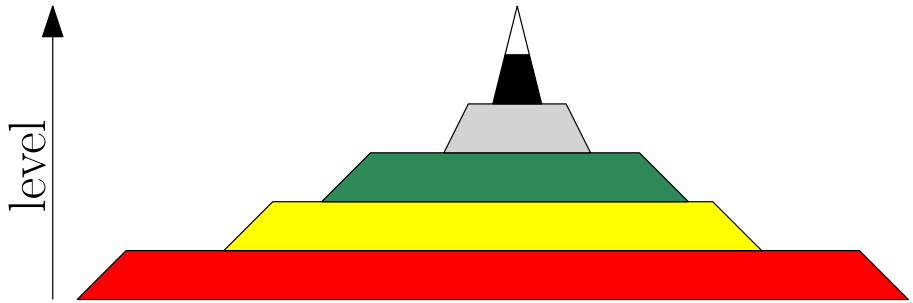
- top-down Prozess ist der Flaschenhals



# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

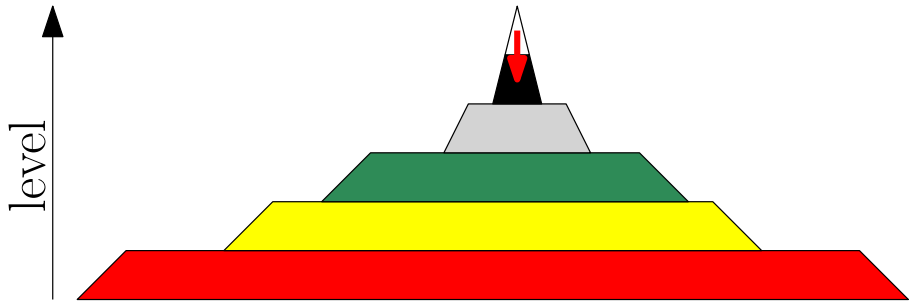


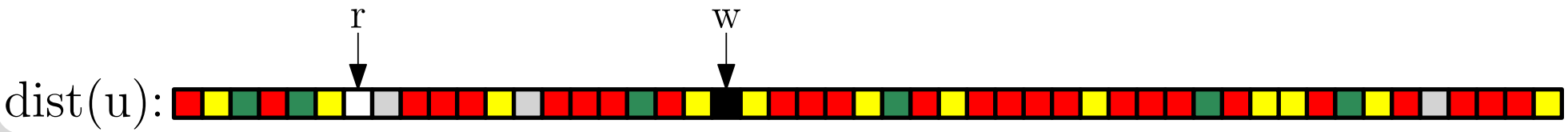

$\mathrm{dist}(u)$:

# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

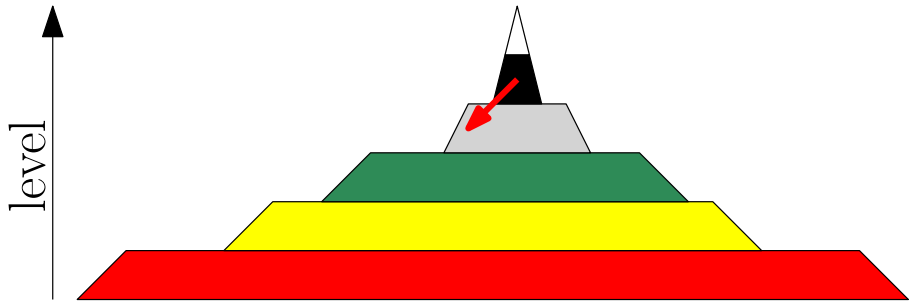


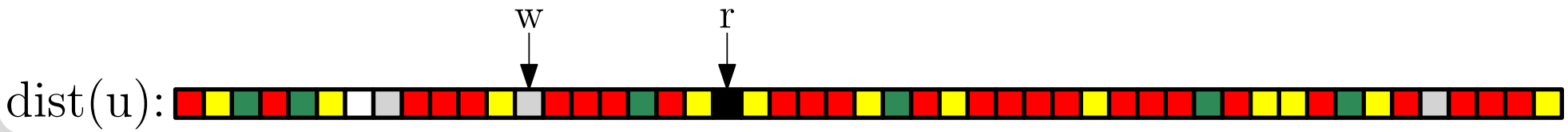



# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

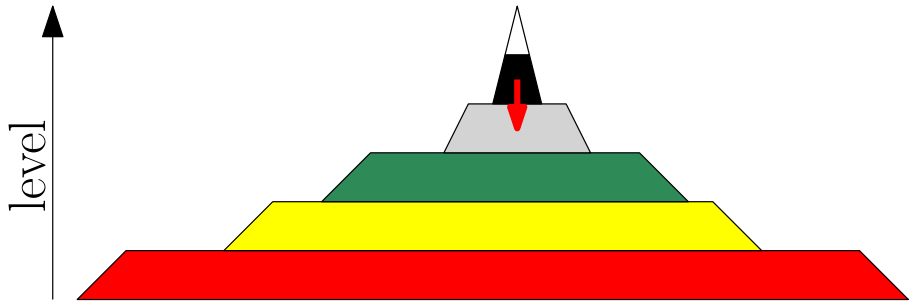


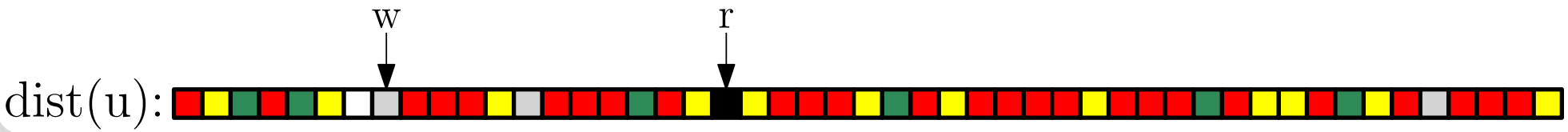



# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

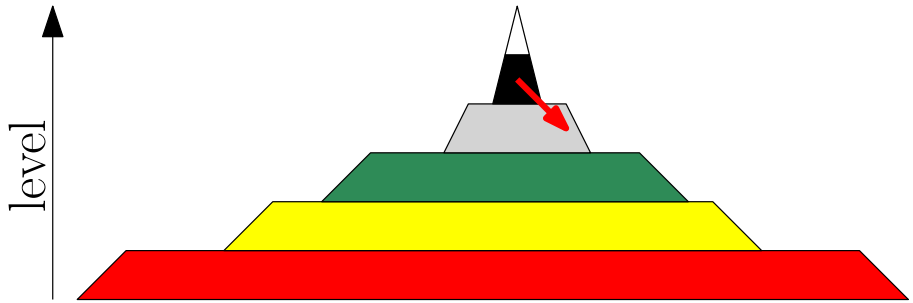


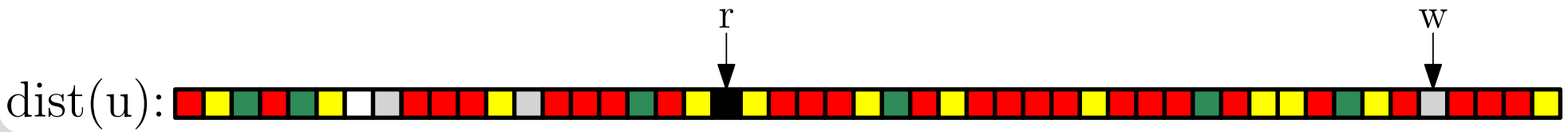



# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

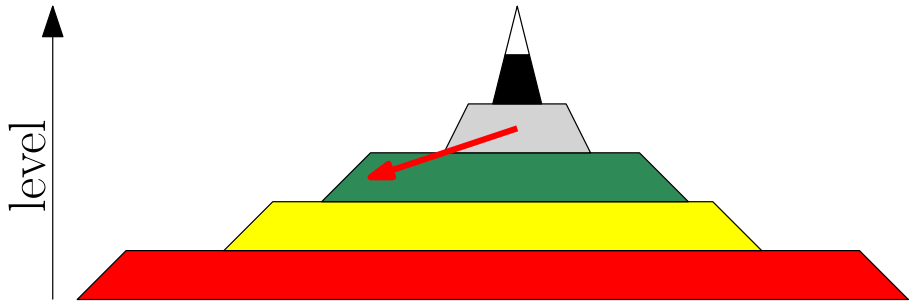


# Wiederholung: PHAST

**Beobachtung:**

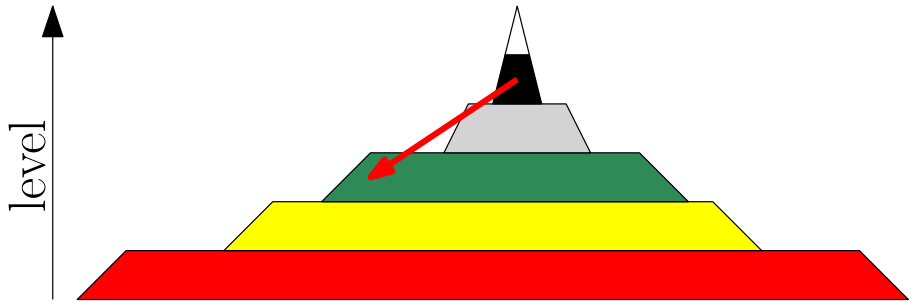
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

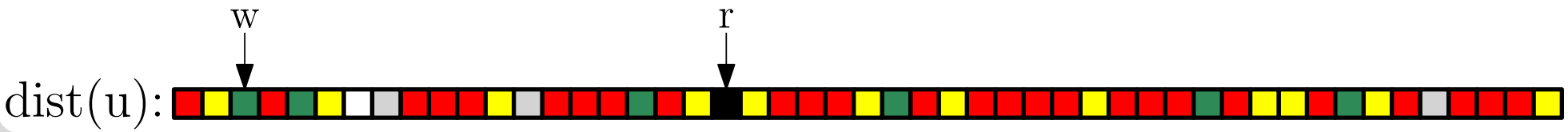



# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient

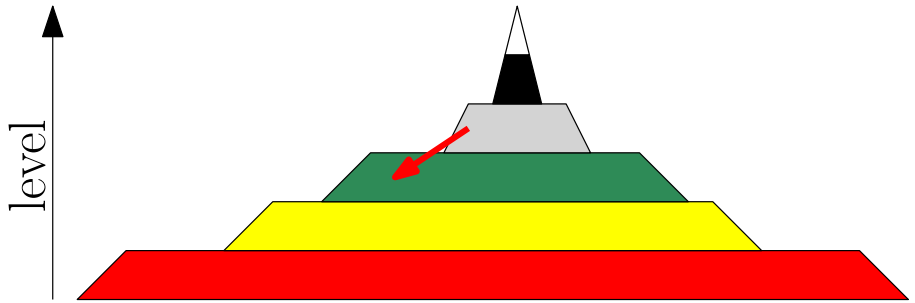


# Wiederholung: PHAST

**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$




# Wiederholung: PHAST

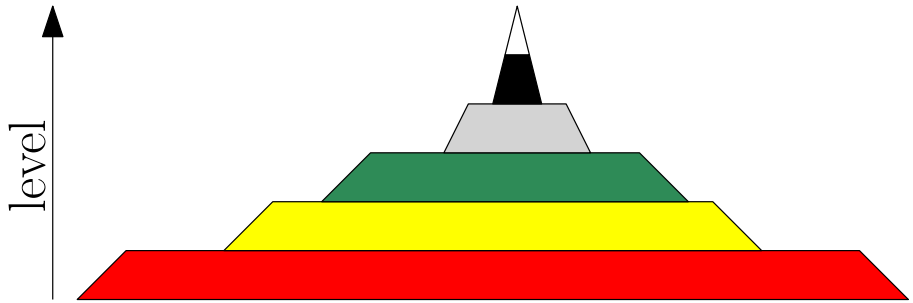
**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep




# Wiederholung: PHAST

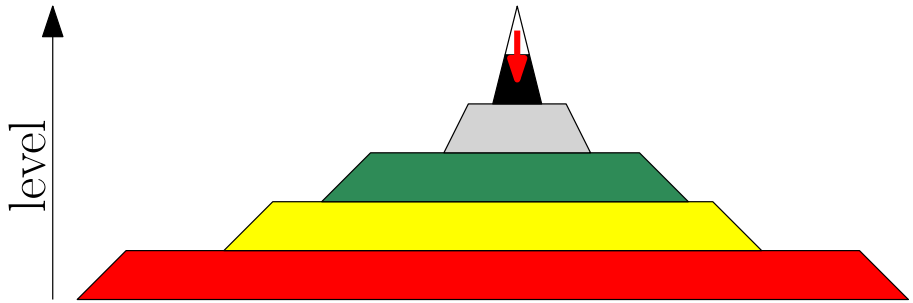
**Beobachtung:**

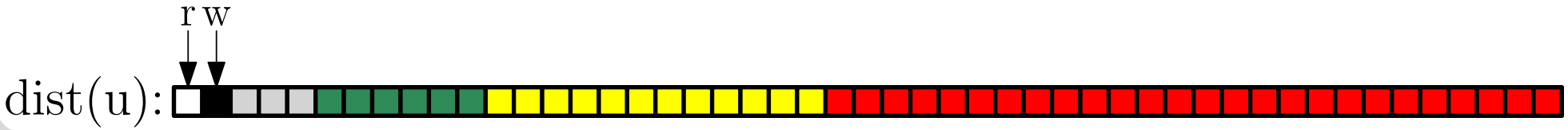
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep



# Wiederholung: PHAST

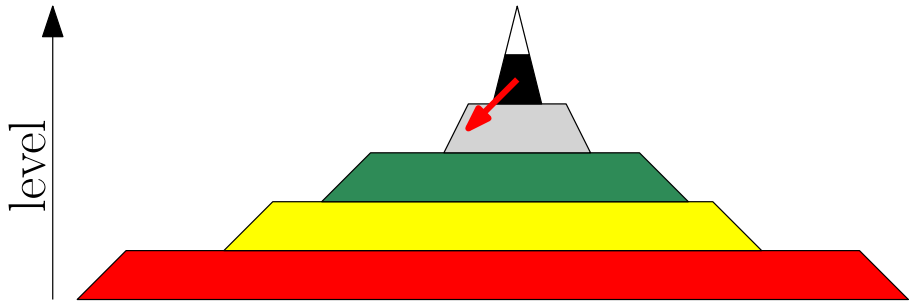
**Beobachtung:**

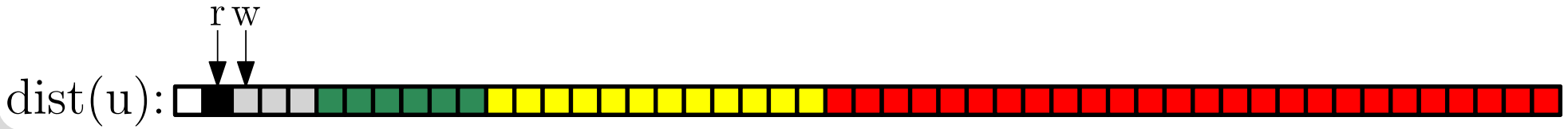
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep



# Wiederholung: PHAST

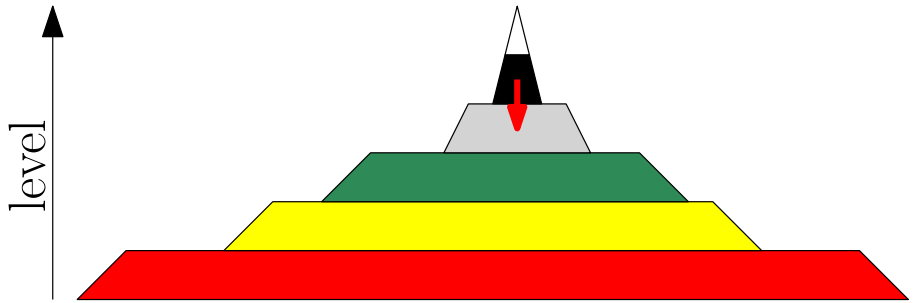
**Beobachtung:**

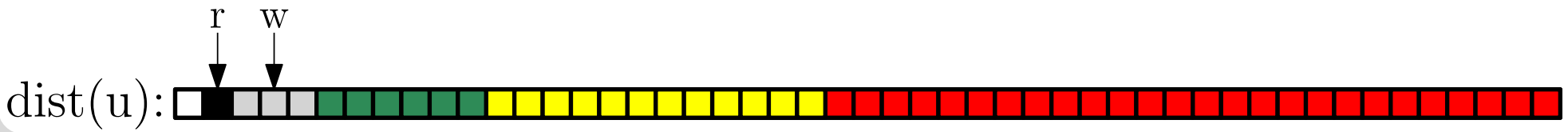
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep



# Wiederholung: PHAST

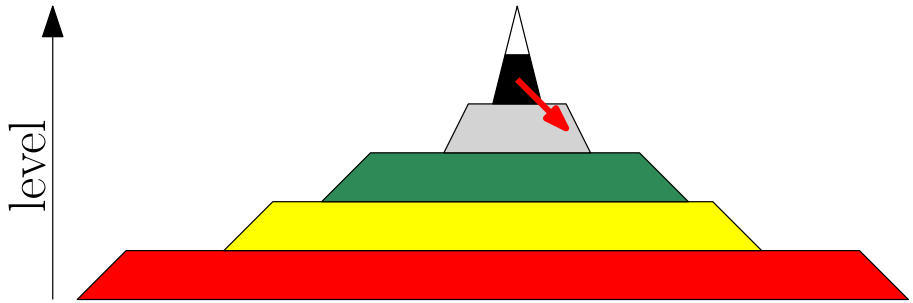
**Beobachtung:**

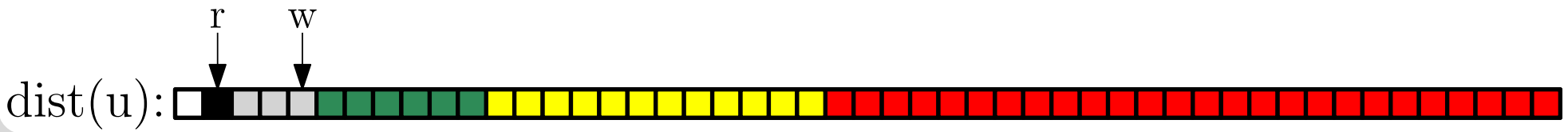
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep



# Wiederholung: PHAST

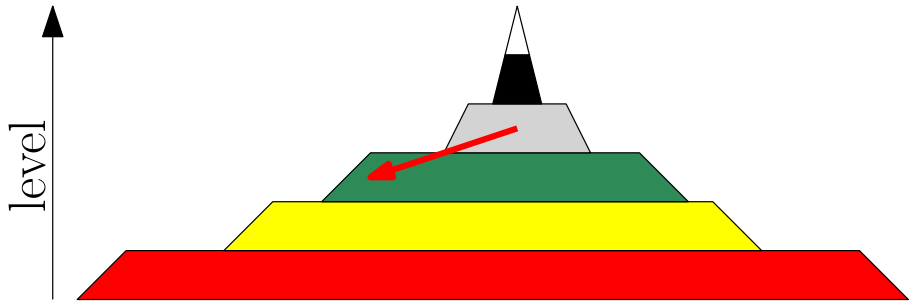
**Beobachtung:**

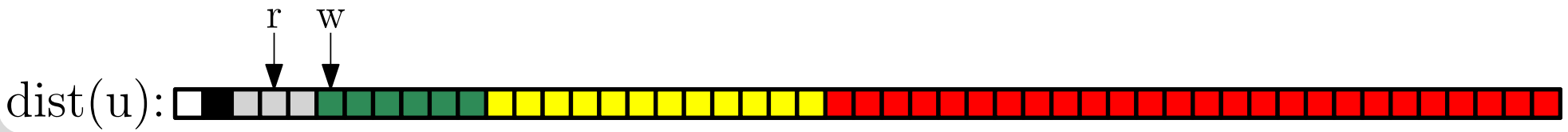
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep



# Wiederholung: PHAST

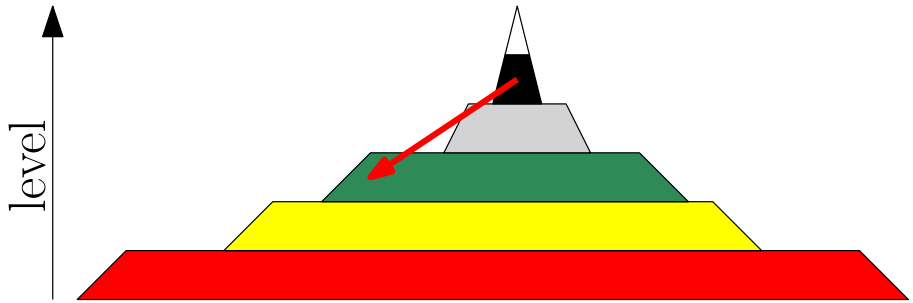
**Beobachtung:**

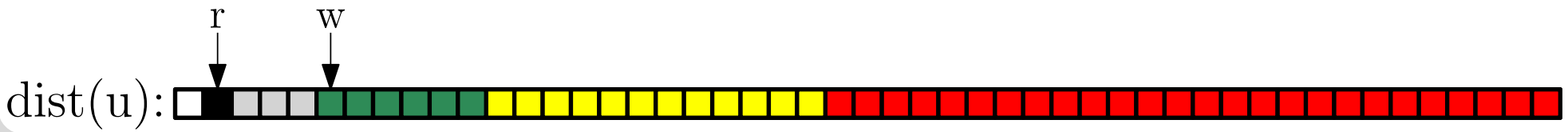
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep



# Wiederholung: PHAST

**Beobachtung:**

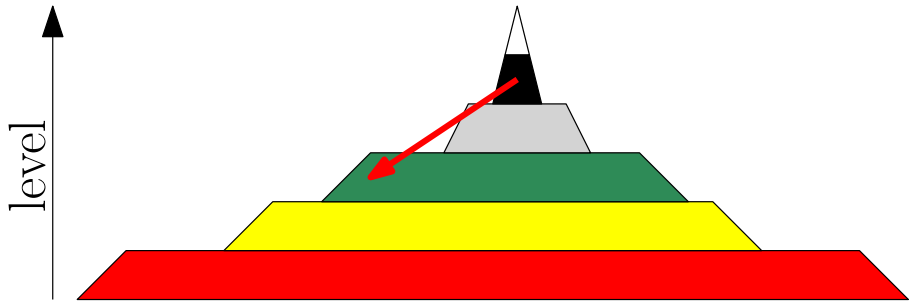
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

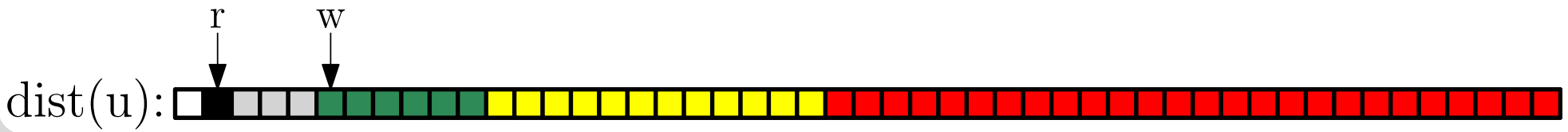
**Idee:**

- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep

$\Rightarrow$ 172 ms pro Baum



# Wiederholung: PHAST

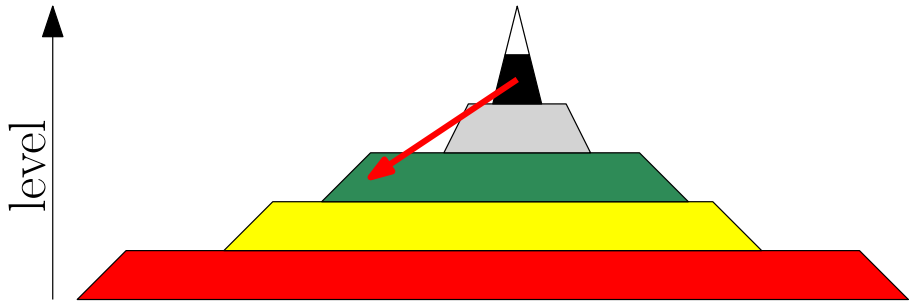
**Beobachtung:**

- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch ineffizient
- Zugriffsmuster sind unabhängig von $s$

**Idee:**

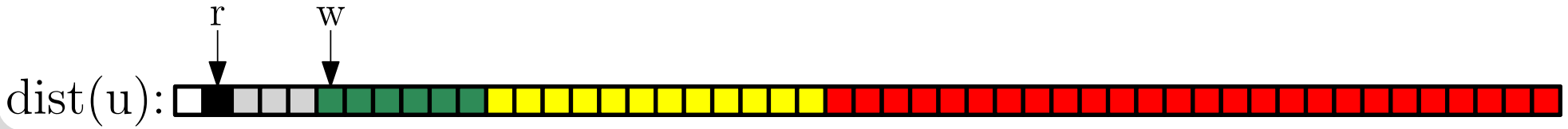
- speicher $G_{\uparrow}$ und $G_{\downarrow}$ separat
- Umordnung der Knoten, Kanten, und Distanzlabel nach Level

$\Rightarrow$ lesen der Kanten und schreiben der Distanzen wird zu einem sequenziellen Sweep




$\Rightarrow$ 172 ms pro Baum

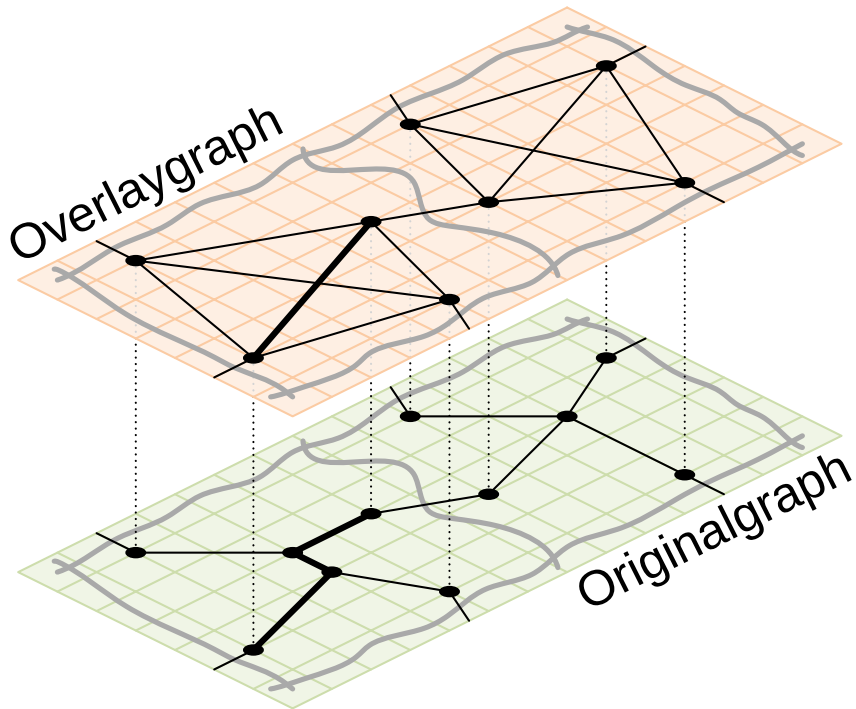
- aber lesen der Distanzen immer noch ineffizient



# Customizable Route Planning (CRP)

**Beschleunigungstechnik für Punkt-zu-Punkt-Anfragen:**

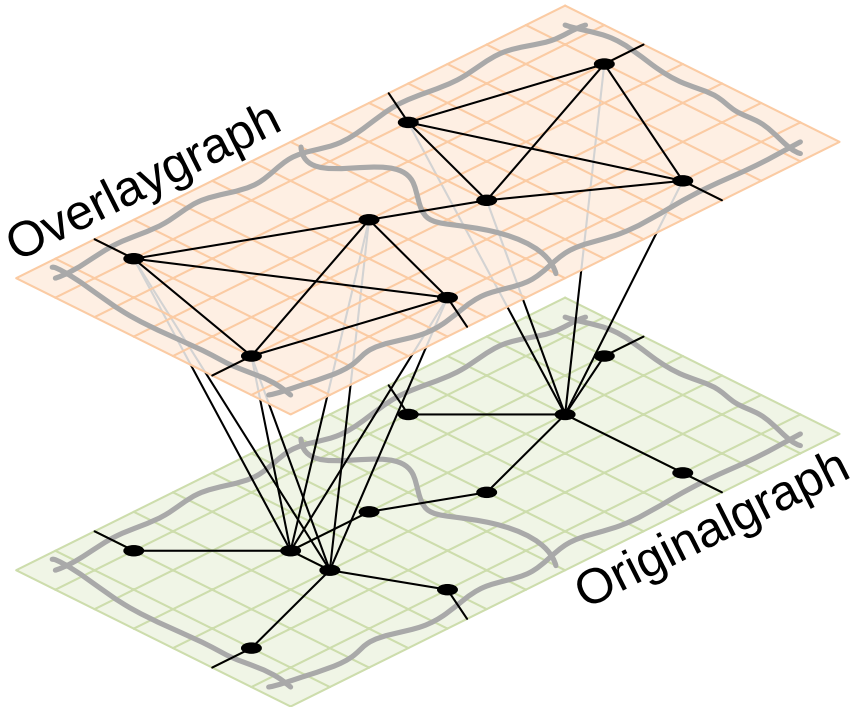
- Vorberechnung: Overlaygraph basierend auf Partition
- Customization: berechne Kosten aller Shortcuts im Overlay
- Anfragen: Dijkstra auf Vereinigung aus Overlay, Startzelle und Zielzelle



# GRASP-Algorithmus

**Erweiterung von CRP zur Berechnung von Kürzeste-Wege-Bäumen:**

- Zusätzliche Abwärtskanten von Randknoten zu inneren Knoten
- One-to-all-Anfragen:
  - Normale CRP-Aufwärtssuche (Aufwärtsphase)
  - Danach relaxiere alle Abwärtskanten (Scanningphase, linearer Sweep)



# Das One-to-Many Problem

**Problem Definition:**

- Eingabe: eine Knoten $s$ und eine Menge $T$
- Ausgabe: Distanz von $s$ zu allen $t \in T$
- Annahme: wir fixieren $T$ und variieren $s$

# Das One-to-Many Problem

**Problem Definition:**

- Eingabe: eine Knoten $s$ und eine Menge $T$
- Ausgabe: Distanz von $s$ zu allen $t \in T$
- Annahme: wir fixieren $T$ und variieren $s$

**offensichtliche Lösungen:**

- Dijkstras Algorithmus (mit Stoppkriterium)
  - $\Rightarrow$ Performance stark abhängig von $|T|$ und Verteilung von $T$

# Das One-to-Many Problem

**Problem Definition:**

- Eingabe: eine Knoten $s$ und eine Menge $T$
- Ausgabe: Distanz von $s$ zu allen $t \in T$
- Annahme: wir fixieren $T$ und variieren $s$

**offensichtliche Lösungen:**

- Dijkstras Algorithmus (mit Stoppkriterium)
  $\Rightarrow$ Performance stark abhängig von $|T|$ und Verteilung von $T$
- $|T|$ p2p Anfragen (z.B. HL)
  $\Rightarrow$ Performance stark abhängig von $|T|$

# Das One-to-Many Problem

**Problem Definition:**

- Eingabe: eine Knoten $s$ und eine Menge $T$
- Ausgabe: Distanz von $s$ zu allen $t \in T$
- Annahme: wir fixieren $T$ und variieren $s$

**offensichtliche Lösungen:**

- Dijkstras Algorithmus (mit Stoppkriterium)
  $\Rightarrow$ Performance stark abhängig von $|T|$ und Verteilung von $T$
- $|T|$ p2p Anfragen (z.B. HL)
  $\Rightarrow$ Performance stark abhängig von $|T|$
- benutze PHAST (kein Stoppkriterium!)
  $\Rightarrow$ Overkill (vor allem für kleine $T$)

# Erste Ideen

**Vorschläge?**



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

# Ungerichtete Bidir. Techniken

**Definition:**

- $\overrightarrow{\sigma}(s,t)$: Suchraum der Vorwärtssuche von $s$ nach $t$
- $\overleftarrow{\sigma}(s,t)$ analog
- eine bidirektionale Suche ist Ziel-unabhängig, gdw.

$$\forall (s,t_1,t_2) \in V^3 : \overrightarrow{\sigma}(s,t_1) = \overrightarrow{\sigma}(s,t_2) \quad \text{und}$$
$$\forall (s_1,s_2,t) \in V^3 : \overleftarrow{\sigma}(s_1,t) = \overleftarrow{\sigma}(s_2,t)$$

# Ungerichtete Bidir. Techniken

**Definition:**

- $\overrightarrow{\sigma}(s,t)$: Suchraum der Vorwärtssuche von $s$ nach $t$
- $\overleftarrow{\sigma}(s,t)$ analog
- eine bidirektionale Suche ist Ziel-unabhängig, gdw.

$$\forall (s,t_1,t_2) \in V^3 : \overrightarrow{\sigma}(s,t_1) = \overrightarrow{\sigma}(s,t_2) \quad \text{und}$$
$$\forall (s_1,s_2,t) \in V^3 : \overleftarrow{\sigma}(s_1,t) = \overleftarrow{\sigma}(s_2,t)$$

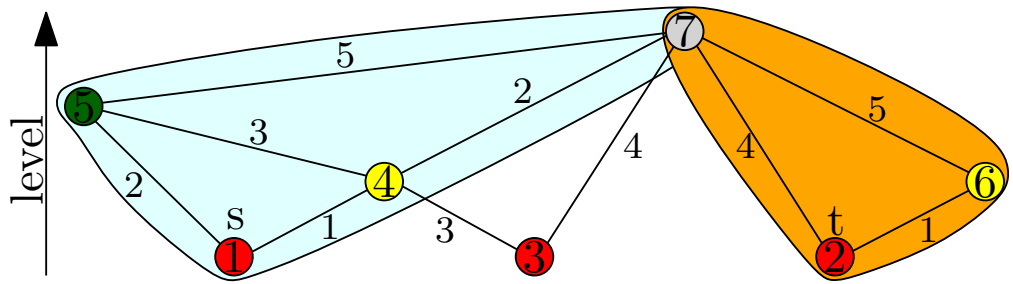
**Beispiele:**

- Bidirektionaler Dijkstra
- ohne Stoppkriterium, lass laufen bis Queues leer sind

# Hierarchische Techniken

**Beobachtung:**

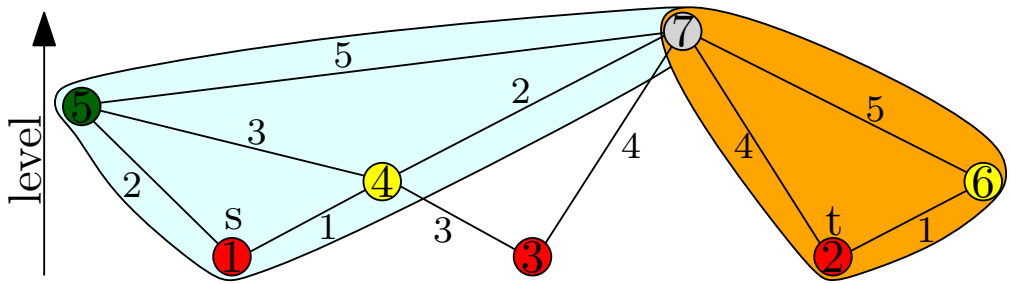
- suchen nur aufwärts
- sind nicht zielgerichtet



# Hierarchische Techniken

**Beobachtung:**

- suchen nur aufwärts
- sind nicht zielgerichtet



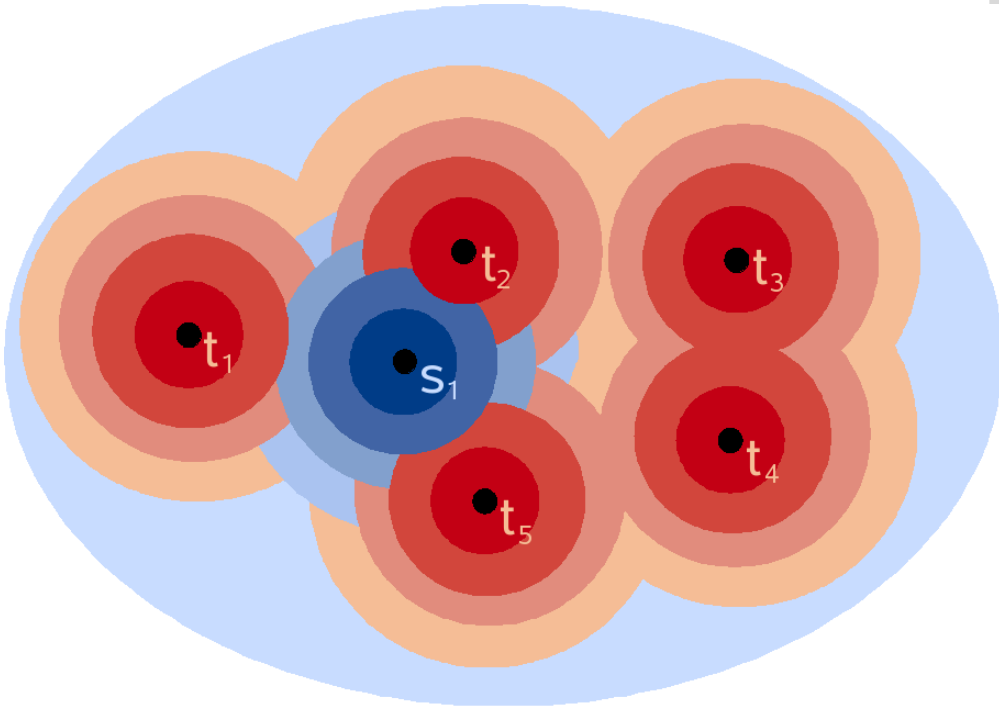

**somit:**

- Bidirektionaler Dijkstra
- Reach
- Contraction Hierarchies
- ohne Stoppkriterium, lass laufen bis Queues leer sind
- HL

# Vorgehen

**Idee:**

- führe $|T|$ Rückwärtssuchen aus
- speicher für jedes besuchte $u$ Abstände zu allen $t \in T$
- verwalte temporäres Distanzarray $D_T$
- führe Vorwärtssuche aus
- aktualisiere Einträge in $D_T$




**Problem:**

- Verwalten der Suchräume?

# Schneiden der Suchräume

**während Rückwärtssuchen: (Target selection phase)**

- je $t \in T$:
- starte Suche im Rückwärts-DAG, breche nicht ab
- für jedes erreichte $u$:
- speichere einen Bucket $\beta(u)$ mit $(t, d(u,t))$
  Alternative Sichtweise: Füge gewichtete Abwärtskanten zu erreichbaren Zielknoten ein

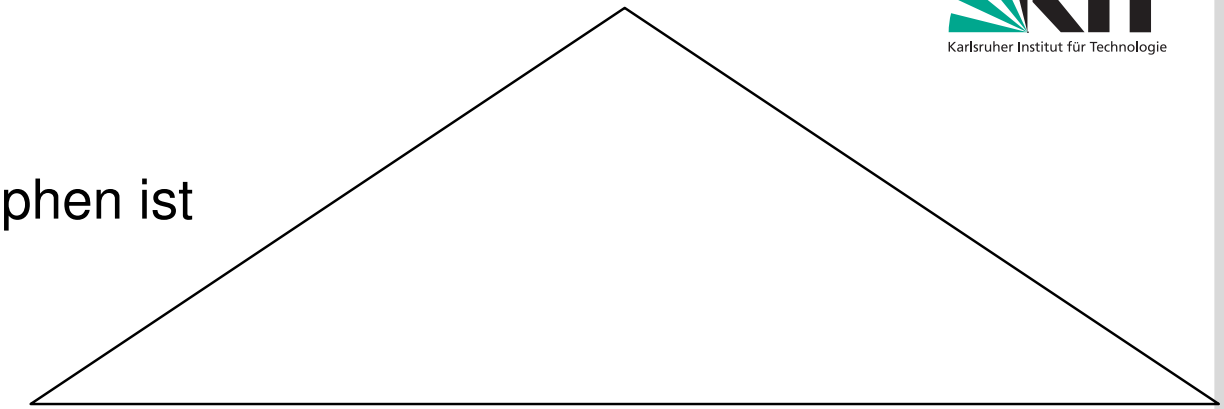
**während Vorwärtssuche: (Query phase)**

- breche nicht ab
- für jedes erreichte $u$:
  - scanne Bucket $\beta(u)$
  - aktualisiere Distanzarray $D_T$

# RPHAST

## Beobachtung PHAST:

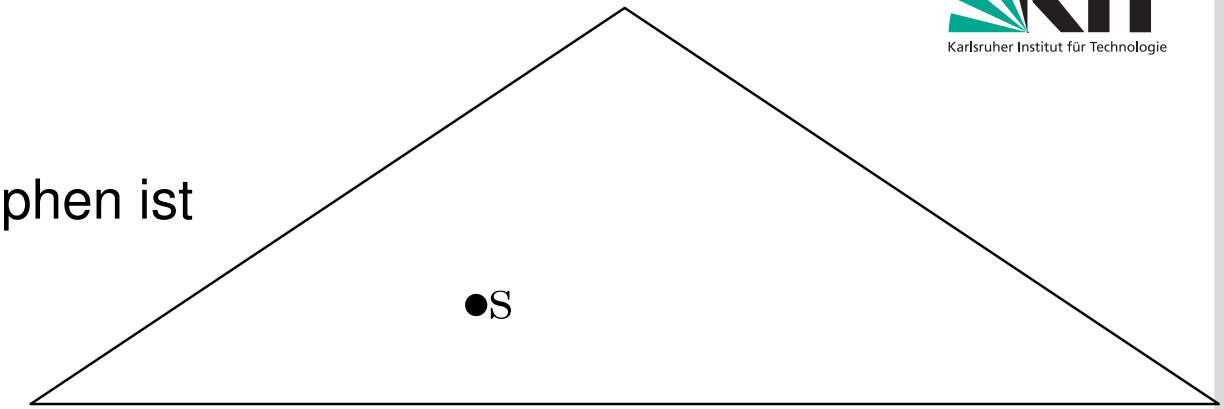
- Sweep über den Graphen ist der Flaschenhals



# RPHAST

## Beobachtung PHAST:

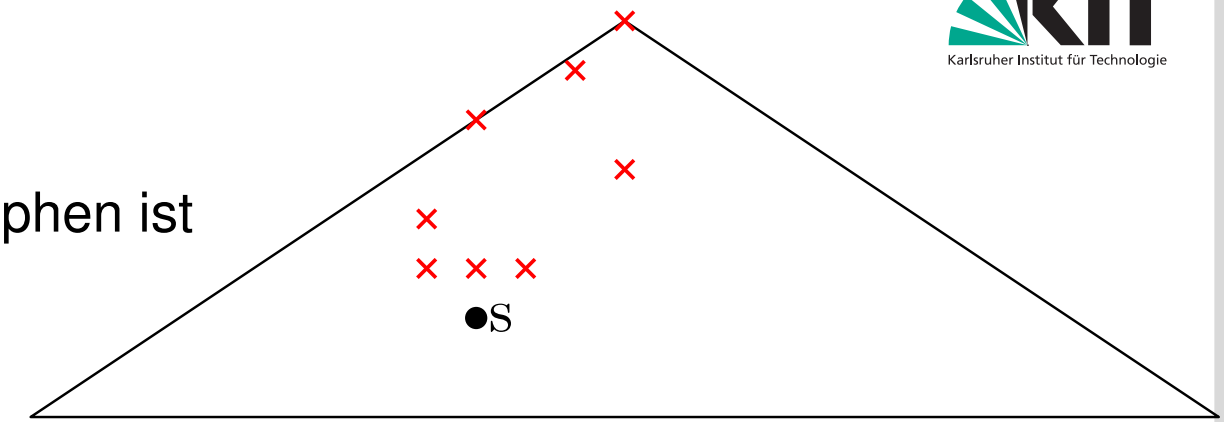
- Sweep über den Graphen ist der Flaschenhals



# RPHAST

## Beobachtung PHAST:

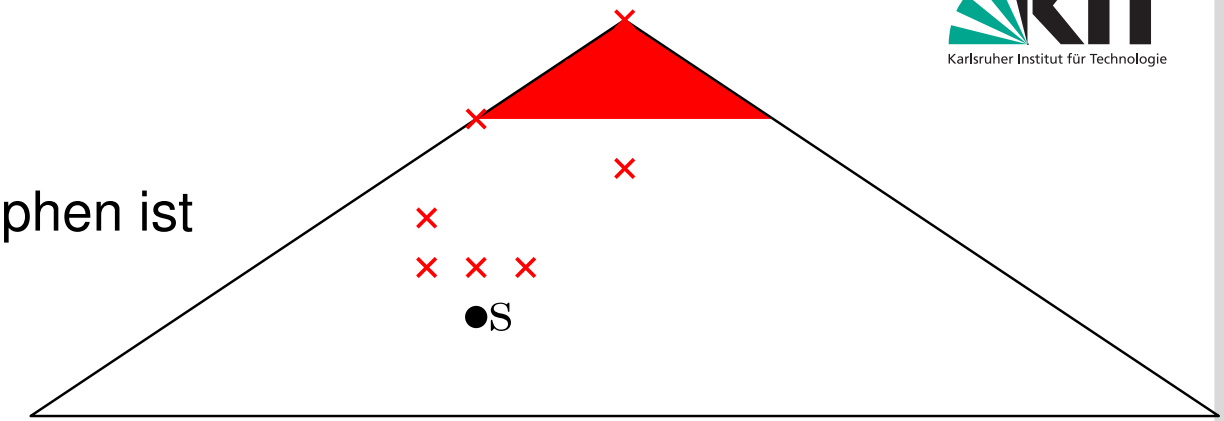
- Sweep über den Graphen ist der Flaschenhals



# RPHAST

## Beobachtung PHAST:

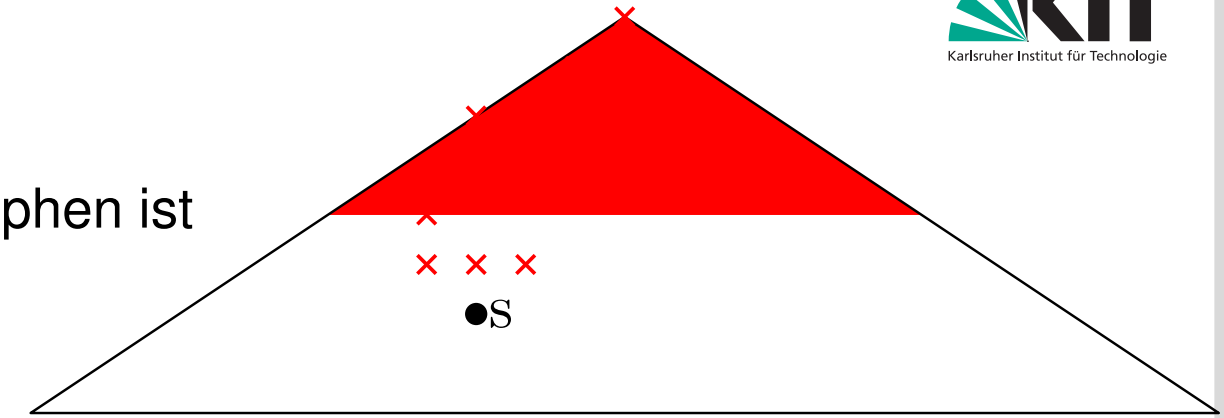
- Sweep über den Graphen ist der Flaschenhals



# RPHAST

## Beobachtung PHAST:

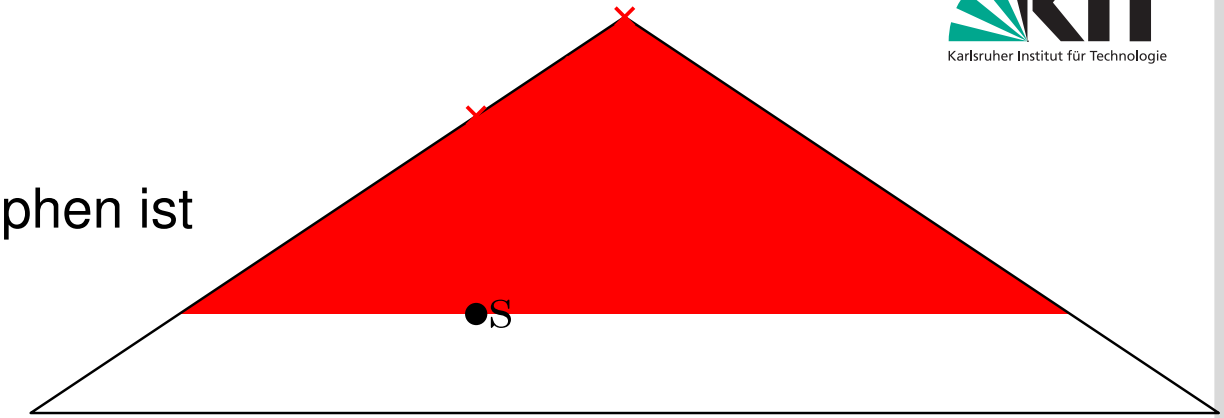
- Sweep über den Graphen ist der Flaschenhals



# RPHAST

## Beobachtung PHAST:

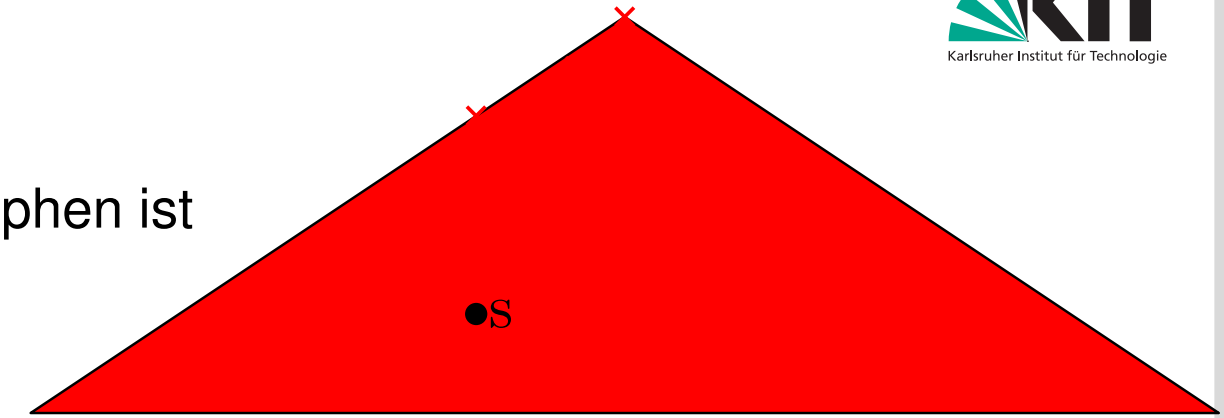
- Sweep über den Graphen ist der Flaschenhals



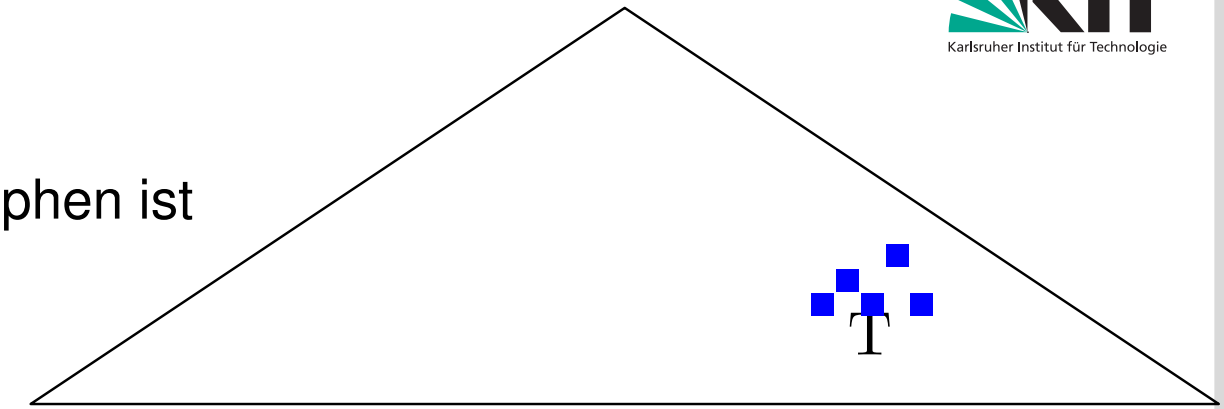
# RPHAST

## Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



# RPHAST



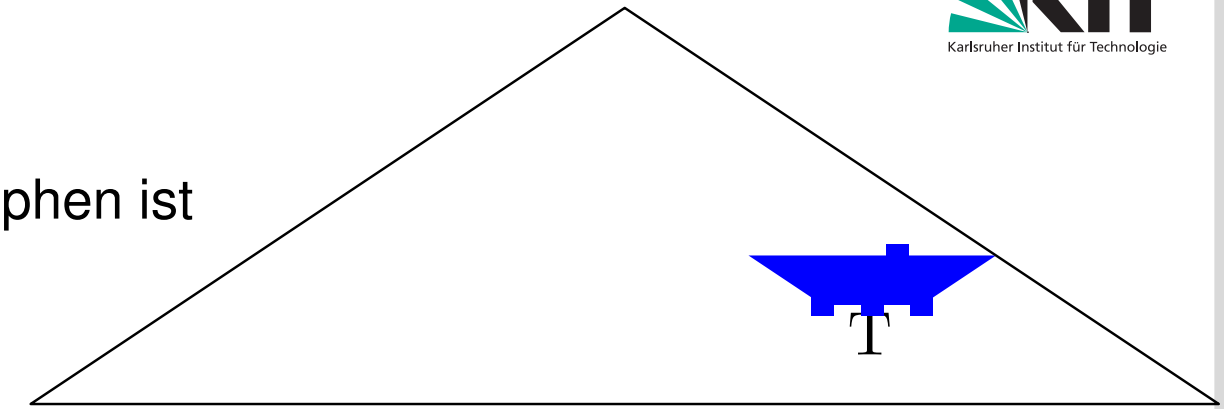

**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals

**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)

# RPHAST




**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals

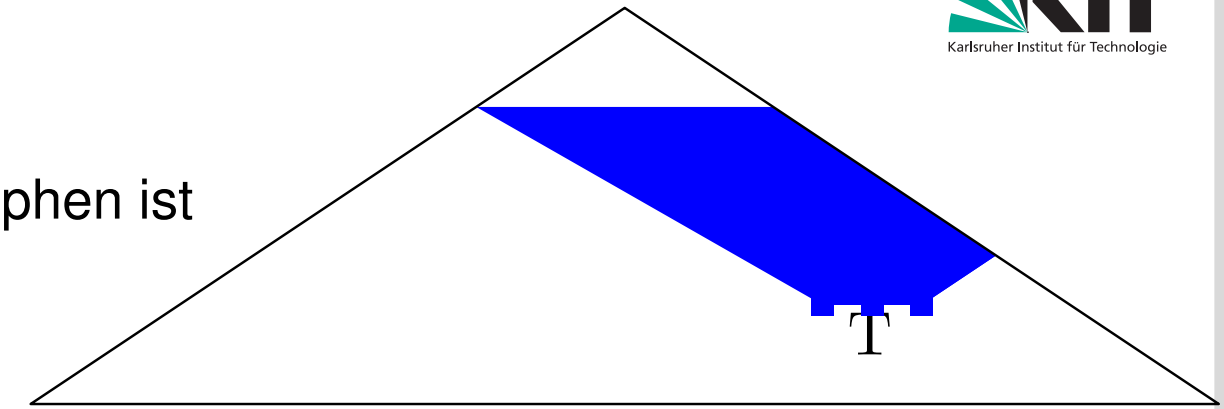
**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)

# RPHAST

**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals



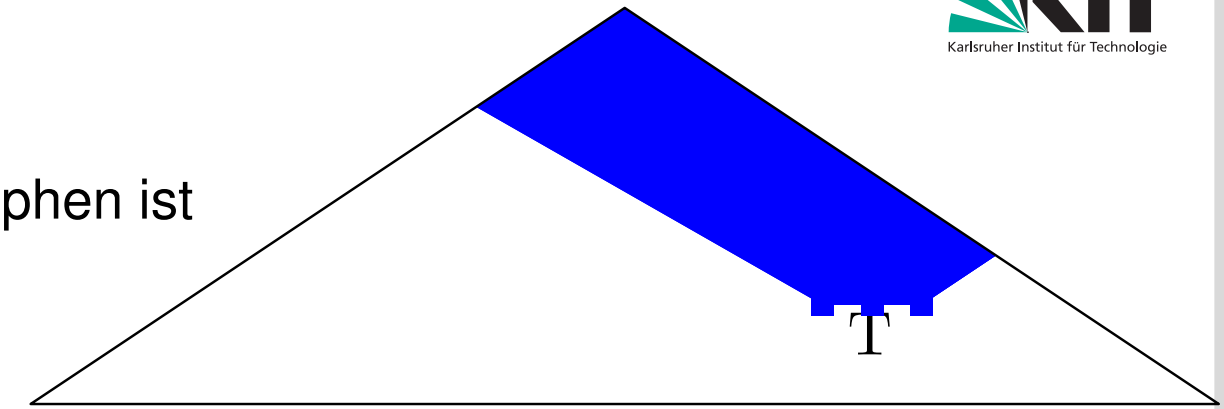

**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)

# RPHAST

**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals



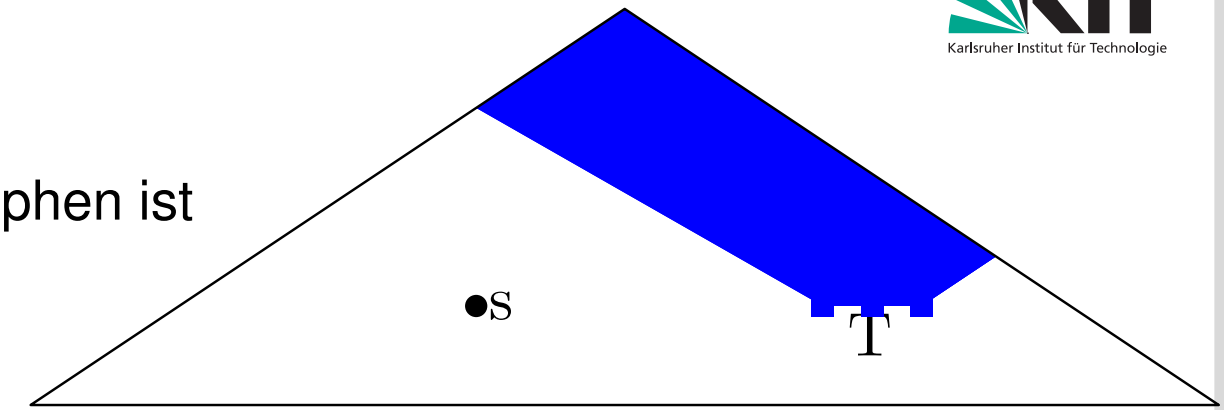

**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)

# RPHAST

**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals



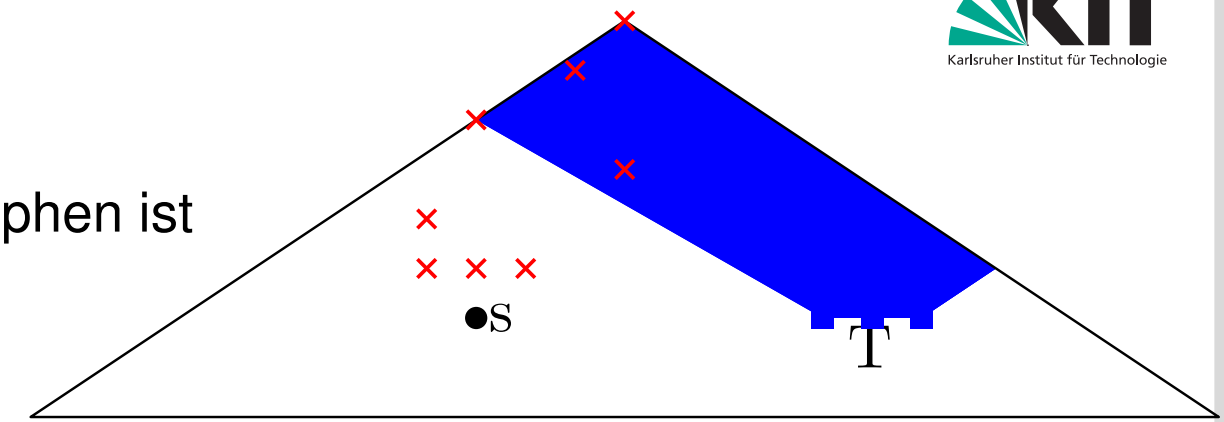

**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)
- Aufwärtssuche im vollen Graphen

# RPHAST

**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals



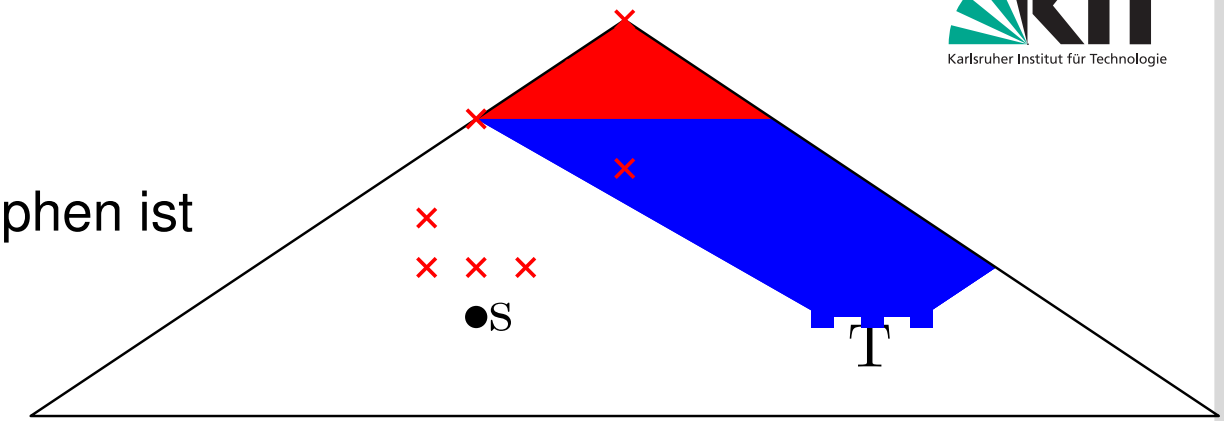

**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)
- Aufwärtssuche im vollen Graphen

# RPHAST

**Beobachtung PHAST:**

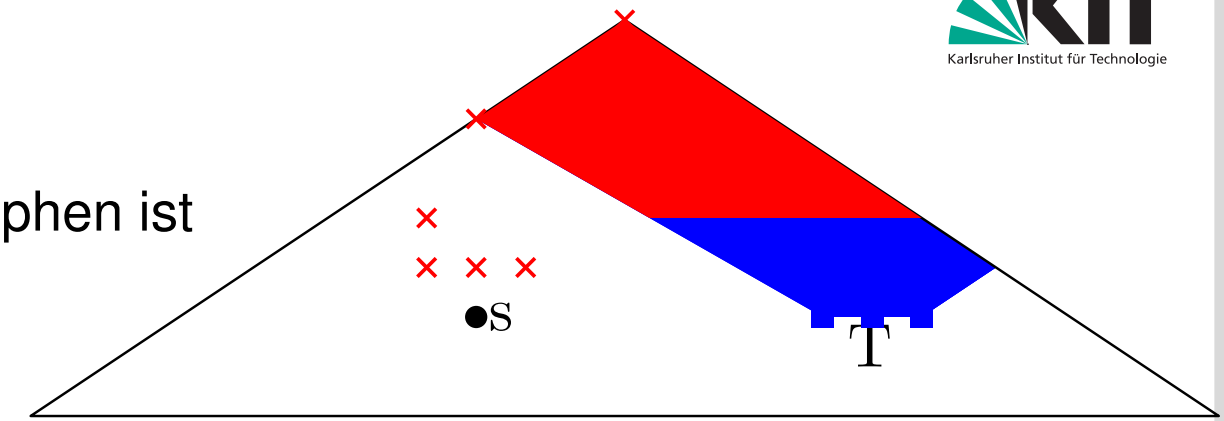
- Sweep über den Graphen ist der Flaschenhals




**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

# RPHAST




**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals

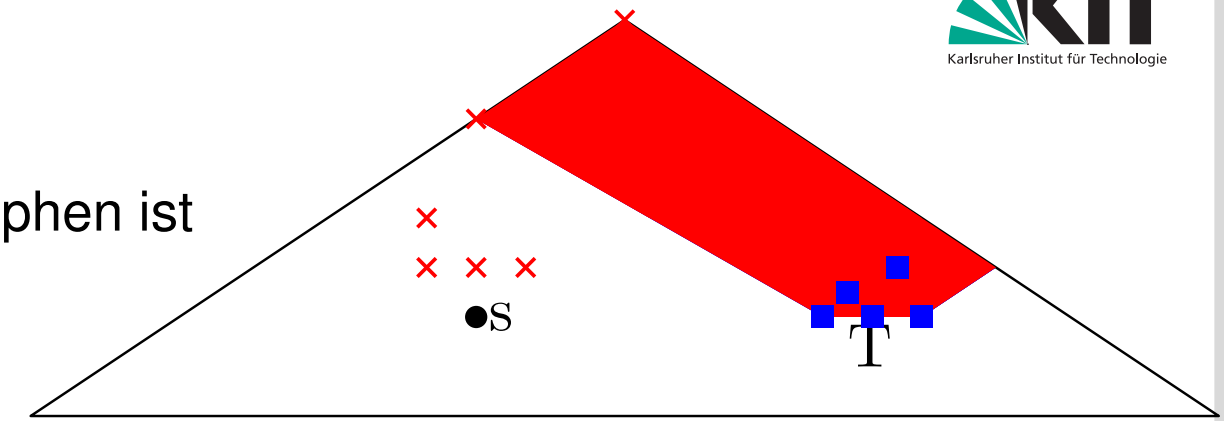
**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

# RPHAST

**Beobachtung PHAST:**

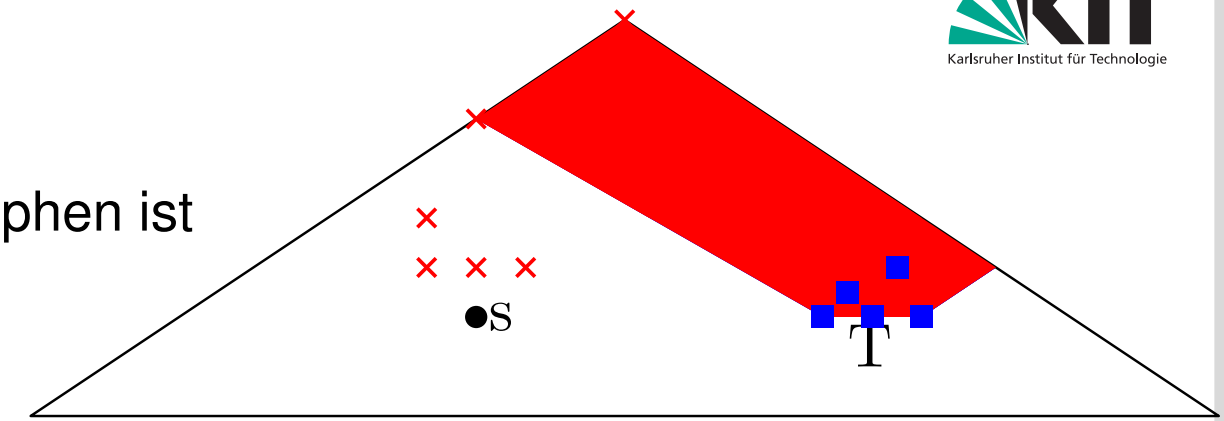
- Sweep über den Graphen ist der Flaschenhals




**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

# RPHAST




**Beobachtung PHAST:**

- Sweep über den Graphen ist der Flaschenhals

**Idee:**

- extrahiere relevanten Teil des Graphen (Ziel Selektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

$\Rightarrow$

- Startknoten kann im ganzem Graphen liegen
- Grösse des extrahierten Graphen hängt von Verteilung und Anzahl $T$ ab
- kann wie PHAST parallelisiert werden
- GPU implementation möglich

# Experimente

**Problem:**
Je nach Szenario liegen die Ziele in einer kleinen Region oder sind über weite Teile des Graphen verteilt.

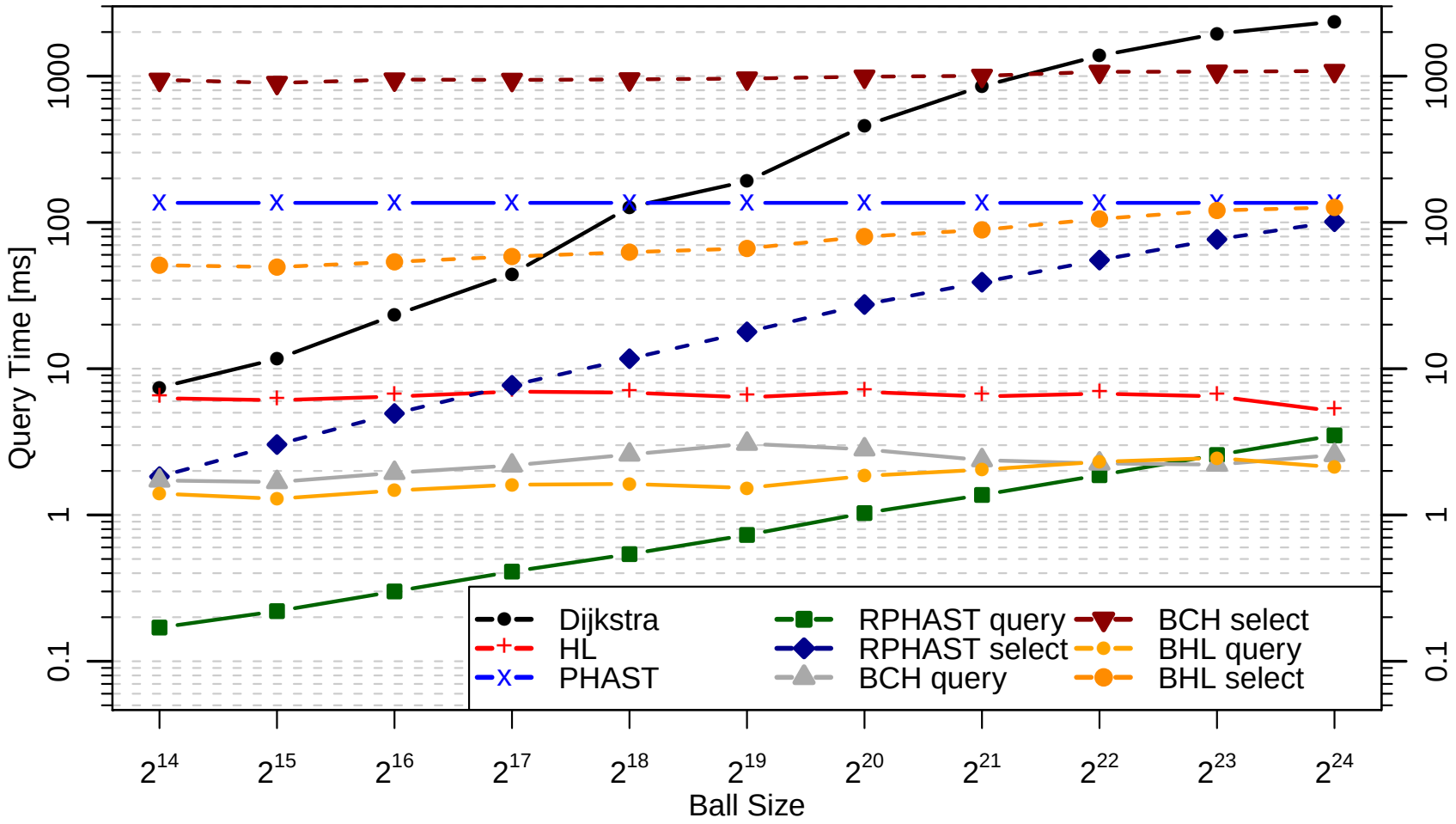
**Setup:**

- starte Dijkstra von zufälligem Knoten $c$
- brich nach $B$ besuchten Knoten ab (Ballsize)
- wähle zufällige Zielknotenmenge $T \subseteq B$

Vergleiche Performance von Bucket CH (BCH), Bucket HL (BHL), RPHAST

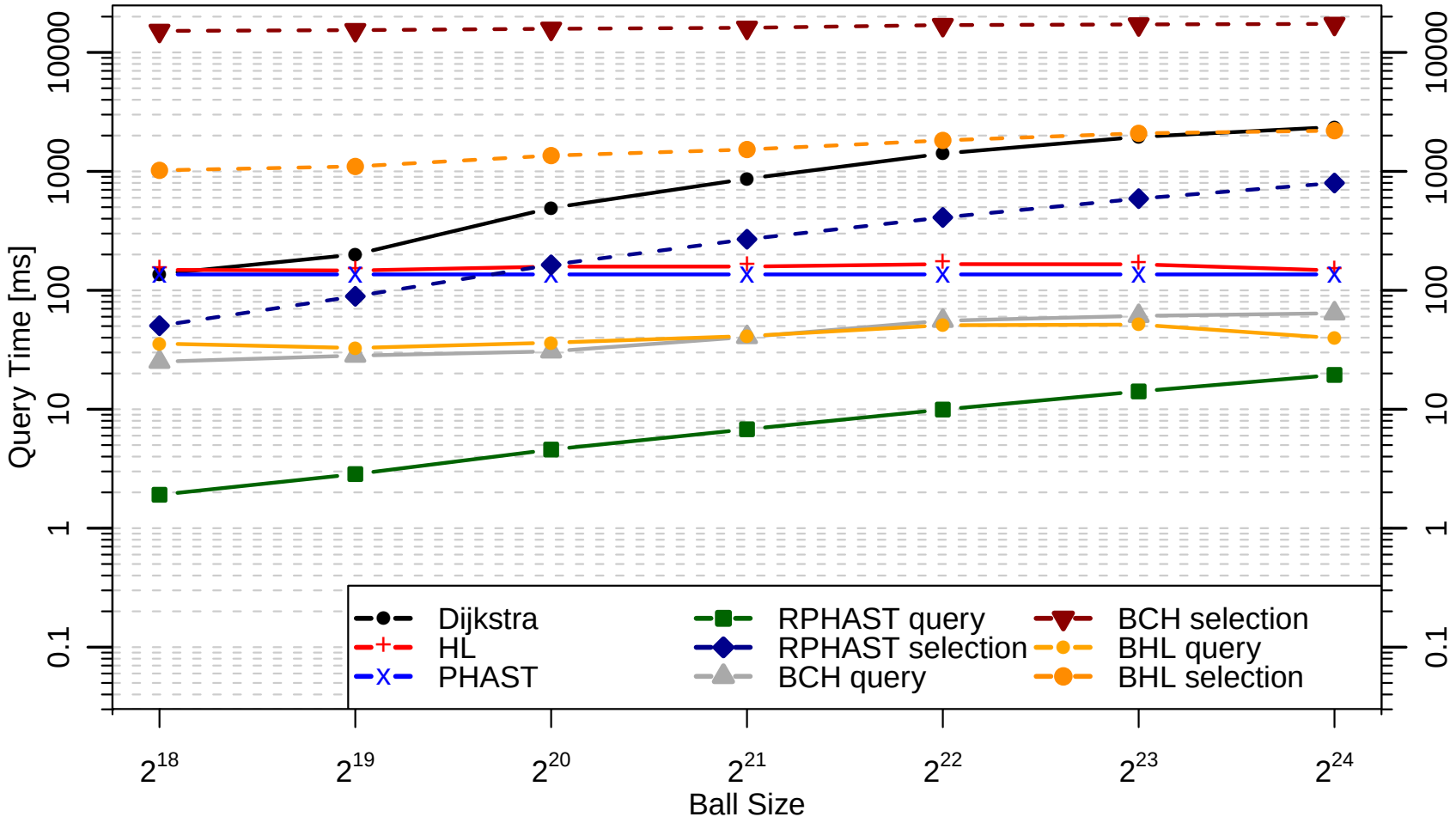
# Experimente I

**input:** Westeuropa (18M Knoten), $|T| = 2^{14}$



# Experimente II

**input:** Westeuropa (18M Knoten), $|T| = 2^{18}$



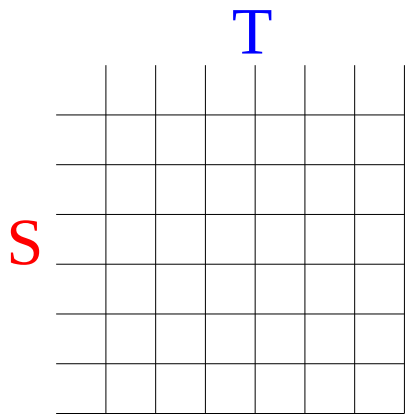
# Many-to-Many Kürzeste Wege

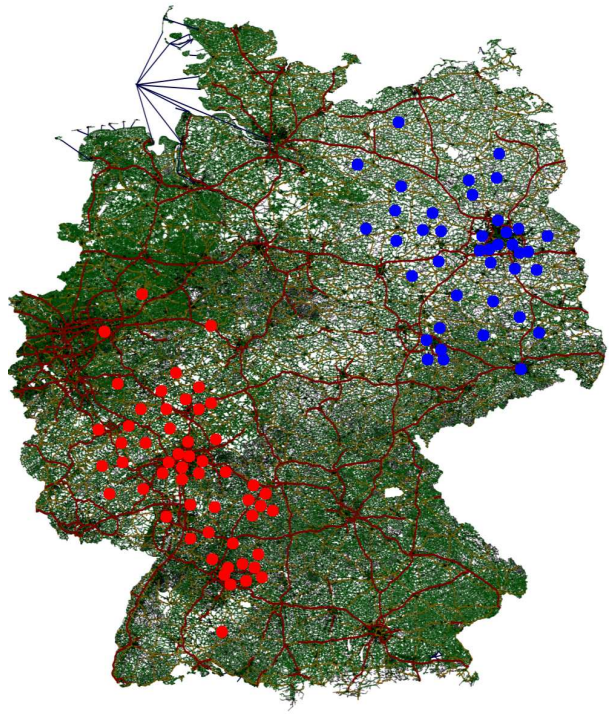
**Gegeben:**

- Graph
- Knotenmengen $S, T \in V$

**Gesucht:**

- Distanzmatrix $D$



# Many-to-Many Kürzeste Wege

**Gegeben:**

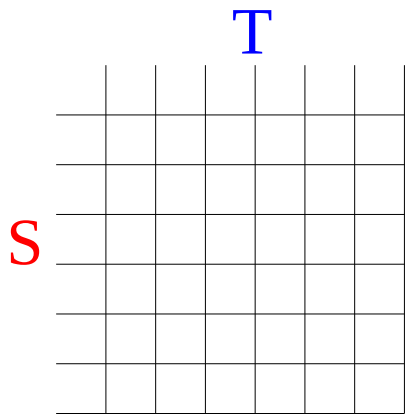
- Graph
- Knotenmengen $S, T \in V$

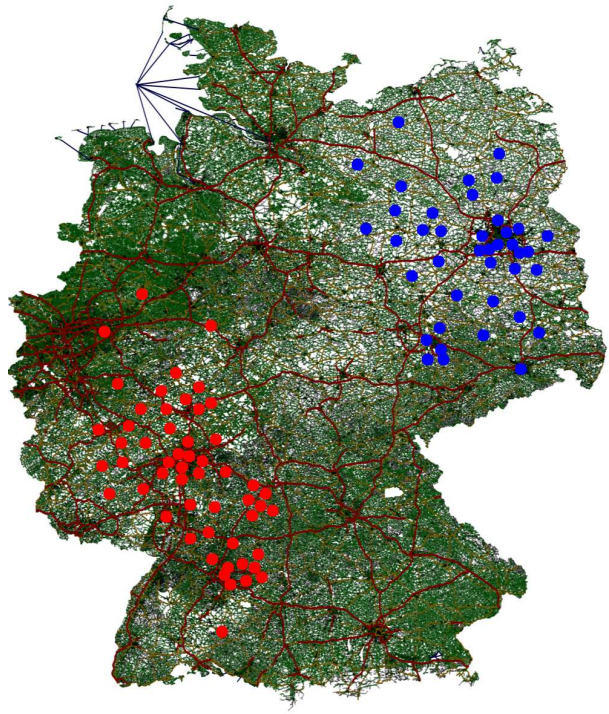
**Gesucht:**

- Distanzmatrix $D$

**Anwendungen:**

- vehicle routing
- traveling salesman



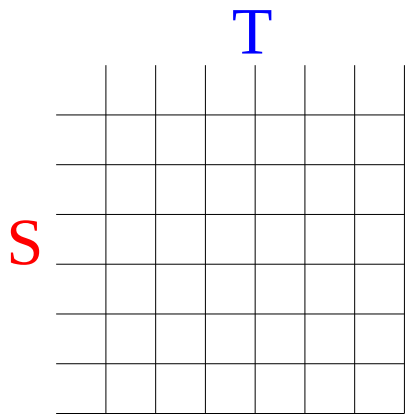
# Many-to-Many Kürzeste Wege

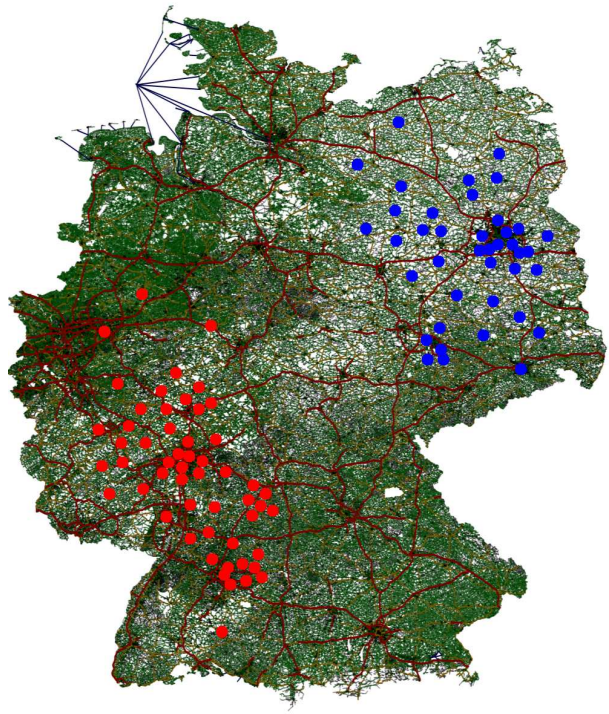
**Gegeben:**

- Graph
- Knotenmengen $S, T \in V$

**Gesucht:**

- Distanzmatrix $D$



**Anwendungen:**

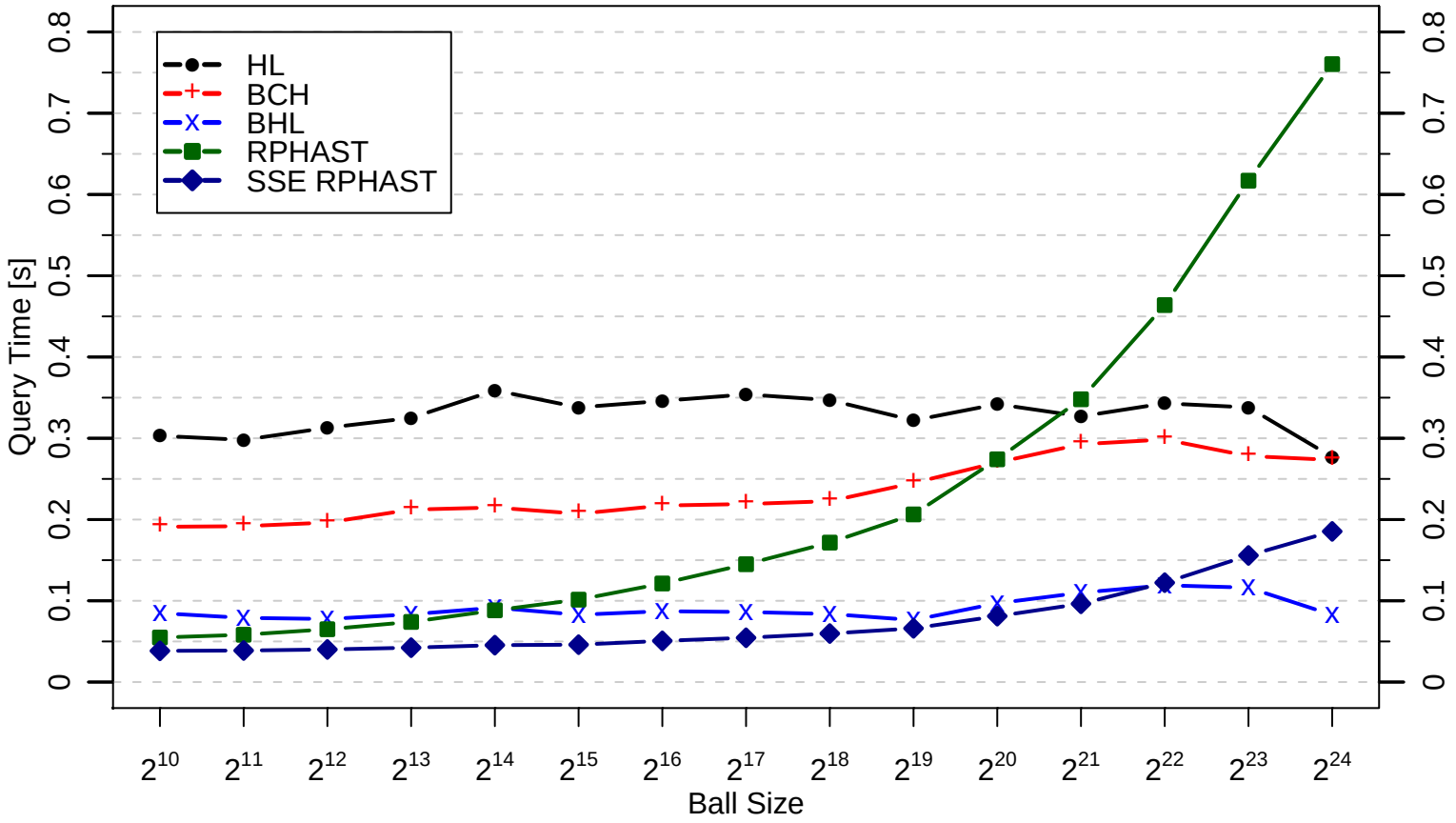
- vehicle routing
- traveling salesman

**Lösung:**

- $|S|$ one-to-many Anfragen
- speicher Distanzen in der Tabelle
- RPHAST kann multiples Setup (SSE) nutzen

# Experimente I

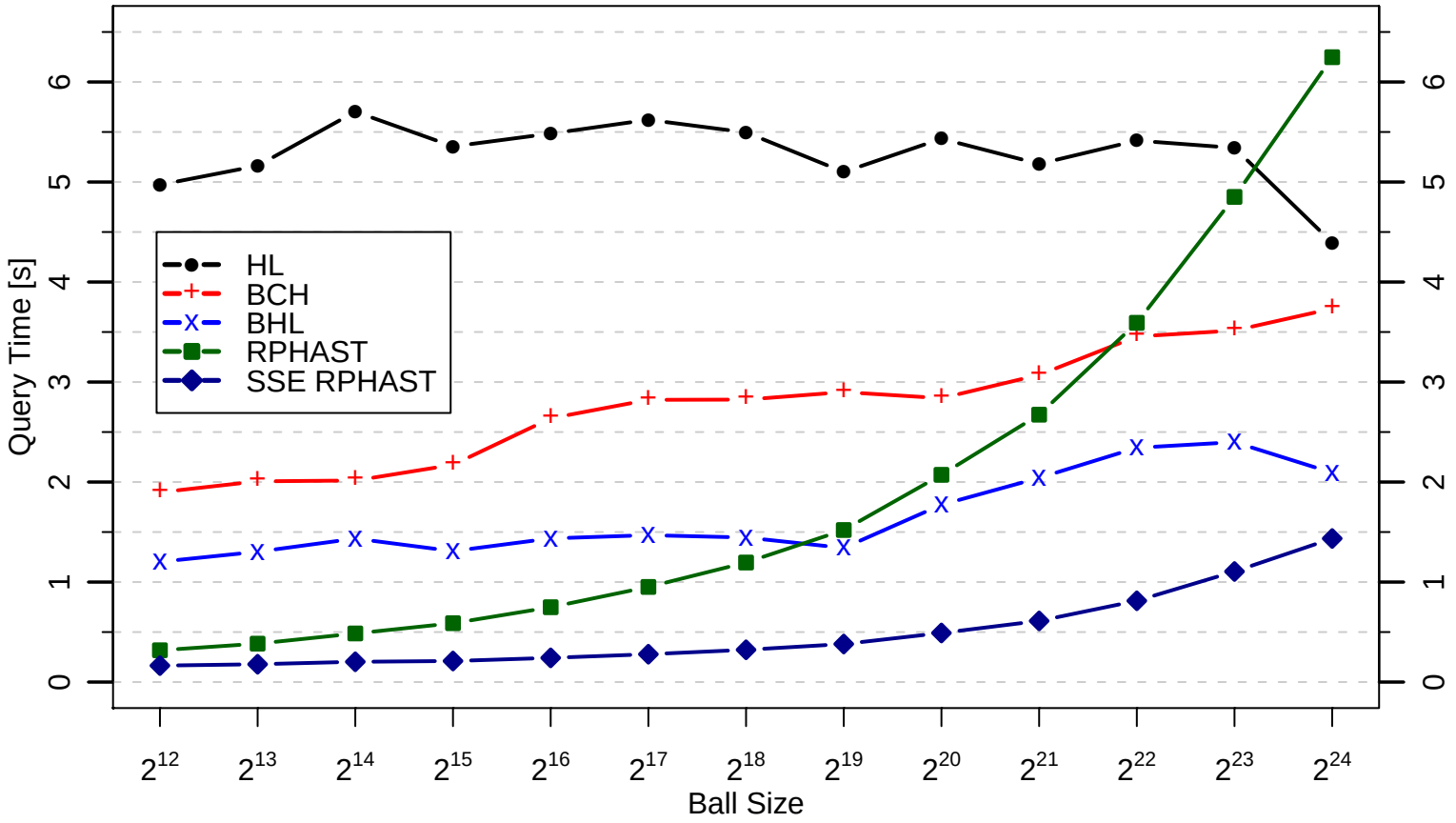
**input:** Westeuropa (18M Knoten), $|S| = |T| = 2^{10}$




**Beobachtung:** alle Techniken unter einer Sekunde

# Experimente II

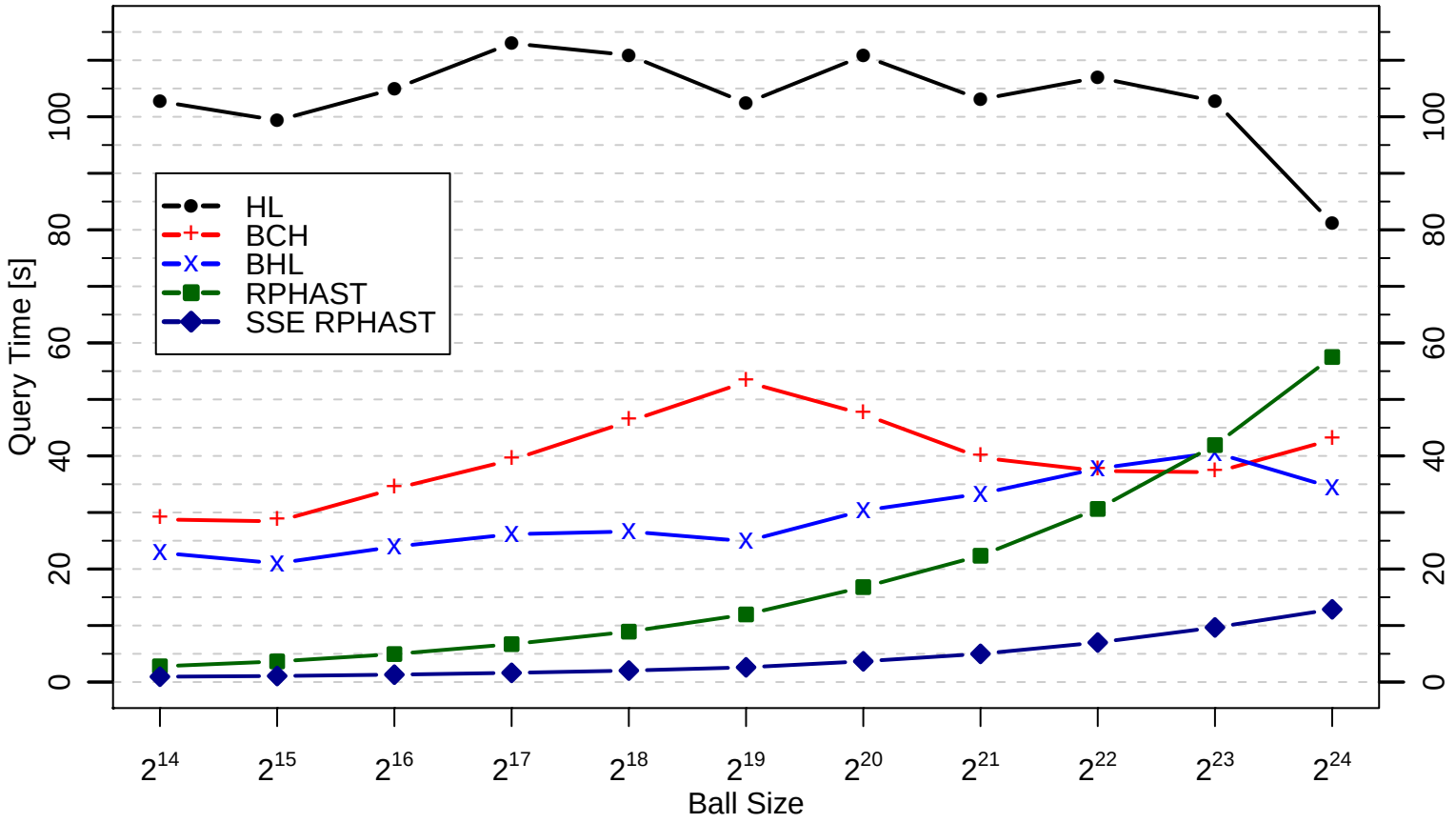
**input:** Westeuropa (18M Knoten), $|S| = |T| = 2^{12}$




**Beobachtung:** SSE PHAST am schnellsten

# Experimente III

**input:** Westeuropa (18M Knoten), $|S| = |T| = 2^{14}$




**Beobachtung:** SSE PHAST am schnellsten

# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

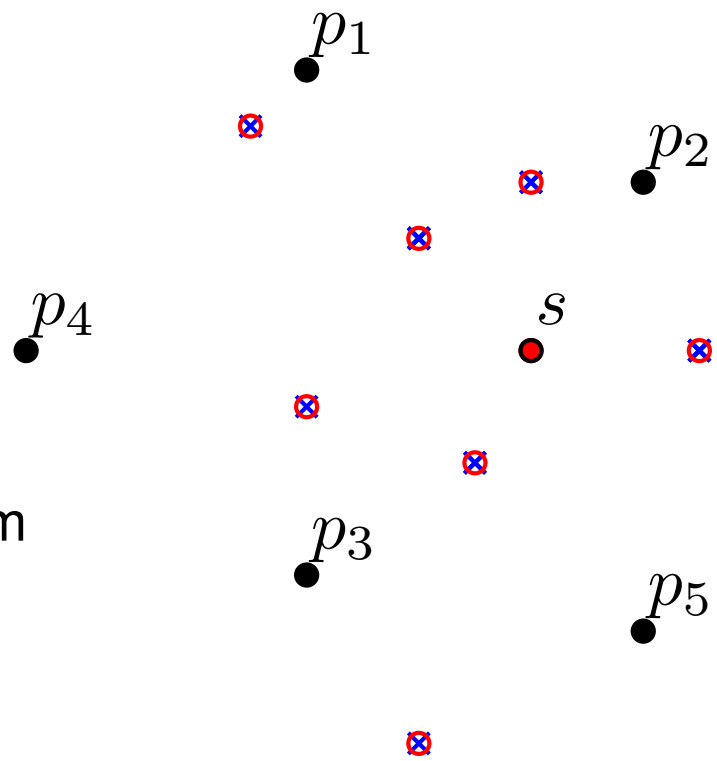
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



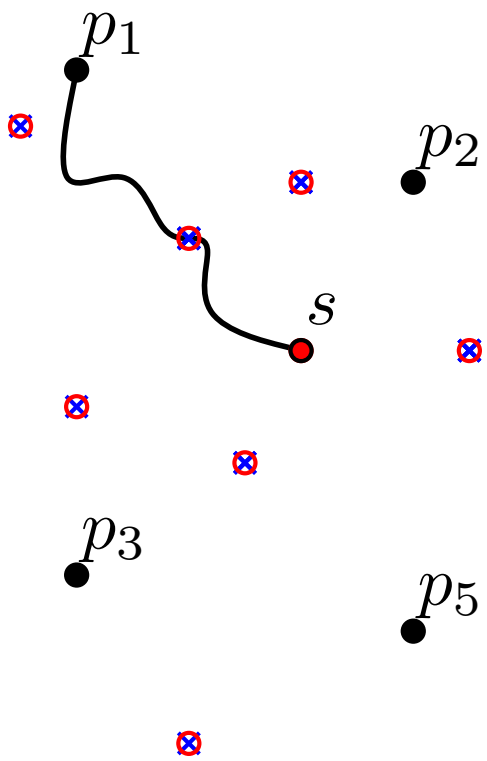
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



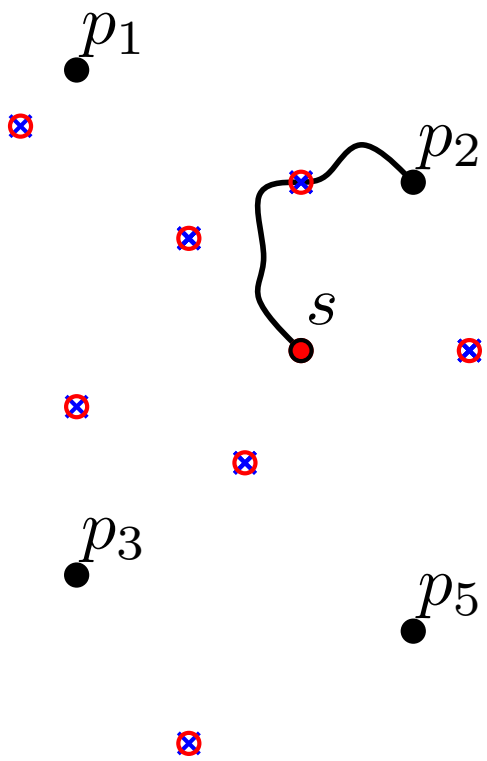
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



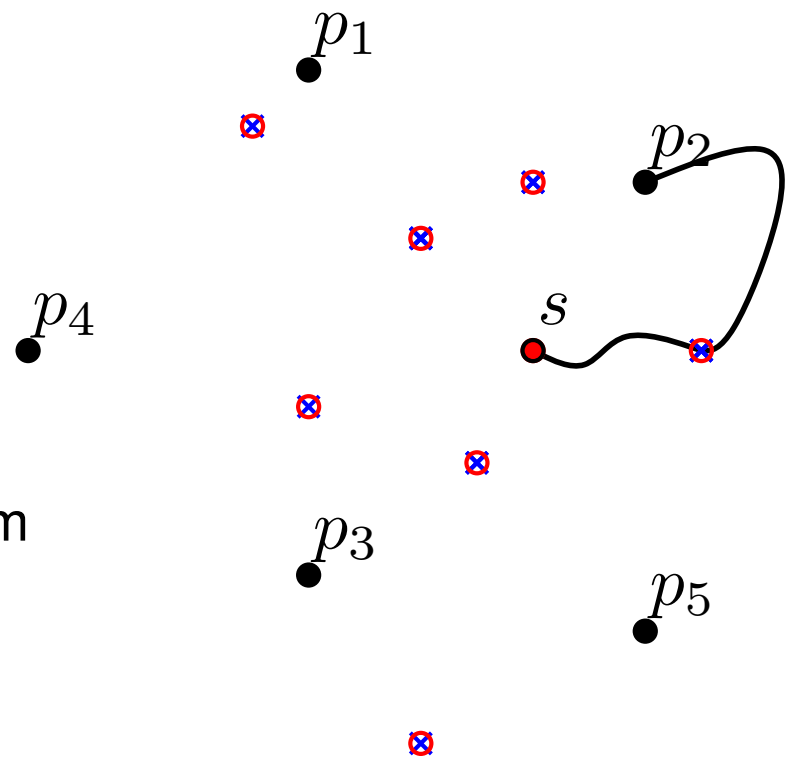
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



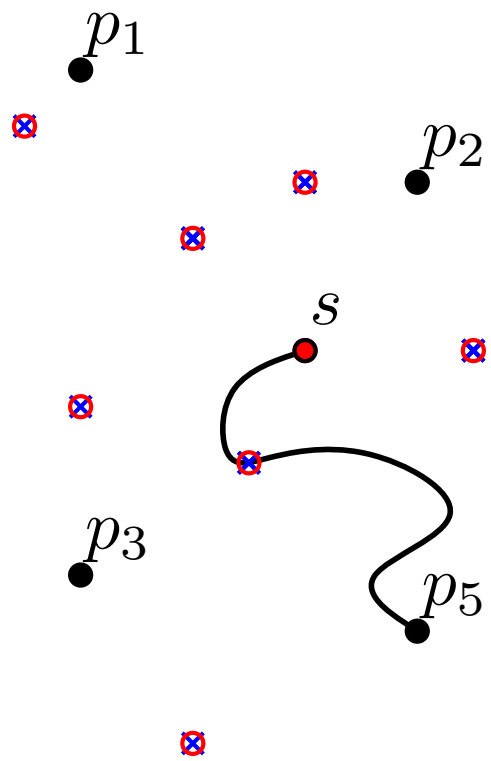
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



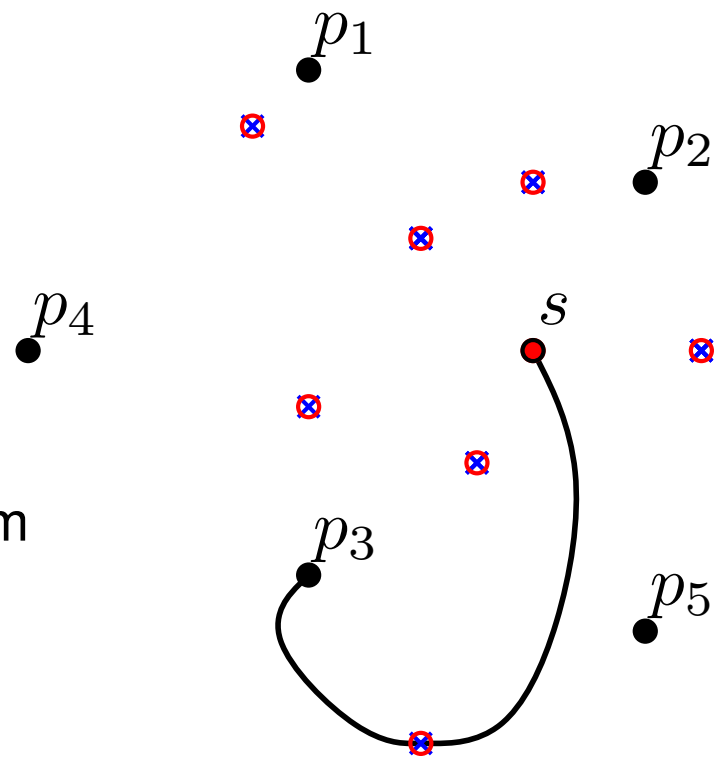
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



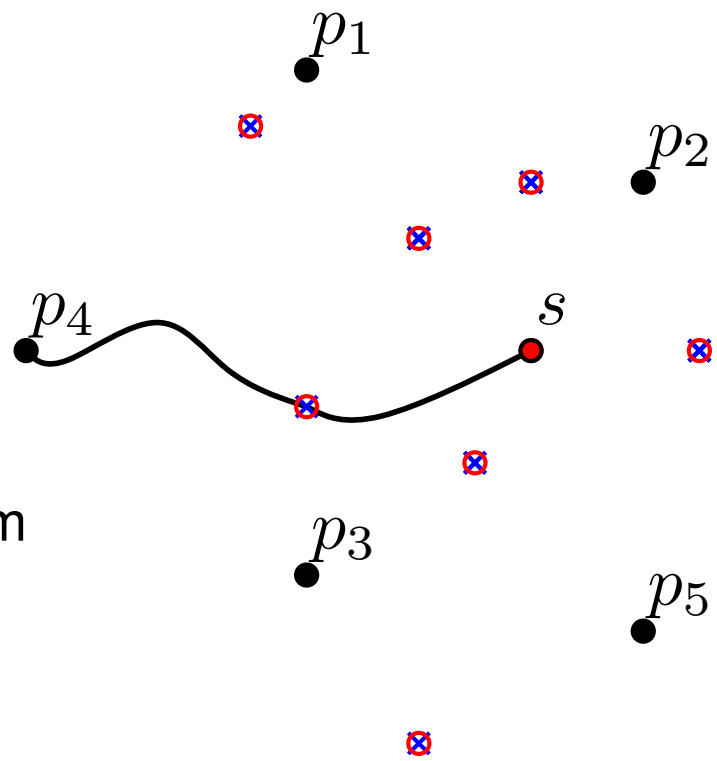
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



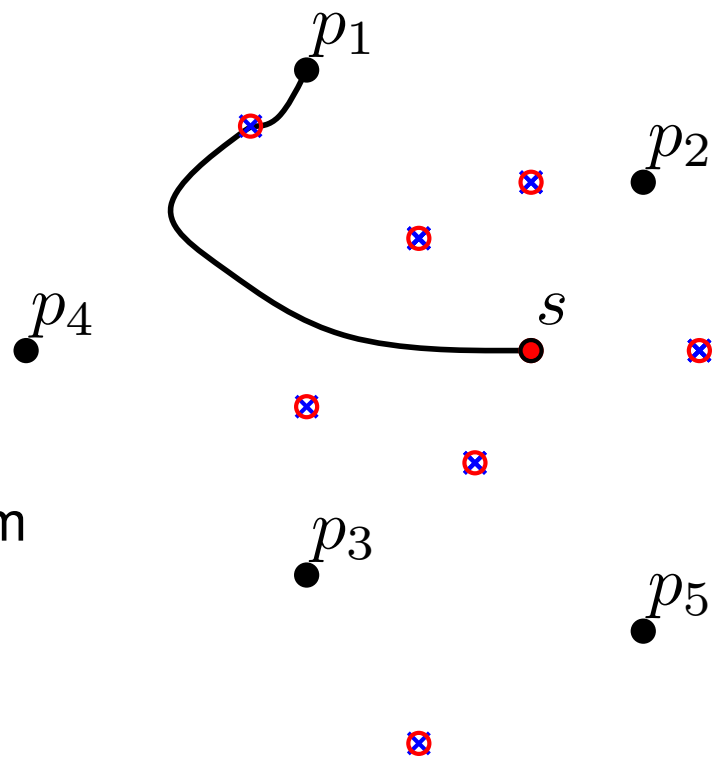
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



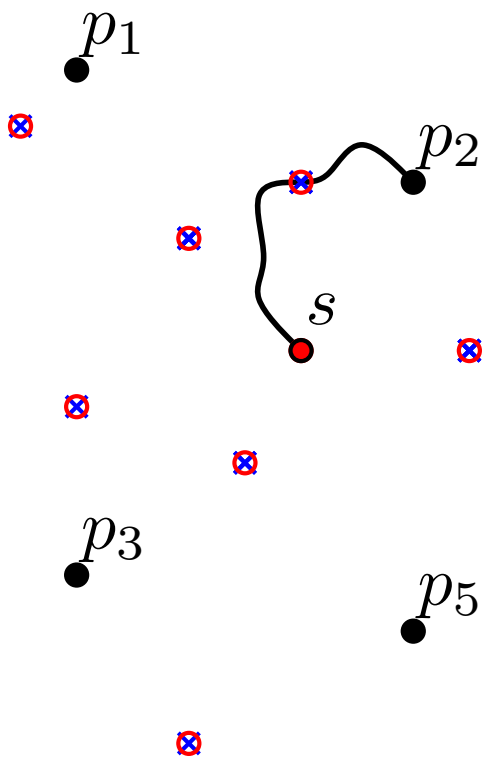
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



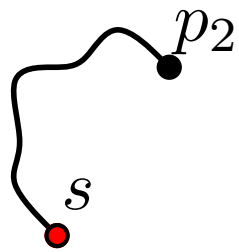
# Nächster POI

**Szenario:**

- Zielknoten sind POIs (z.B. Paketshops)
- finde $k$ nächste POIs von einem Startknoten $s$

**Lösung:**

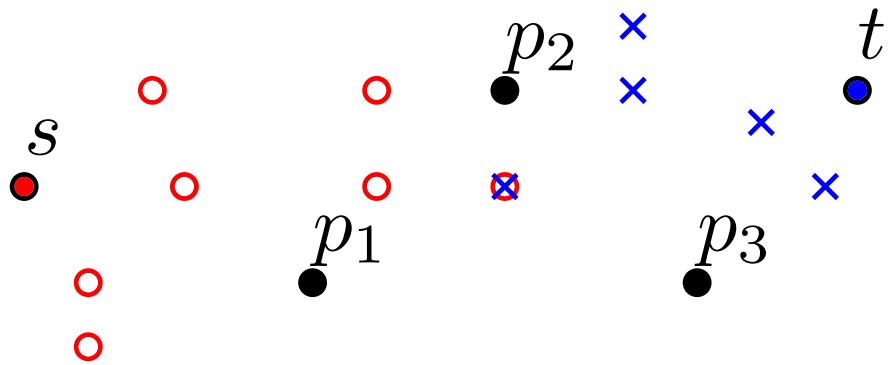
- wie one-to-many
- ordne die buckets pro Knoten auch nach aufsteigender Distanz
- in jedem Bucket müssen nur die $k$ nächsten POIs durchsucht werden
- Laufzeit für POI Query nicht abhängig von Anzahl POIs im System
- Laufzeit: Suchraum $\cdot$ $k$



# Bester Via Knoten

**Szenario:**

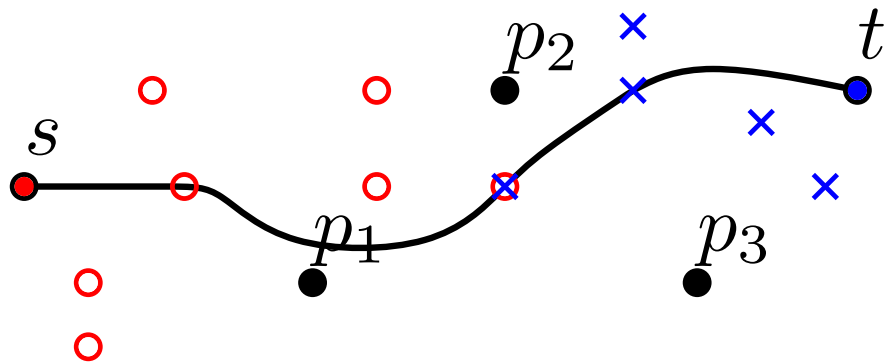
- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$



# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$



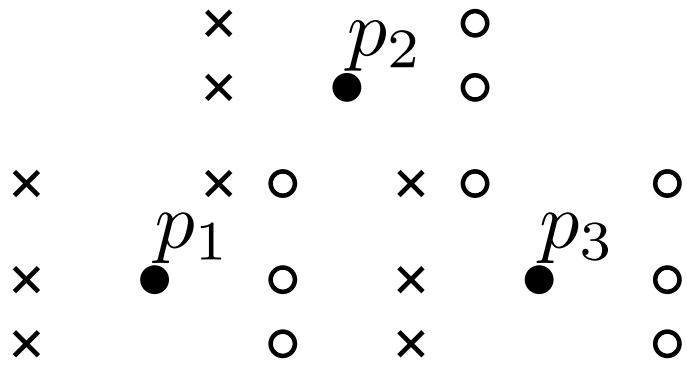
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



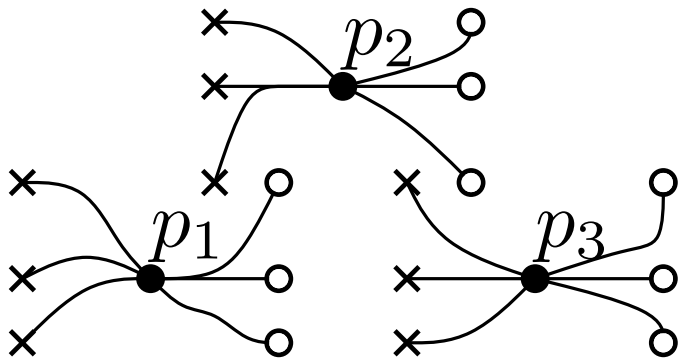
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



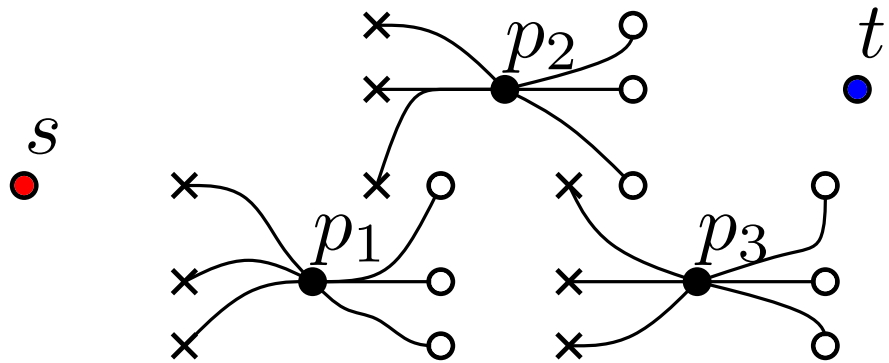
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



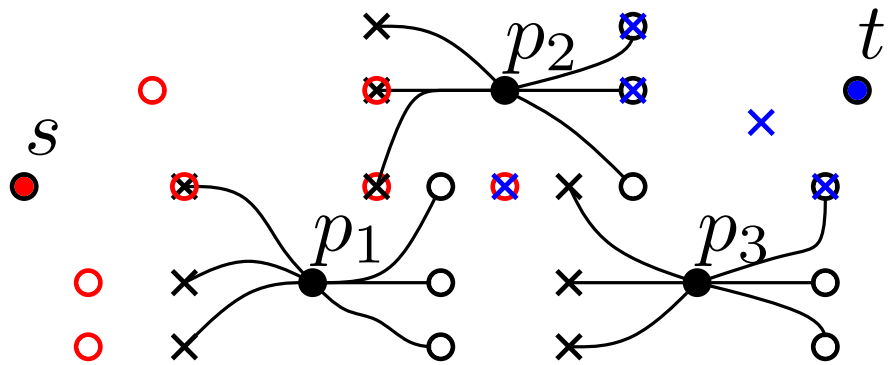
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



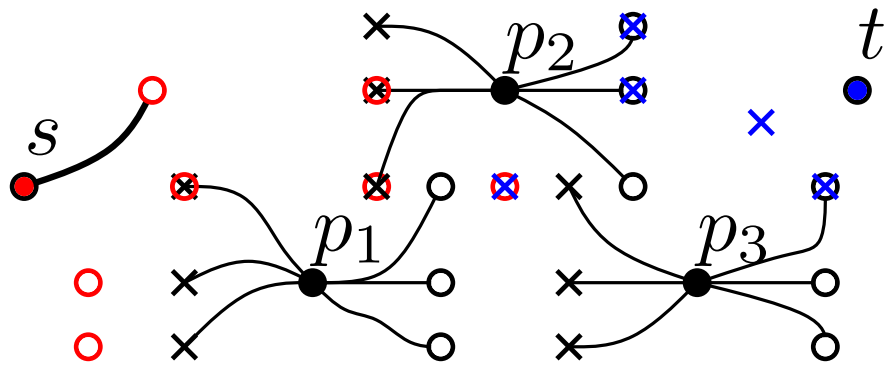
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



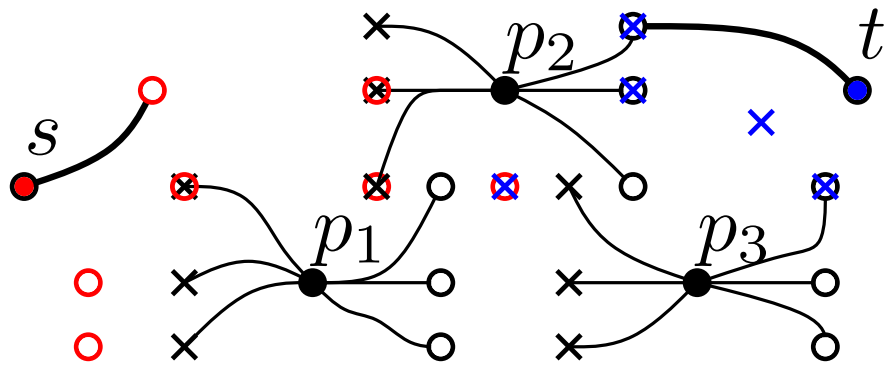
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



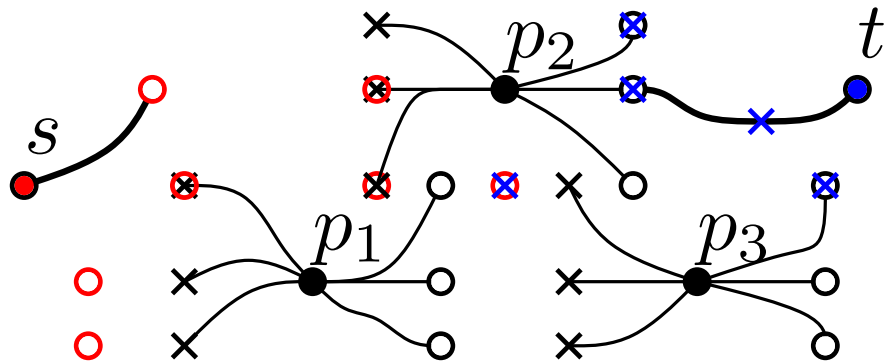
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



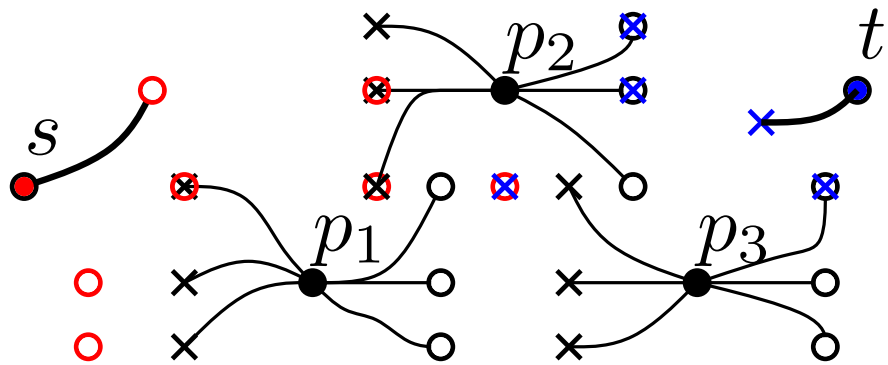
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



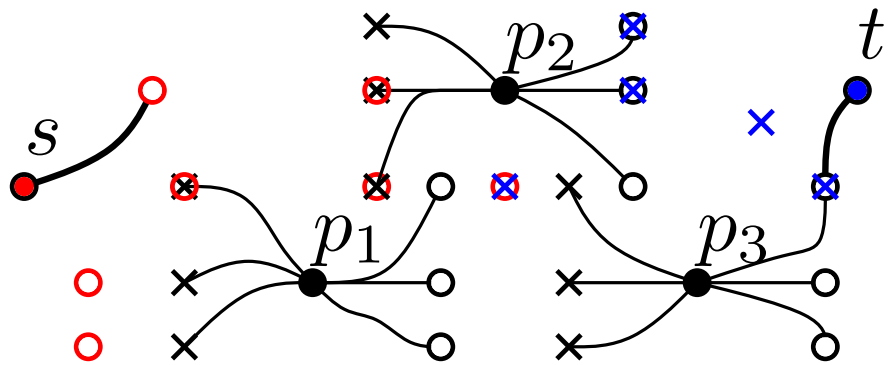
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



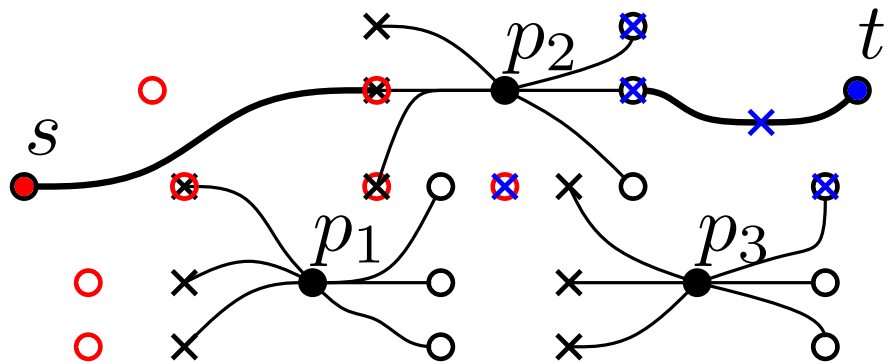
# Bester Via Knoten

**Szenario:**

- finde $k$ best Via Knoten POIs von einem Startknoten $s$ zu einem Zielknoten $t$
- minimiere $\text{dist}(s, p) + \text{dist}(p, t)$ über alle POIs $p$

**Lösung:**

- Vorwärts- und Rückwartssuche von jedem POI
- speicher Kreuzprodukt der beiden Suchräume mit Distanz durch den POI
- Such von $s$ und $t$: evaluiere jedes Paar
- Laufzeit: Suchraum$^2 \cdot k$



# Wiederholung: HLDB SQL Query

Tabellen `forward` und `backward`
mit Spalten `node`, `hub`, `dist`

**Algorithm:** SQL_DIST

**Input**: source $s \in V$, target $t \in V$

```
1 SELECT
2       MIN(forward.dist+backward.dist)
3 FROM forward,backward
4 WHERE
5       forward.node = s AND
6       backward.node = t AND
7       forward.hub = backward.hub
```

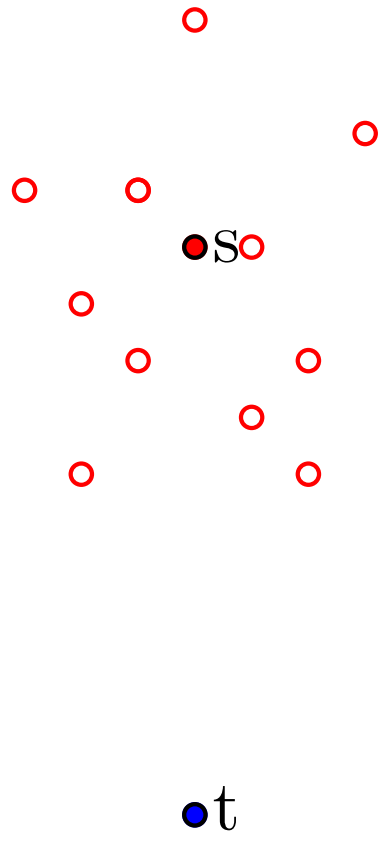
# Wiederholung: HLDB SQL Query

Tabellen `forward` und `backward`
mit Spalten `node`, `hub`, `dist`

**Algorithm:** SQL_DIST

**Input**: source $s \in V$, target $t \in V$

```
1 SELECT
2       MIN(forward.dist+backward.dist)
3 FROM forward,backward
4 WHERE
5       forward.node = s AND
6       backward.node = t AND
7       forward.hub = backward.hub
```



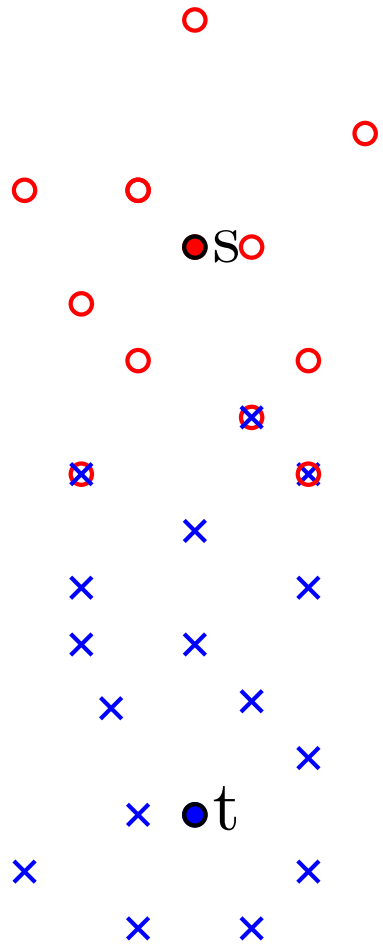
# Wiederholung: HLDB SQL Query

Tabellen `forward` und `backward`
mit Spalten `node`, `hub`, `dist`

**Algorithm:** SQL_DIST

**Input**: source $s \in V$, target $t \in V$

```
1 SELECT
2       MIN(forward.dist+backward.dist)
3 FROM forward,backward
4 WHERE
5       forward.node = s AND
6       backward.node = t AND
7       forward.hub = backward.hub
```



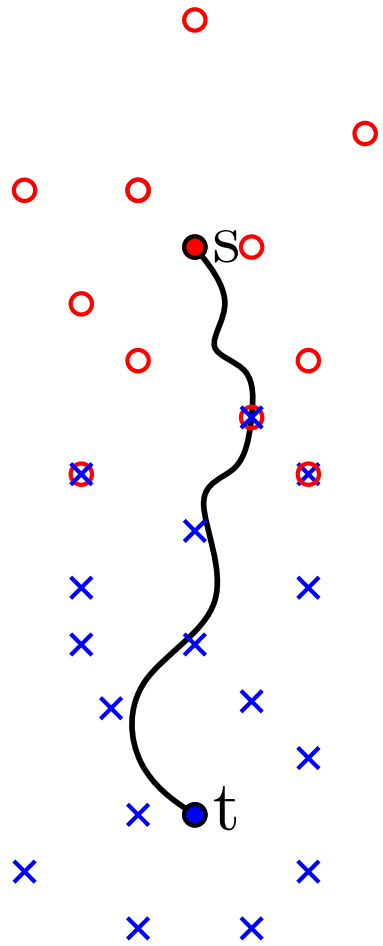
# Wiederholung: HLDB SQL Query

Tabellen `forward` und `backward`
mit Spalten `node`, `hub`, `dist`

**Algorithm:** SQL_DIST

**Input**: source $s \in V$, target $t \in V$

```
1 SELECT
2       MIN(forward.dist+backward.dist)
3 FROM forward,backward
4 WHERE
5       forward.node = s AND
6       backward.node = t AND
7       forward.hub = backward.hub
```



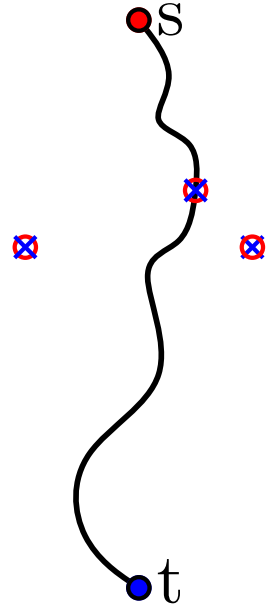
# Wiederholung: HLDB SQL Query

Tabellen `forward` und `backward`
mit Spalten `node`, `hub`, `dist`

**Algorithm:** SQL_DIST

**Input**: source $s \in V$, target $t \in V$

```
1 SELECT
2       MIN(forward.dist+backward.dist)
3 FROM forward,backward
4 WHERE
5       forward.node = s AND
6       backward.node = t AND
7       forward.hub = backward.hub
```



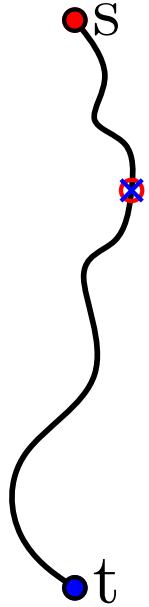
# Wiederholung: HLDB SQL Query

Tabellen `forward` und `backward`
mit Spalten `node`, `hub`, `dist`

**Algorithm:** SQL_DIST

**Input**: source $s \in V$, target $t \in V$

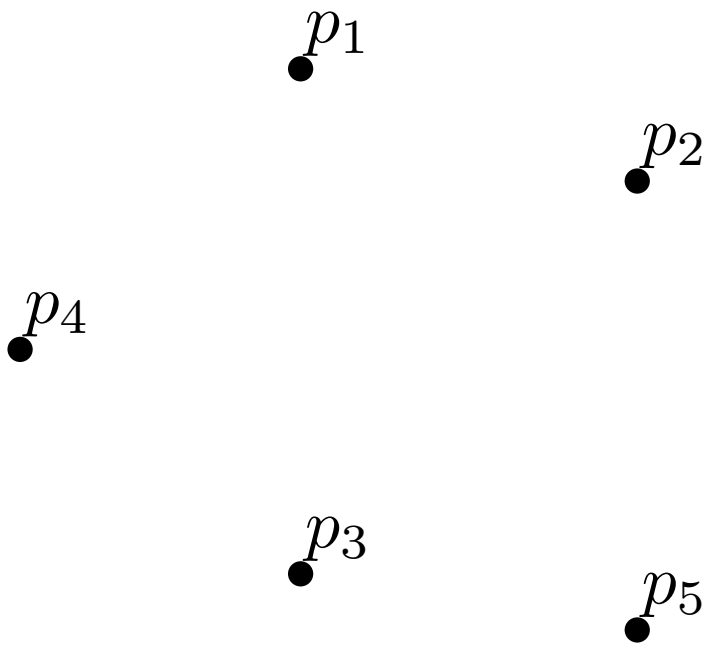
```
1 SELECT
2       MIN(forward.dist+backward.dist)
3 FROM forward,backward
4 WHERE
5       forward.node = s AND
6       backward.node = t AND
7       forward.hub = backward.hub
```



# POI Anfragen in SQL

**Idee:**

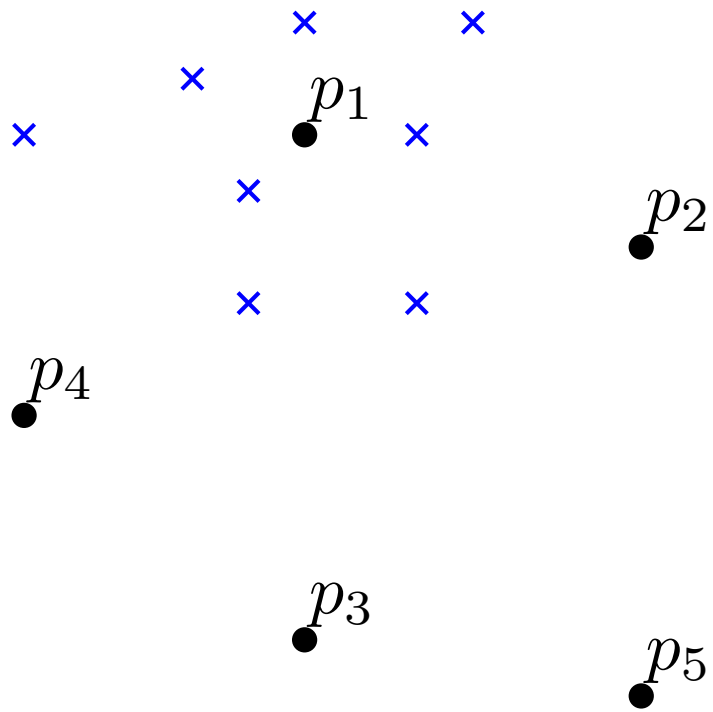
- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



# POI Anfragen in SQL

**Idee:**

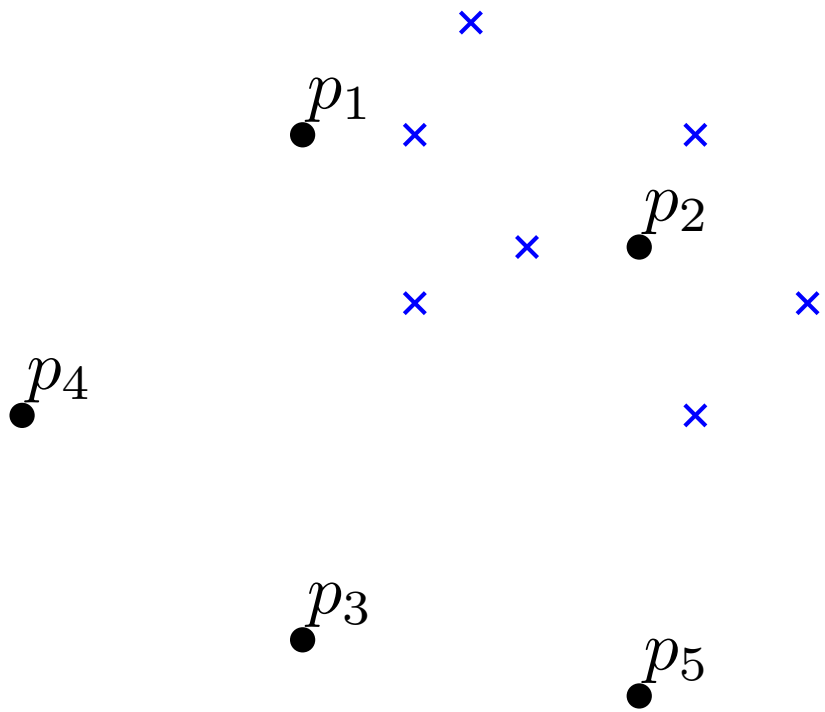
- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



# POI Anfragen in SQL

**Idee:**

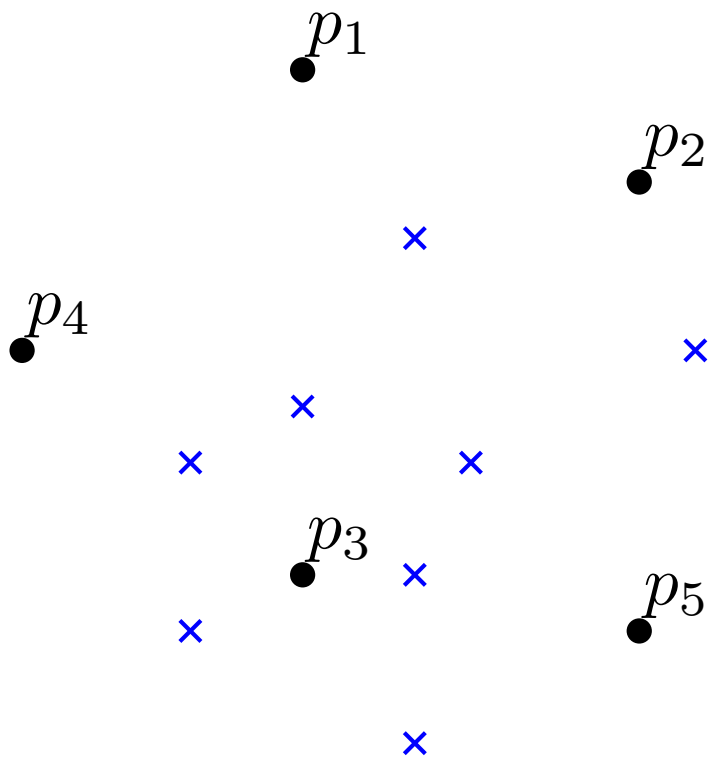
- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



# POI Anfragen in SQL

**Idee:**

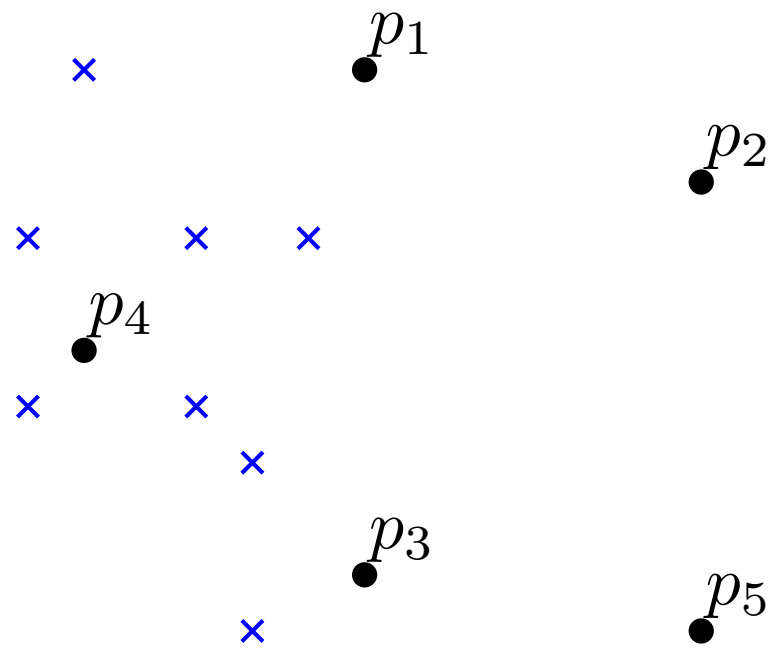
- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



# POI Anfragen in SQL

**Idee:**

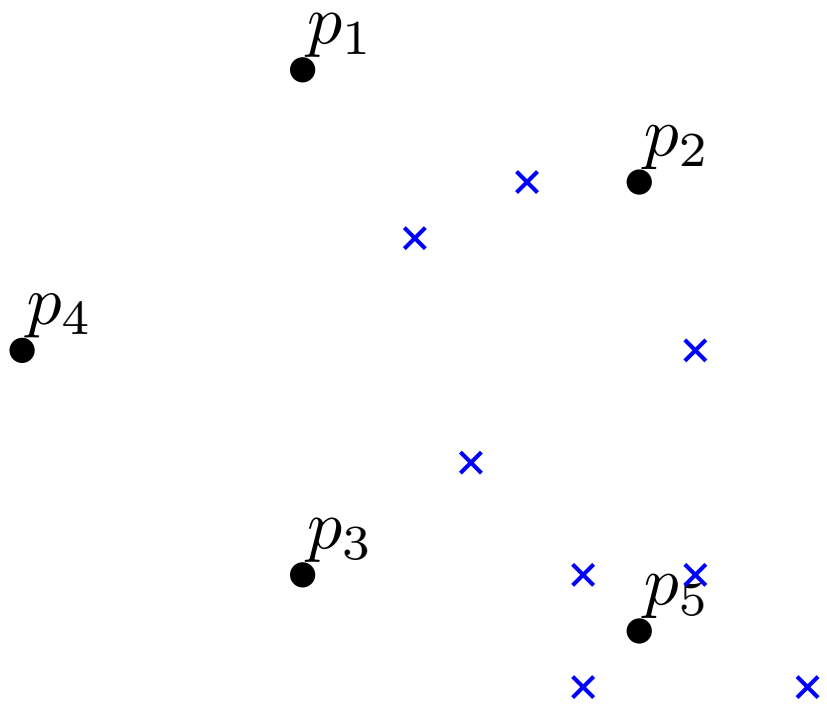
- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



# POI Anfragen in SQL

**Idee:**

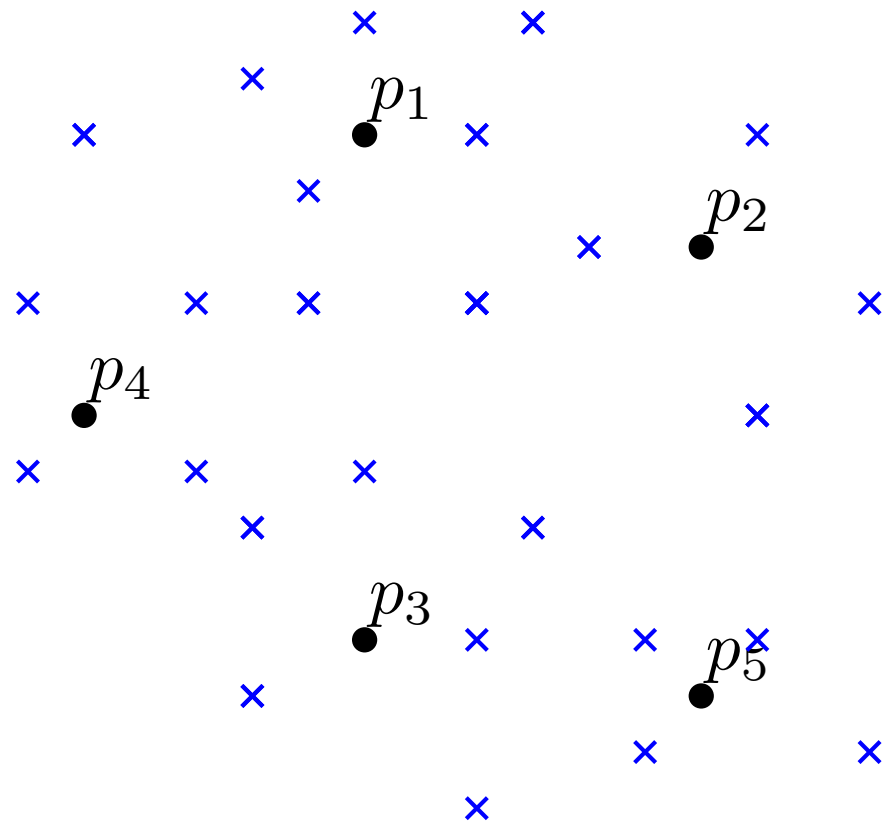
- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



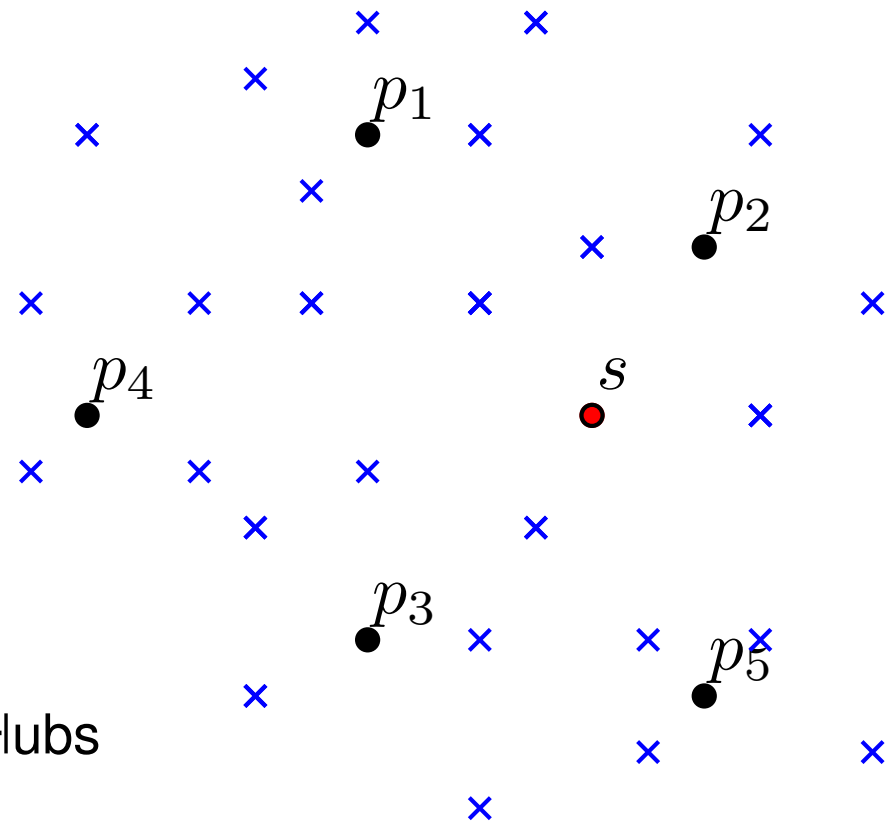
# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`

**Query:**

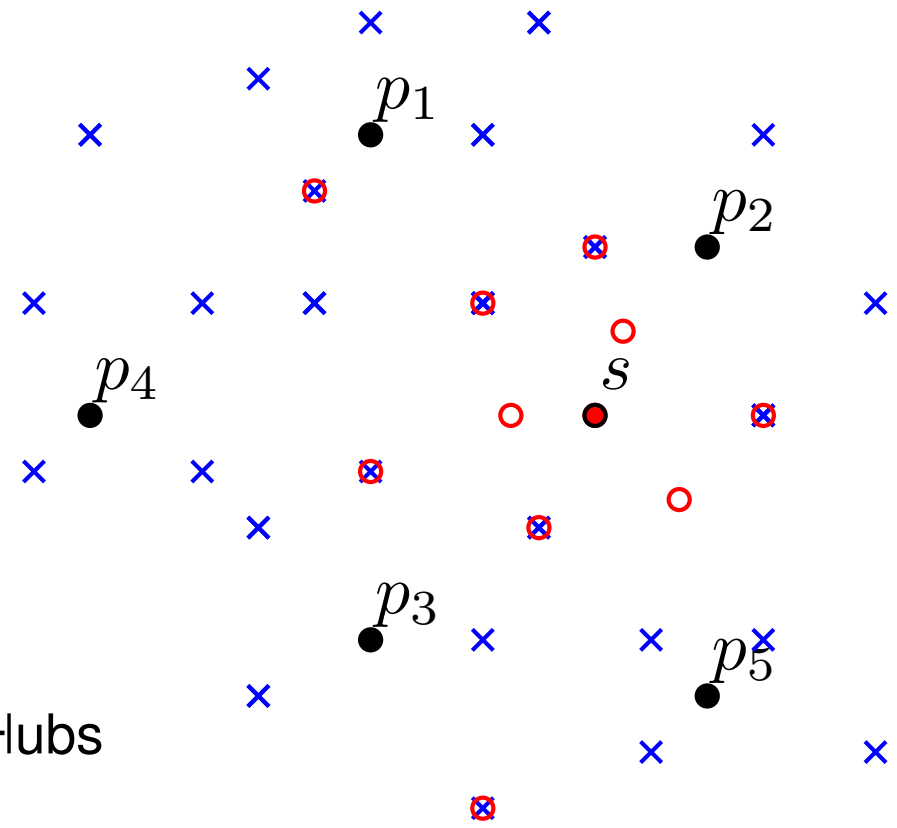
- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs



# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



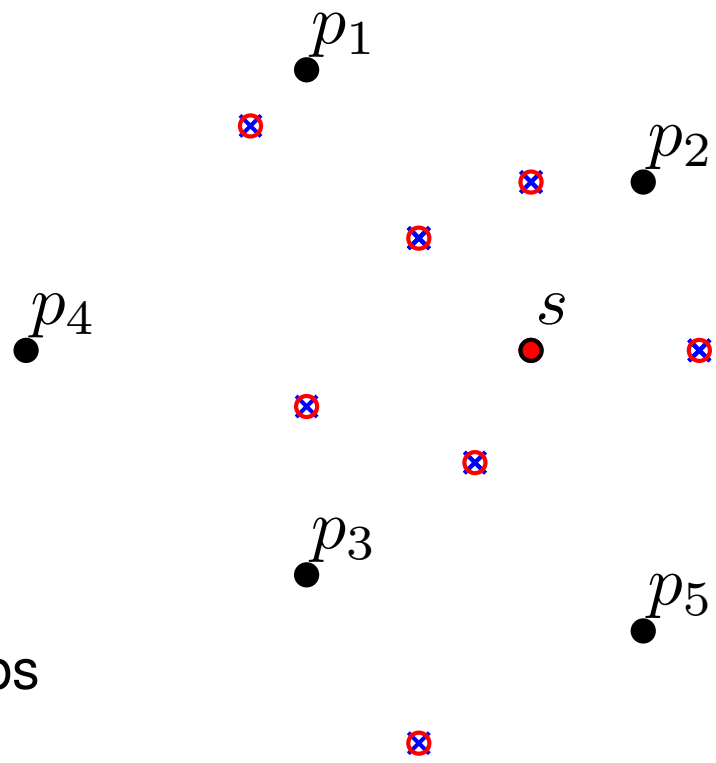

**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`




**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

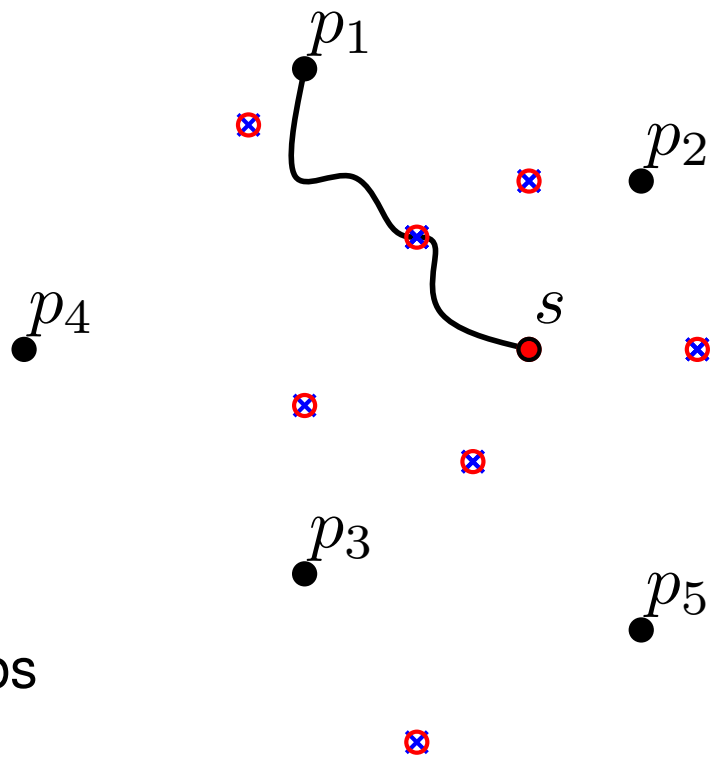
# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`

**Query:**

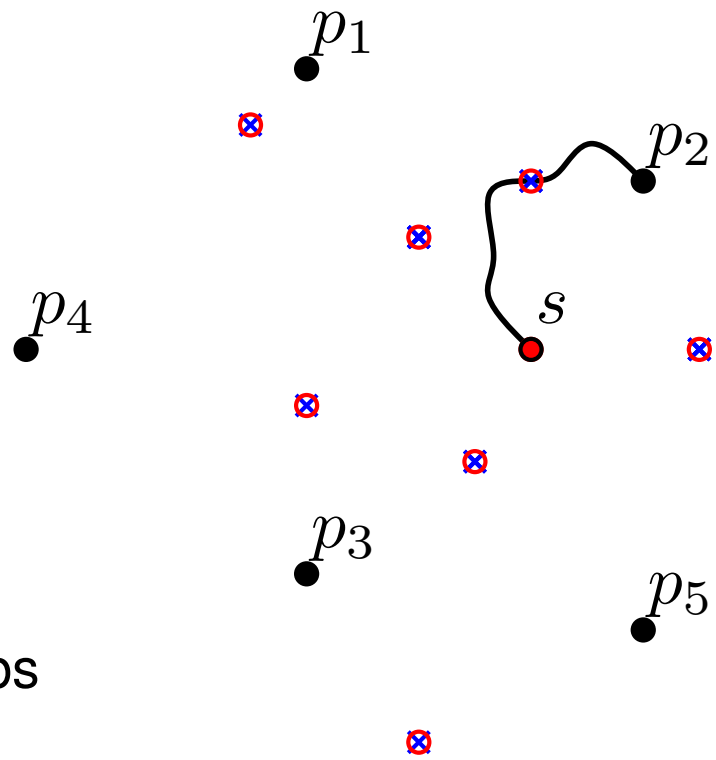
- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs



# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



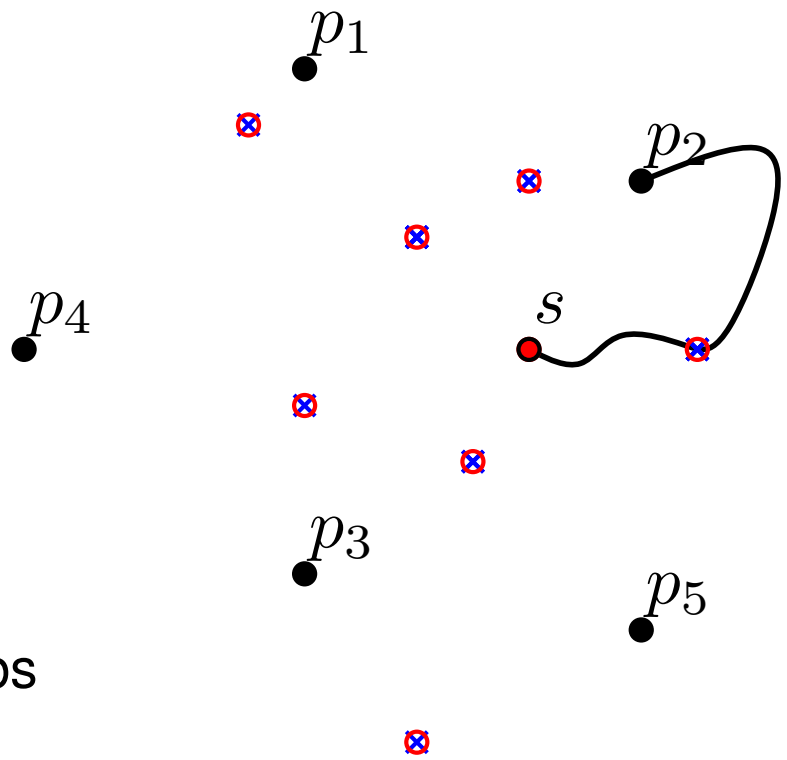

**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`




**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

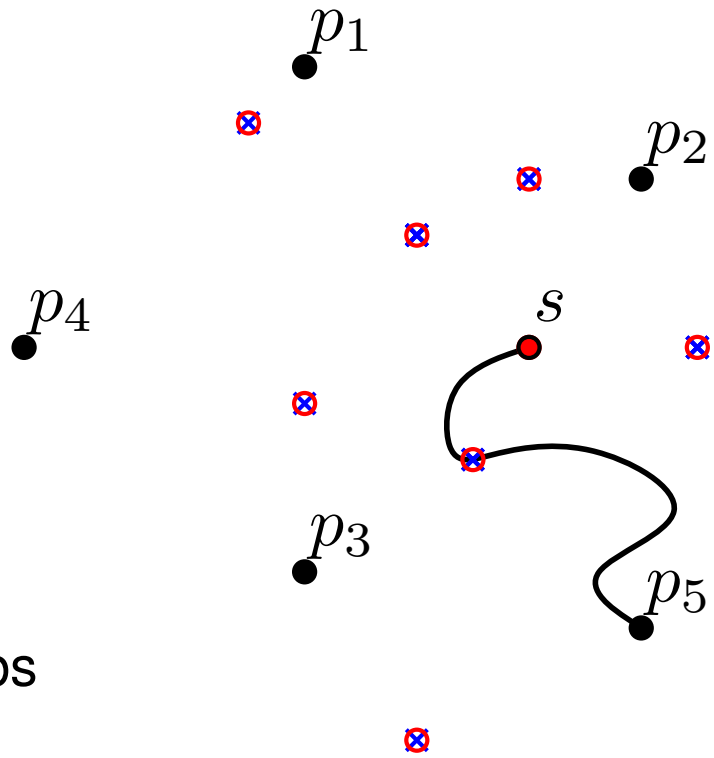
# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`

**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs



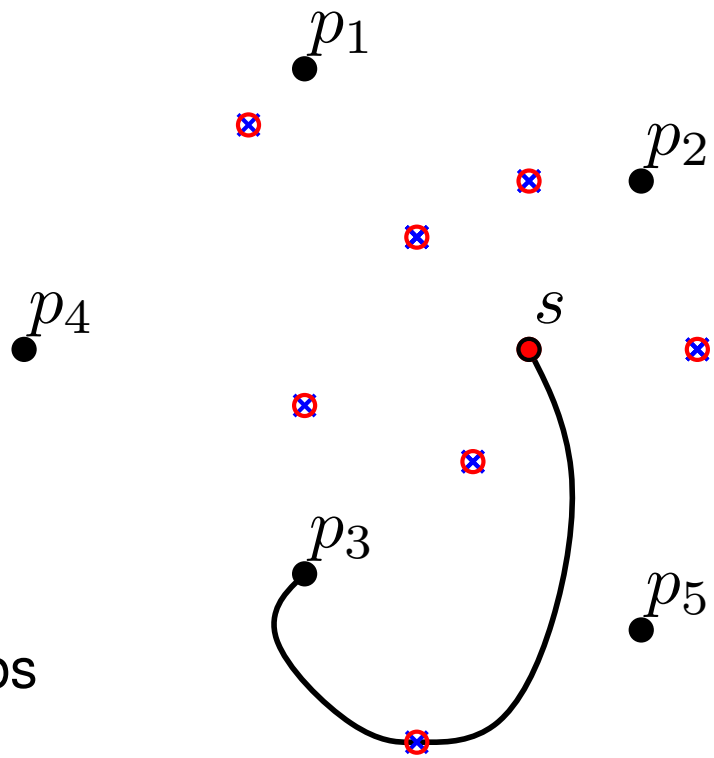
# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`

**Query:**

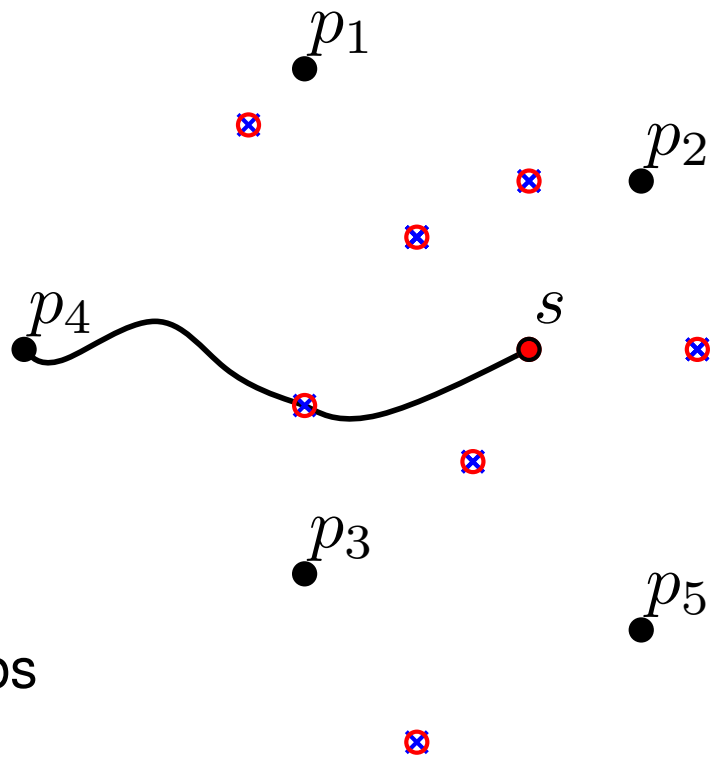
- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs



# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



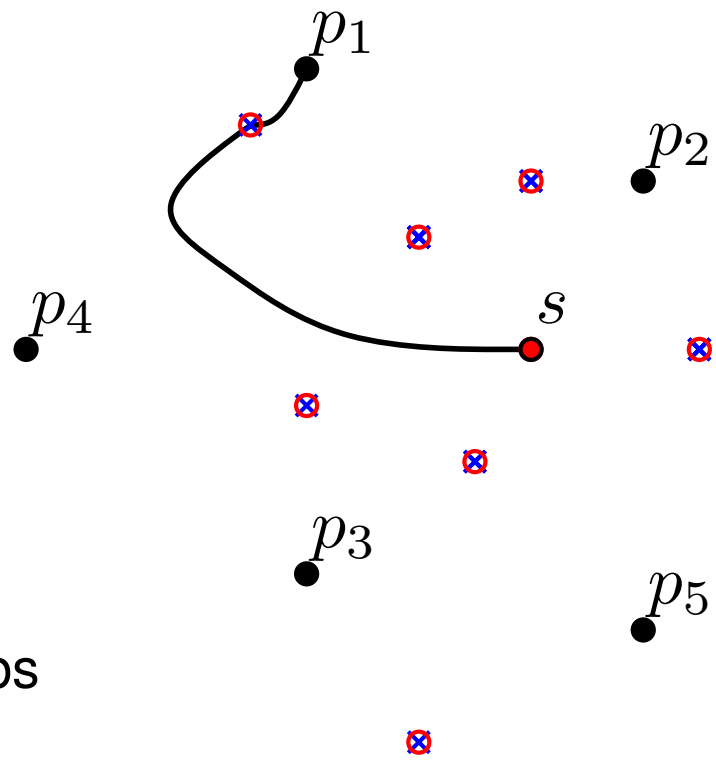

**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`



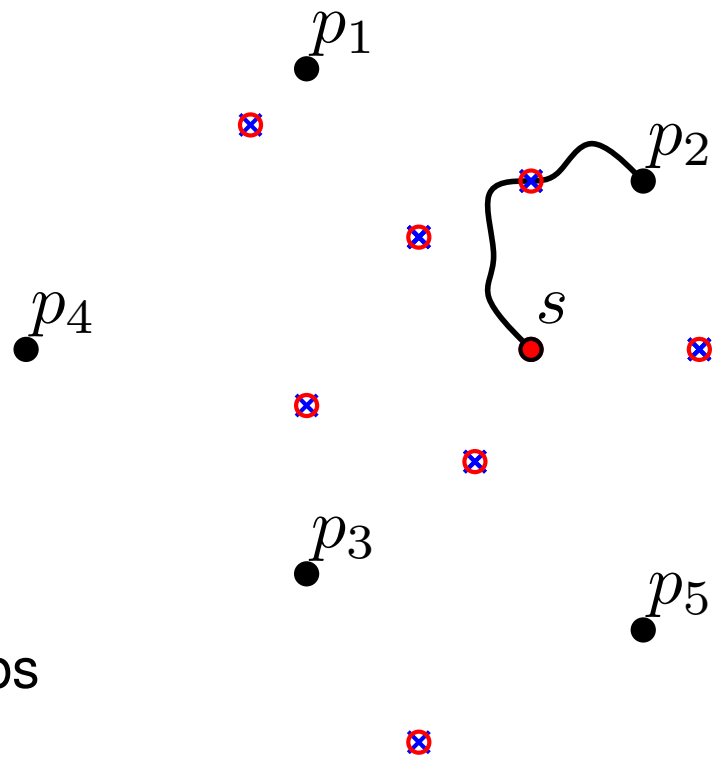

**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

# POI Anfragen in SQL

**Idee:**

- extrahiere die Rückwartslabel aus `backward`
- speicher sie in neuer Tabelle `poilab`
- indiziere nach `hub` und `dist`

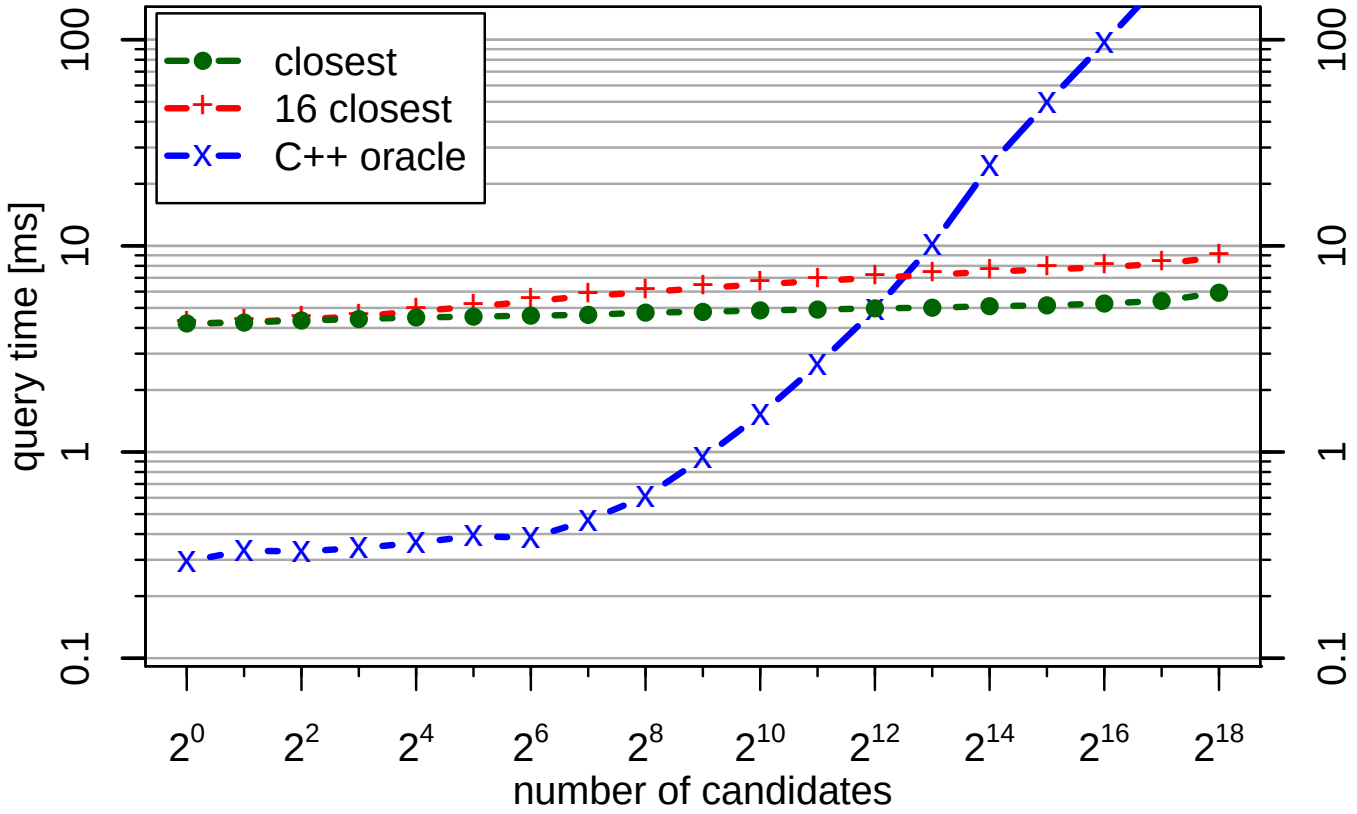



**Query:**

- iteriert über alle ausgehenden Hubs
- für jeden Hub werden nur die ($k$) nächsten POIs betrachtet
- Antwort: die $k$ insgesamt nächsten POIs

# Ergebnisse POI

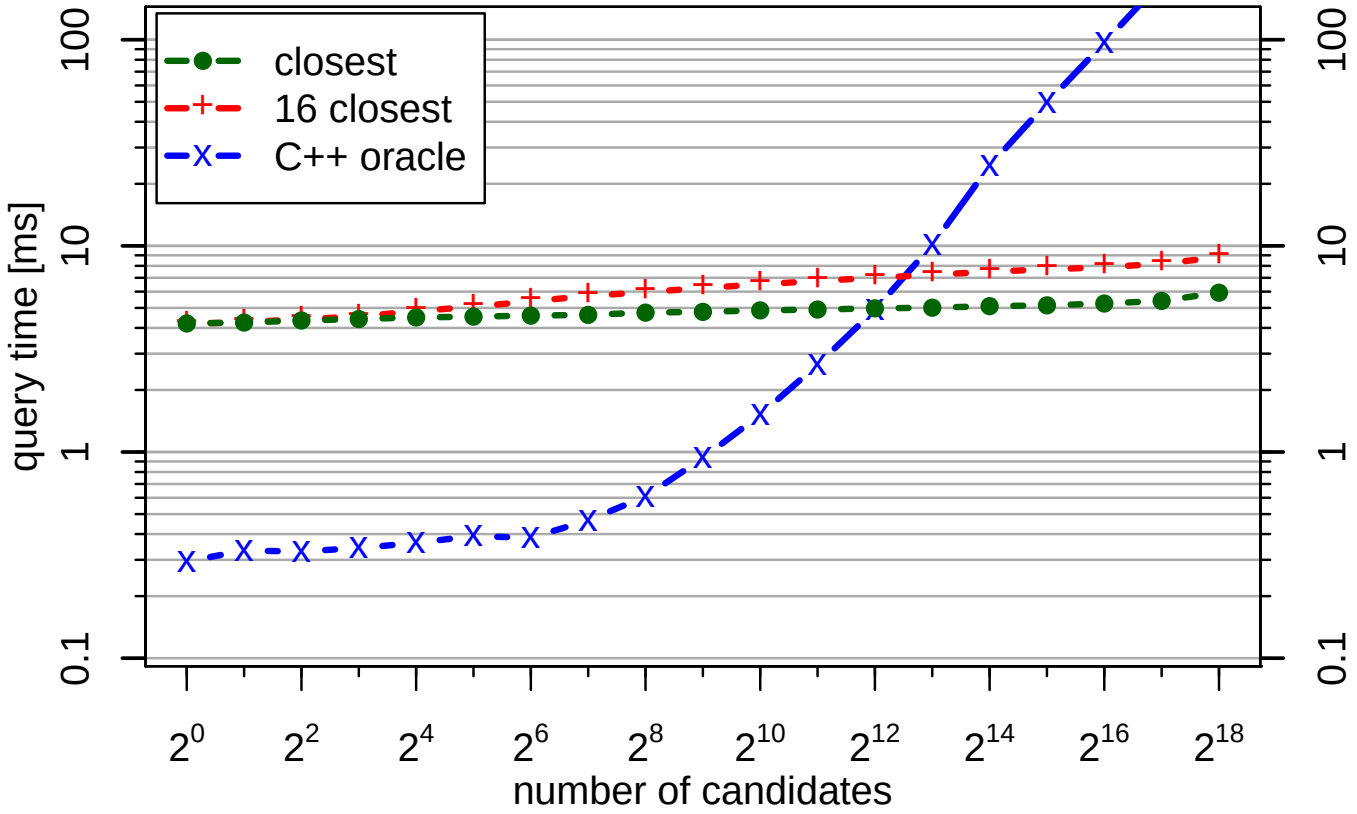
**Setup:** verschiedene Anzahl POIs, zufällig gewählt




- externes Punkt-zu-Punkt Orakel: skaliert schlecht

# Ergebnisse POI

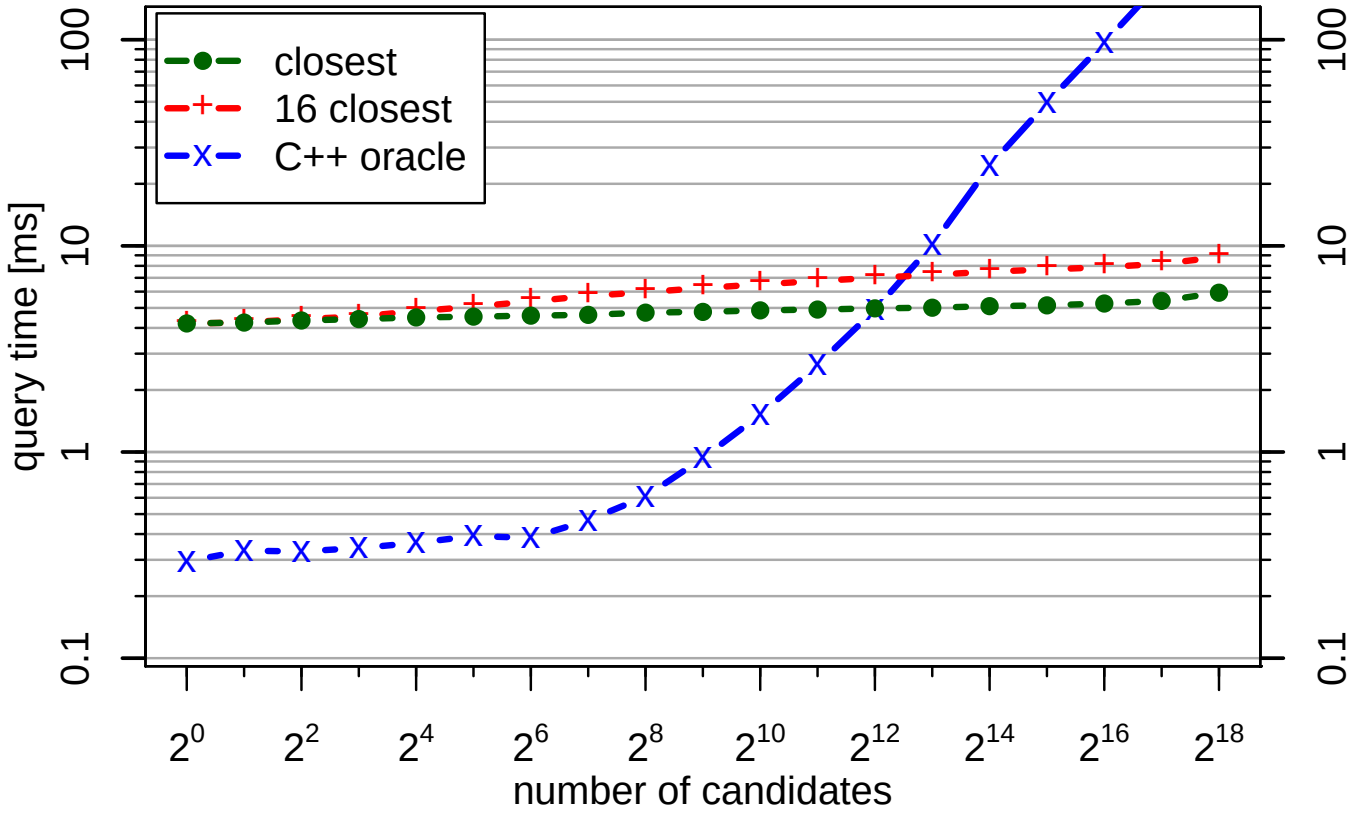
**Setup:** verschiedene Anzahl POIs, zufällig gewählt




- externes Punkt-zu-Punkt Orakel: skaliert schlecht
- SQL Anfragen unabhängig von Anzahl POIs

# Ergebnisse POI

**Setup:** verschiedene Anzahl POIs, zufällig gewählt




- externes Punkt-zu-Punkt Orakel: skaliert schlecht
- SQL Anfragen unabhängig von Anzahl POIs
- weitere Constraints einfach (“jetzt geöffnet”)

# Isochronen



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

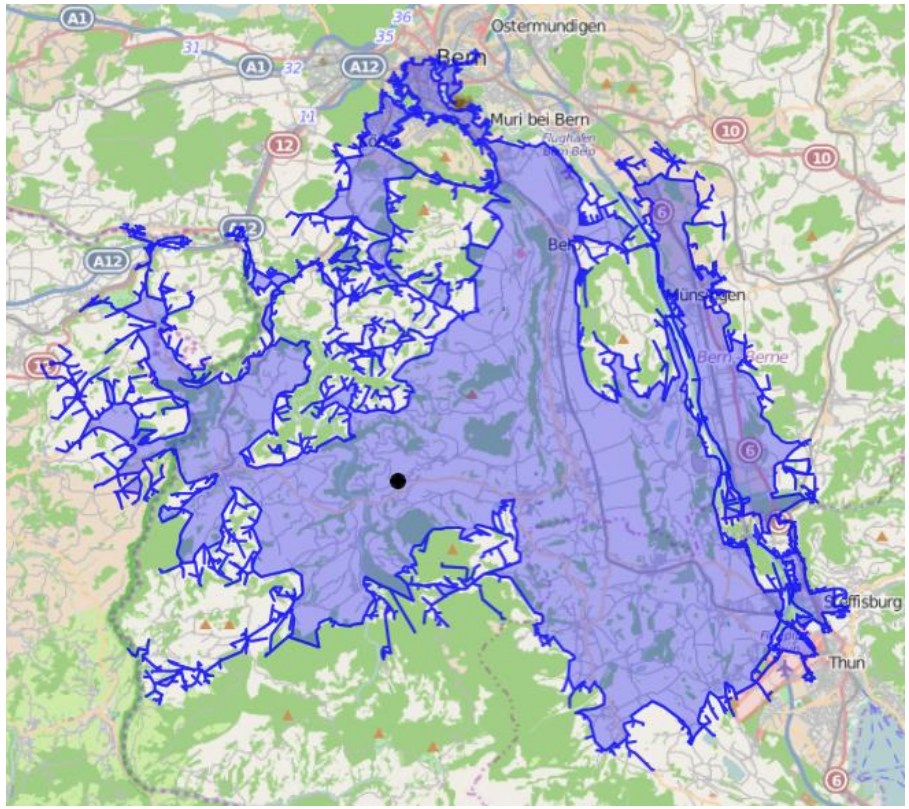
# Isochronen und ihre Anwendungen

## Isochronen:

Innerhalb einer gewissen Zeit von einem fixen Startpunkt erreichbare Region

## Praktische Anwendungen:

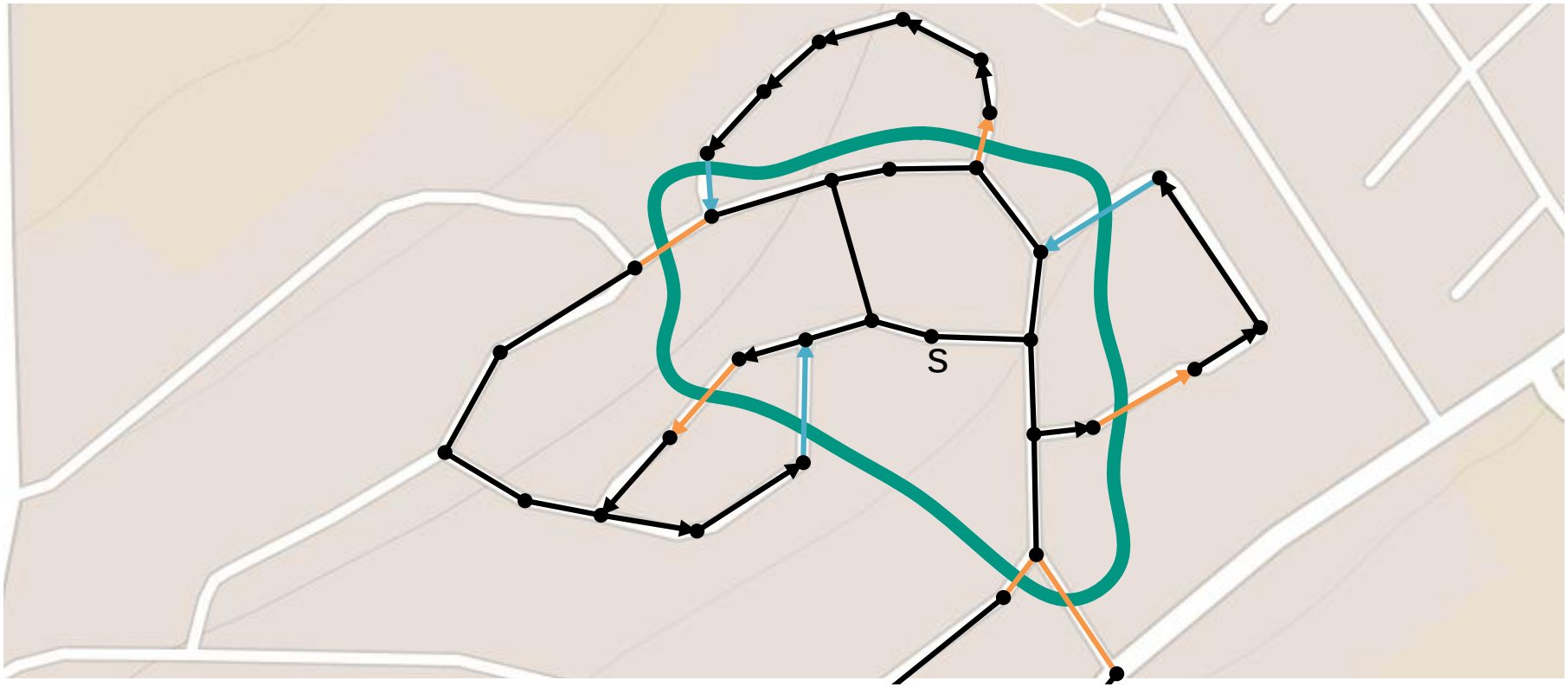
- Erreichbarkeitsanalysen in der Städteplanung
- Geomarketing
- Anzeige der verbleibenden Reichweite eines Fahrzeugs



# Problemstellung

## Isochronenproblem:

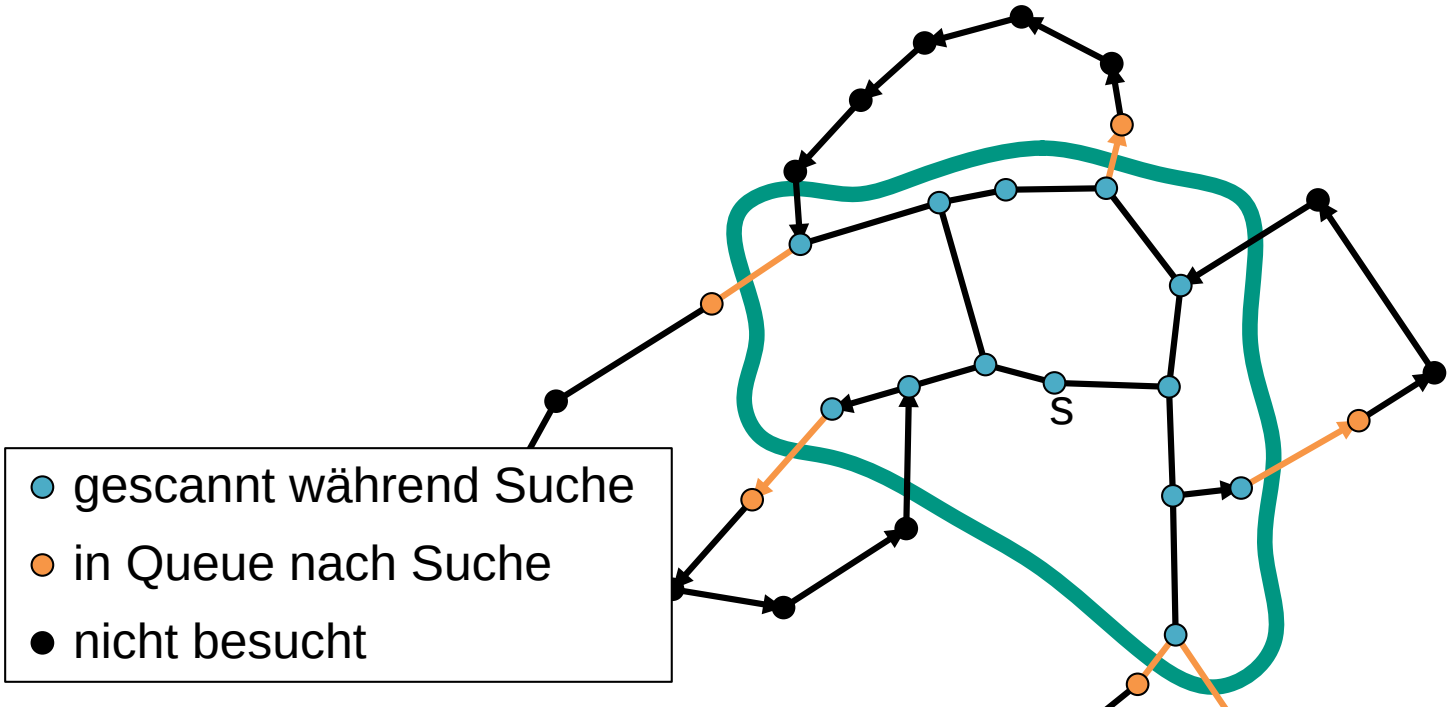
- Eingabe: Graph $G = (V, E)$, Funktion $\ell: E \to \mathbb{R}_{\geq 0}$, Startknoten $s$, Zeitlimit $\tau$
- Ausgabe: alle ausgehenden und eingehenden Isochronenkanten



# Isochronen berechnen

**Variante von Dijkstras Algorithmus (isoDijkstra):**

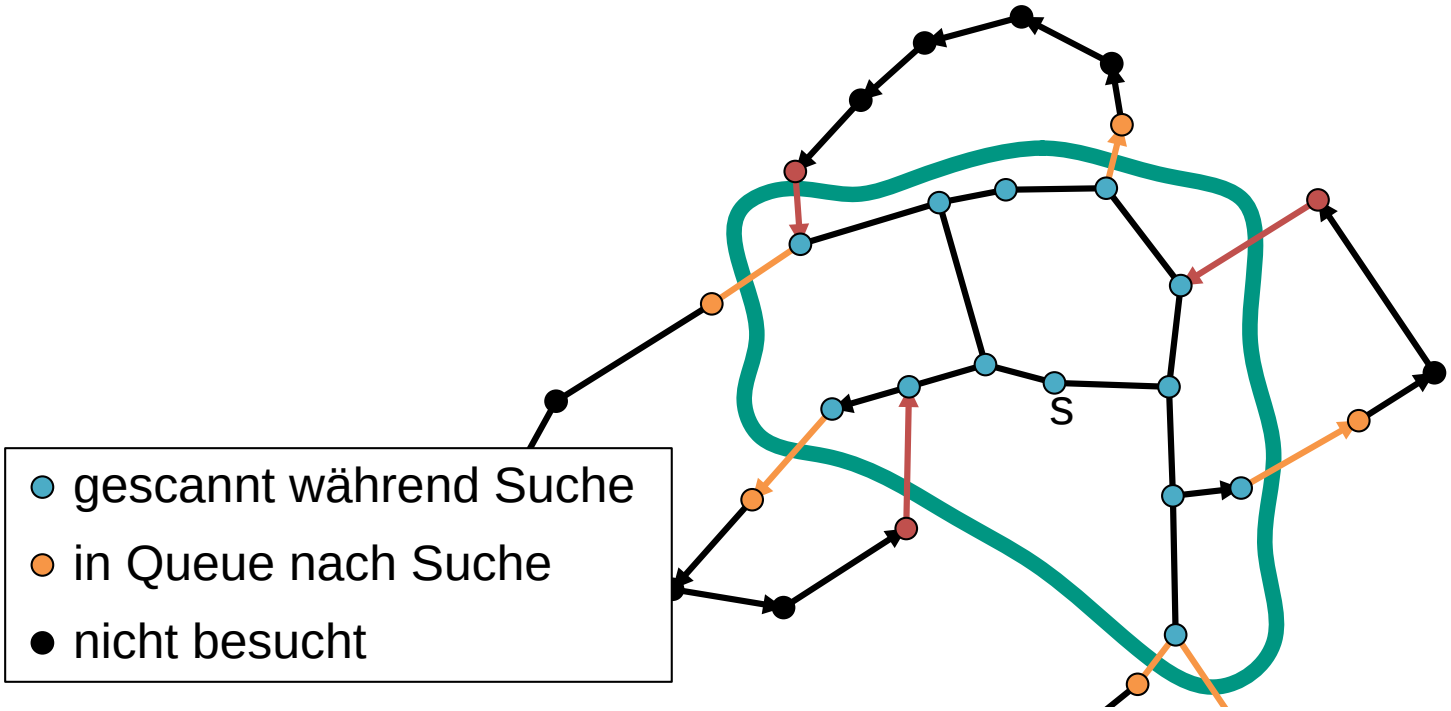
- Stoppkriterium: alle Elemente in Queue haben Distanzlabel größer als $\tau$
- Postprocessing: teste inzidente Kanten der in Queue verbliebenen Knoten



# Isochronen berechnen

**Variante von Dijkstras Algorithmus (isoDijkstra):**

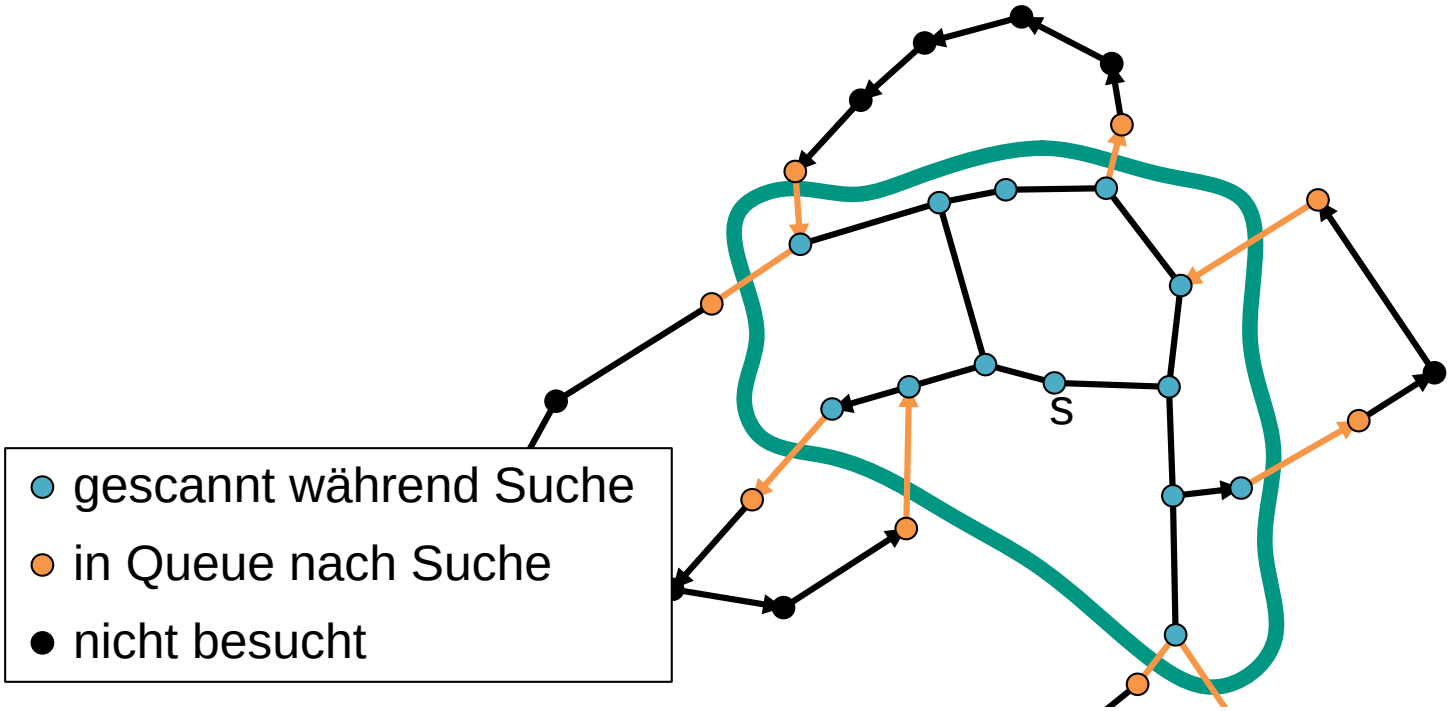
- Stoppkriterium: alle Elemente in Queue haben Distanzlabel größer als $\tau$
- Postprocessing: teste inzidente Kanten der in Queue verbliebenen Knoten
- Problem: übersieht eingehende Isochronenkanten



# Isochronen berechnen

## Variante von Dijkstras Algorithmus (isoDijkstra):

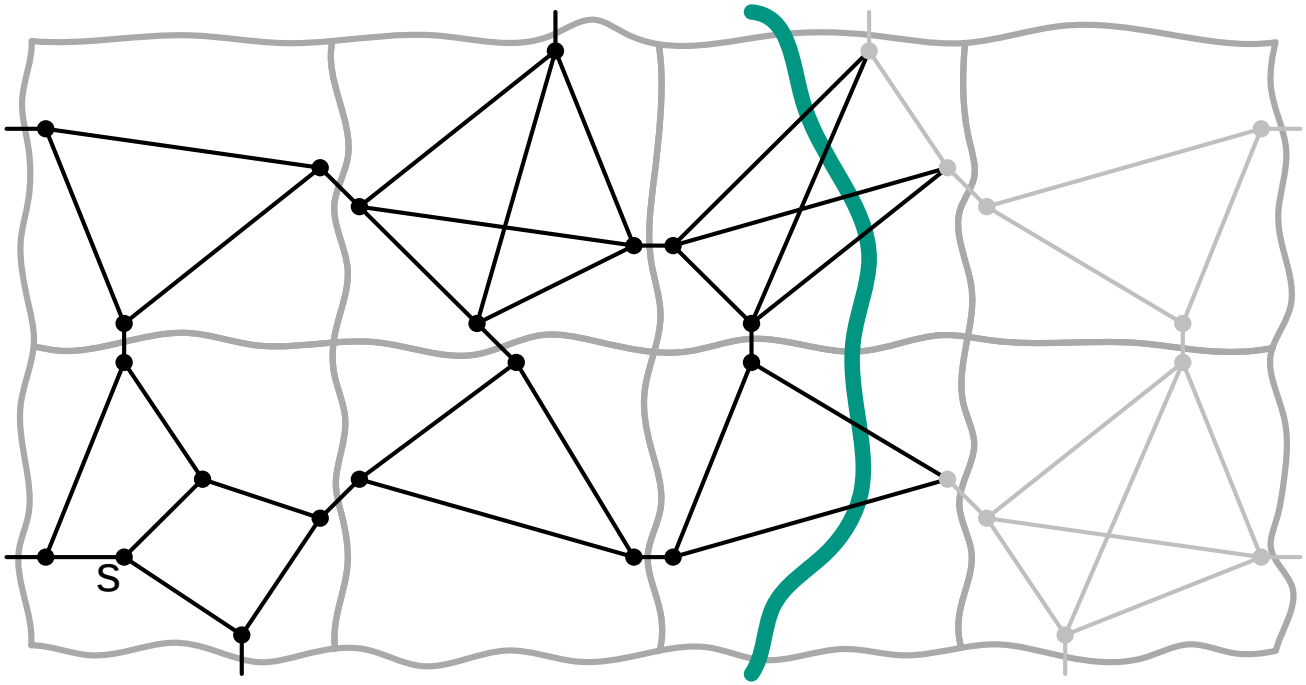
- Stoppkriterium: alle Elemente in Queue haben Distanzlabel größer als $\tau$
- Postprocessing: teste inzidente Kanten der in Queue verbliebenen Knoten
- Nach Scan von Knoten $u$, teste eingehende Nachbarn $v$:
  wenn $d[v] = \infty$, füge $v$ mit $\text{key}(v) = \infty$ in Queue ein



# Isochronen berechnen mit Overlays

**Variante von One-to-many-Anfragen, Zielknoten aber nicht bekannt:**

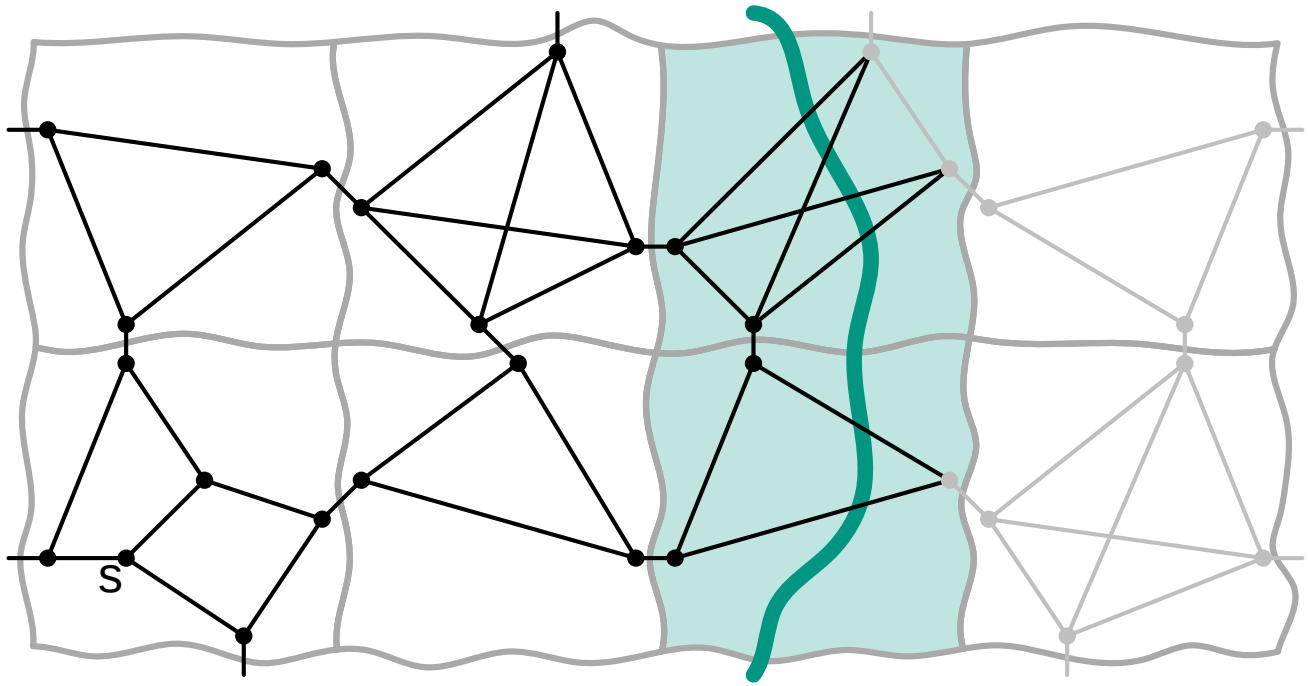
- isoDijkstra auf Startzelle und Overlay



# Isochronen berechnen mit Overlays

**Variante von One-to-many-Anfragen, Zielknoten aber nicht bekannt:**

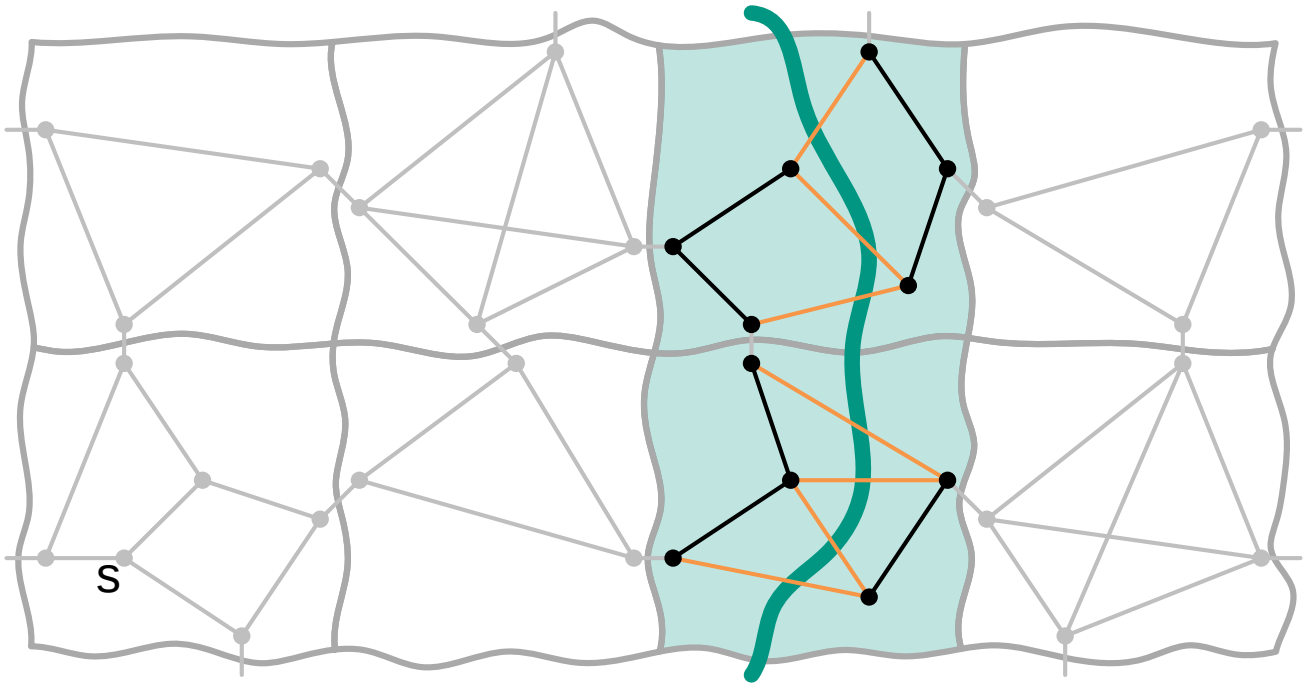
- isoDijkstra auf Startzelle und Overlay
- Markiere dabei besuchte Zellen mit unerreichbaren Knoten



# Isochronen berechnen mit Overlays

**Variante von One-to-many-Anfragen, Zielknoten aber nicht bekannt:**

- isoDijkstra auf Startzelle und Overlay
- Markiere dabei besuchte Zellen mit unerreichbaren Knoten
- Berechne Distanzen und Isochronenkanten in markierten Zellen



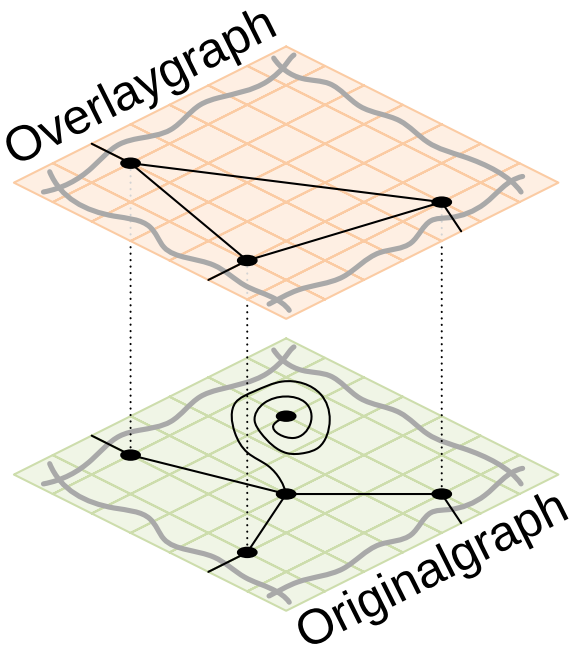
# Isochronen berechnen mit Overlays

**Variante von One-to-many-Anfragen, Zielknoten aber nicht bekannt:**

- isoDijkstra auf Startzelle und Overlay
- Markiere dabei besuchte Zellen mit unerreichbaren Knoten
- Berechne Distanzen und Isochronenkanten in markierten Zellen

**Wann muss Zelle markiert werden?**

- Unerreichbare Knoten in Zelle selbst wenn alle Shortcuts befahrbar (→ Berge)
- Exzentrizität für jeden Randknoten $u$ vorberechnen (Distanz von $u$ zum am weitesten entfernten Knoten in Zelle)
- Teste Exzentrizitäten während Anfragen



# Iso-Anfragen mit CRP und GRASP

**Aufwärtsphase:**

- isoDijkstra auf Startzelle und Overlay
- Zwei Flags $i[\cdot]$ (in range, initial $0$) und $o[\cdot]$ (out of range, initial $1$) pro Zelle
- Nach Scan von Knoten $u$ in Zelle $C$, setze $i[C]$, und lösche $o[C]$ wenn…
  - $\text{dist}(s,u) + \text{ecc}(u) \leq \tau$ und
  - Einige zusätzliche Bedingungen, falls $C$ nicht stark zusammenhängend

**Abwärtsphase:**

- isoDijkstra auf Zellen, für die beide Flags gesetzt sind

isoCRP

- Relax. Abwärtskanten in Zellen, für die beide Flags gesetzt sind

isoGRASP

# Isochronen-Anfragen mit PHAST

**Variante von One-to-many-Anfragen:**

- Endpunkte der Isochronenkanten sind Zielknoten
- Zielknoten aber nicht a priori bekannt

**Probleme:**

- Kein frühes Stoppen am Isochronenrand bei PHAST möglich
- RPHAST nicht anwendbar, da Zielknoten nicht bekannt

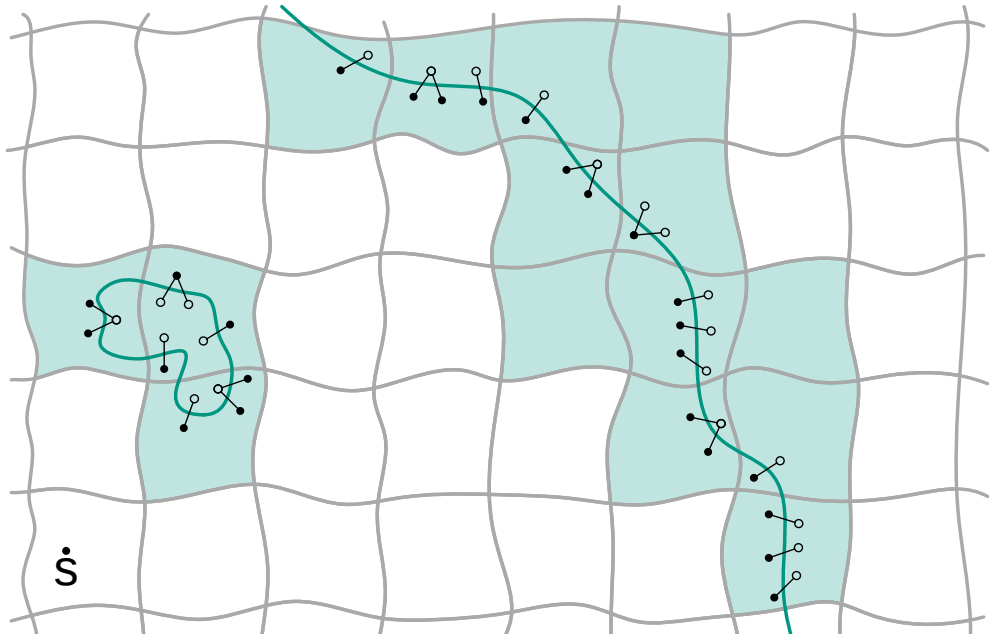
**Wie irrelevante Teile des Graphen überspringen?**

- Partitionen
- Beschränke PHAST auf relevante Zellen

# isoPHAST

## Generischer Ansatz:

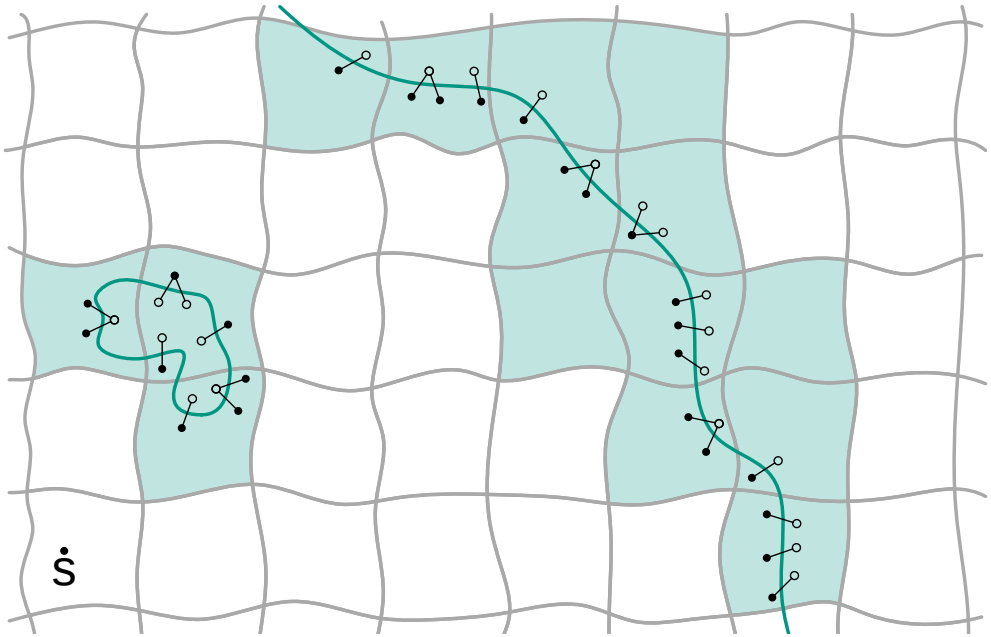
- Vorberechnung: partitioniere und kontrahiere Straßennetz
- Anfragen arbeiten in drei Phasen:
  1. CH-Aufwärtssuche vom Startknoten aus
  2. Bestimme Zellen mit Isochronenkanten (aktive Zellen)
  3. Verarbeite aktive Zelle mit (R)PHAST



# Aktive Zellen bestimmen

## Möglichkeit 1: Suche auf Core-Graphen

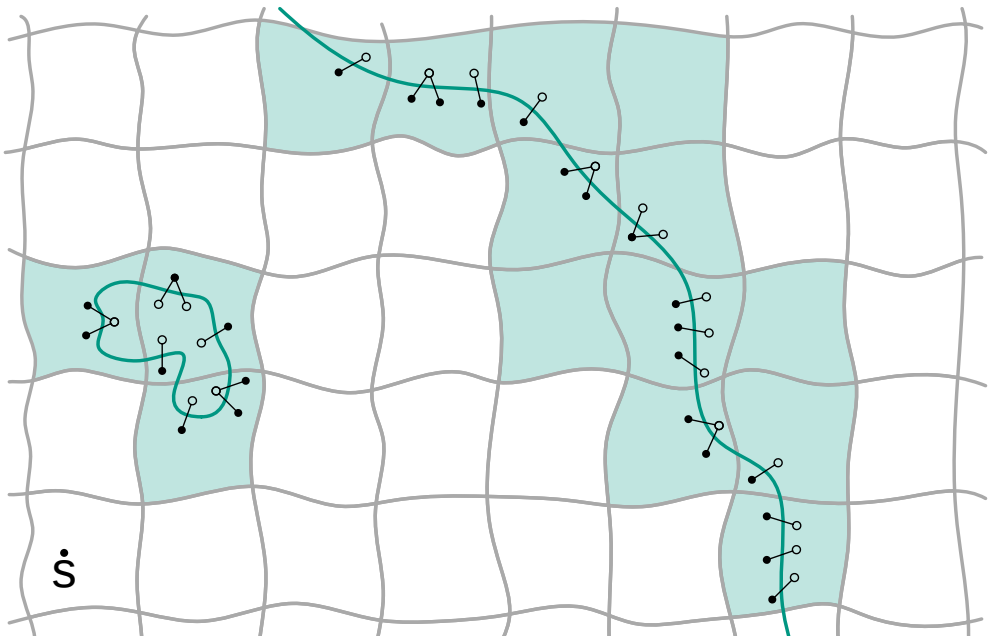
- Kontrahiere innere Knoten, Randknoten induzieren Core-Graphen
- Core-Suche verwaltet zwei Flags $i[\cdot]$ und $o[\cdot]$ pro Zelle (ähnlich isoCRP)
- Markiere Zellen als aktiv, für die beide Flags gesetzt sind
- Varianten: Core-Dijkstra (isoPHAST-CD), Core-PHAST (isoPHAST-CP)



# Aktive Zellen bestimmen

## Möglichkeit 2: Distanzschranken zwischen Zellen vorberechnen

- Untere/obere Schranke $\mathrm{lb}(C_1, C_2)$ / $\mathrm{ub}(C_1, C_2)$ für $\mathrm{dist}(u, v)$ , $u \in C_1, v \in C_2$
- Während Anfragen: markiere $C$ als aktiv, wenn $\mathrm{lb}(C_s, C) \leq \tau < \mathrm{ub}(C_s, C)$
- Variante: Distance-Table (isoPHAST-DT)

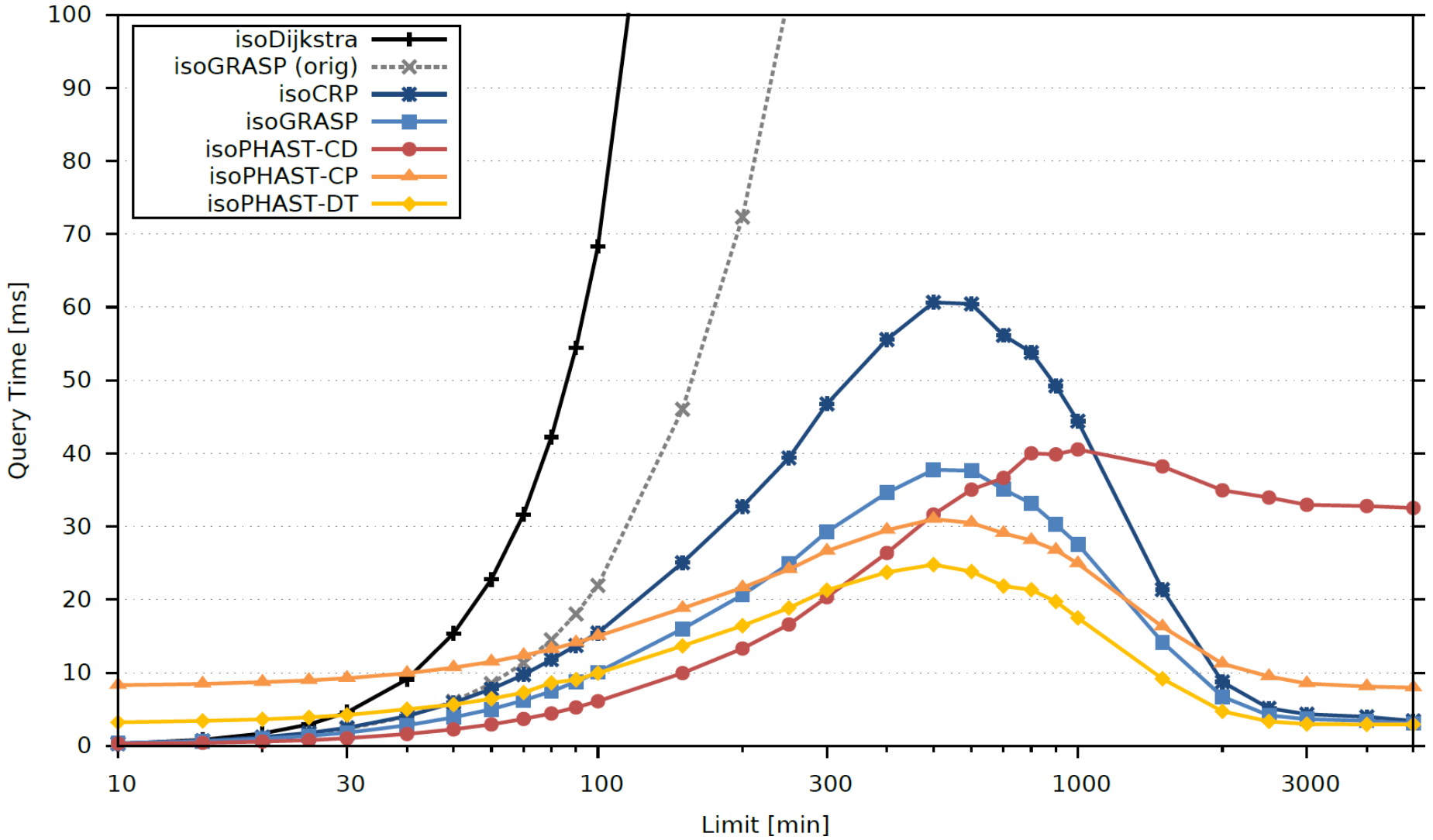


# Performance der Algorithmen

| | par. Customization | | | seq. Anfragezeit [ms] | | par. Anfragezeit [ms] | |
|---|---|---|---|---|---|---|---|
| Algorithmus | Zeit [s] | Platz [MiB] | | $\tau = 100\text{min}$ | $\tau = 500\text{min}$ | $\tau = 100\text{min}$ | $\tau = 500\text{min}$ |
| isoDijkstra | – | 646 | | 68.32 | 1184.06 | – | – |
| isoCRP | **1.70** | 900 | (138) | 15.44 | 60.67 | 2.73 | 7.86 |
| isoGRASP | 2.50 | 1 856 | (1 094) | 10.06 | 37.77 | 2.35 | 5.93 |
| isoPHAST-CD | 38.07 | 769 | | **6.09** | 31.63 | **1.61** | 8.22 |
| isoPHAST-CP | 1 432.39 | **766** | | 15.02 | 31.00 | 4.47 | 7.86 |
| isoPHAST-DT | 865.50 | 1 066 | | 9.96 | **24.80** | 1.74 | **3.80** |

**Maschine** 2×8 Intel Xeon E5-2670, 2.60 GHz, 64 GiB DDR3-1600 RAM
**Eingabe** DIMACS Europe (18M Knoten, 42M Kanten)
**Methodik** 1 000 Zufallsanfragen pro Zeitlimit $\tau$
**Partition** verschiedene, siehe Referenzen

# Seq. Anfragezeiten für div. Zeitlimits



# Fazit Isochronen

**Alle Beschleunigungstechniken schnell genug für Praxis:**

- isoCRP und isoGRASP bieten schnellere Customization
- isoPHAST bietet schnellere Anfragen

**Richtige Wahl hängt von Anwendung ab:**

- Speichern mehrerer benutzerspezifischer Metriken → isoCRP
- Schnellere Anfragen und Echtzeit-Metrikupdates → isoGRASP
- Schnellste Anfragen für niedrige Zeitlimits → isoPHAST-CD
- Bestes Skalierungsverhalten → isoPHAST-DT

# Zusammenfassung

- one-to-all shortest paths
- one-to-many shortest paths
- many-to-many
- POI Anfragen
- bester Via Knoten Anfragen
- Location Services in SQL
- Isochronen

# Ende

## Literatur:


- Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, Renato F. Werneck:
  **PHAST: Hardware-accelerated shortest path trees** In: *Journal of Parallel and Distributed Computing*, pages 940–952, 2013.
- Alexandros Efentakis, Dieter Pfoser:
  **GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem**
  In: *Proceedings of the 22nd European Symposium on Algorithms (ESA'14)*, pages 358-370, 2014.
- Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner:
  **Computing Many-to-Many Shortest Paths Using Highway Hierarchies**
  In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36-45, 2007.

# Ende

**Literatur:**


- Daniel Delling, Andrew V. Goldberg, Renato F. Werneck:
  **Faster Batched Shortest Paths in Road Networks**
  In: *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, pages 52-63, 2011.
- Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, Renato F. Werneck:
  **HLDB: Location-Based Services in Databases**
  In: *Proceedings of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 339-348, 2012.
- Moritz Baum, Valentin Buchhold, Julian Dibbelt, Dorothea Wagner:
  **Fast Exact Computation of Isochrones in Road Networks**
  In: *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, pages 17-32, 2016.