

Algorithmen für Routenplanung

9. Vorlesung, Sommersemester 2021

Valentin Buchhold | 10. Mai 2021



Punkt-zu-Punkt

- zwei Punkte → kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken

Punkt-zu-Punkt

- zwei Punkte → kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken

All-Pairs

- Distanzen zwischen allen Knotenpaaren
- wird für Vorbereitung benutzt

Punkt-zu-Punkt

- zwei Punkte → kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken

All-Pairs

- Distanzen zwischen allen Knotenpaaren
- wird für Vorbereitung benutzt

One-to-All

- ein Knoten → Distanzen zu allen Knoten
- wird für Vorbereitung benutzt

Punkt-zu-Punkt

- zwei Punkte → kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken

All-Pairs

- Distanzen zwischen allen Knotenpaaren
- wird für Vorberechnung benutzt

One-to-Many

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

One-to-All

- ein Knoten → Distanzen zu allen Knoten
- wird für Vorberechnung benutzt

Punkt-zu-Punkt

- zwei Punkte → kürzester Weg
- wird für Routenplanung benutzt
- Beschleunigungstechniken

All-Pairs

- Distanzen zwischen allen Knotenpaaren
- wird für Vorbereitung benutzt

One-to-Many

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

One-to-All

- ein Knoten → Distanzen zu allen Knoten
- wird für Vorbereitung benutzt

Many-to-Many

- zwei Mengen → Distanztabelle
- wichtig für Vehicle Routing

Kürzeste Wege von einem Startknoten aus (**single-source**):

- Dijkstras Algorithmus [Dij59, Dan62]
- Bellman-Ford-Algorithmus [Bel58, FF62]

Kürzeste Wege zwischen allen Knotenpaaren (**all-pairs**):

- n -mal Dijkstra oder Bellman-Ford
- Floyd-Warshall-Algorithmus [Flo62, War62]
- Johnsons Algorithmus [Joh77]

- Sei $P_{st}^k = (s, v_1, \dots, v_j, t)$ ein kürzester s - t -Pfad, für den alle **Zwischenknoten** v_i in der Menge $\{1, \dots, k\}$ liegen
- Beobachtung:
 - Wenn $k \notin P_{st}^k$, dann ist P_{st}^k auch ein P_{st}^{k-1}
 - Wenn $k \in P_{st}^k$, dann ist P_{st}^k die Konkatenation von P_{sk}^{k-1} und P_{kt}^{k-1}
- Sei $d_k(s, t)$ die Länge eines P_{st}^k , dann gilt:

- Sei $P_{st}^k = (s, v_1, \dots, v_j, t)$ ein kürzester s - t -Pfad, für den alle **Zwischenknoten** v_i in der Menge $\{1, \dots, k\}$ liegen
- Beobachtung:
 - Wenn $k \notin P_{st}^k$, dann ist P_{st}^k auch ein P_{st}^{k-1}
 - Wenn $k \in P_{st}^k$, dann ist P_{st}^k die Konkatination von P_{sk}^{k-1} und P_{kt}^{k-1}
- Sei $d_k(s, t)$ die Länge eines P_{st}^k , dann gilt:

$$d_k(s, t) = \begin{cases} \ell(s, t) & \text{wenn } k = 0 \\ \min\{d_{k-1}(s, t), d_{k-1}(s, k) + d_{k-1}(k, t)\} & \text{wenn } k \geq 1 \end{cases}$$

FloydWarshall($G = (V, E), \ell : E \rightarrow \mathbb{R}$)

```
1 set  $d[s][t] \leftarrow \infty$  for all  $s, t \in V$ 
2 set  $parent[s][t] \leftarrow \perp$  for all  $s, t \in V$ 
3 foreach vertex  $u \in V$  do  $d[u][u] \leftarrow 0$ 
4 foreach edge  $e = (u, v) \in E$  do  $d[u][v] \leftarrow \ell(e)$     $parent[u][v] \leftarrow u$ 
5 for  $k \leftarrow 1$  to  $n$  do
6   foreach vertex  $s \in V$  do
7     foreach vertex  $t \in V$  do
8       if  $d[s][k] + d[k][t] < d[s][t]$  then
9          $d[s][t] \leftarrow d[s][k] + d[k][t]$ 
10         $parent[s][t] \leftarrow parent[k][t]$ 
```

FloydWarshall($G = (V, E), \ell : E \rightarrow \mathbb{R}$)

```
1 set  $d[s][t] \leftarrow \infty$  for all  $s, t \in V$ 
2 set  $parent[s][t] \leftarrow \perp$  for all  $s, t \in V$ 
3 foreach vertex  $u \in V$  do  $d[u][u] \leftarrow 0$ 
4 foreach edge  $e = (u, v) \in E$  do  $d[u][v] \leftarrow \ell(e)$     $parent[u][v] \leftarrow u$ 
5 for  $k \leftarrow 1$  to  $n$  do
6   foreach vertex  $s \in V$  do
7     foreach vertex  $t \in V$  do
8       if  $d[s][k] + d[k][t] < d[s][t]$  then
9          $d[s][t] \leftarrow d[s][k] + d[k][t]$ 
10         $parent[s][t] \leftarrow parent[k][t]$ 
```

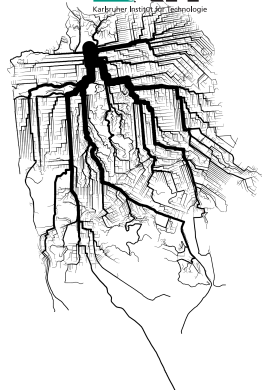
Zeit: $O(n^3)$ (aber kleiner konstanter Faktor) Platz: $O(n^2)$

PHAST

Kürzeste-Wege Bäume

Anfrage:

- gegeben ein nichtnegativ gewichteter gerichteter Graph und Knoten s
- berechne Distanzen von s zu *allen* anderen Knoten



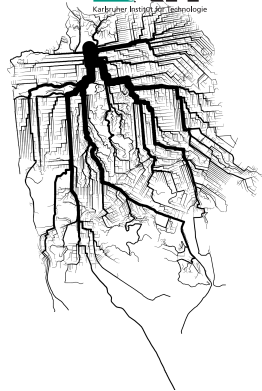
Kürzeste-Wege Bäume

Anfrage:

- gegeben ein nichtnegativ gewichteter gerichteter Graph und Knoten s
- berechne Distanzen von s zu *allen* anderen Knoten

Lösung:

- Dijkstra [Dij59]



Kürzeste-Wege Bäume

Anfrage:

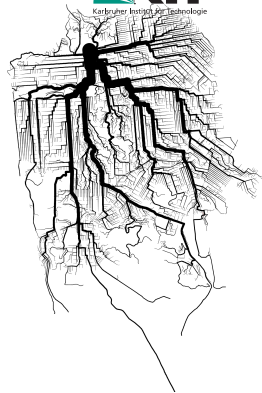
- gegeben ein nichtnegativ gewichteter gerichteter Graph und Knoten s
- berechne Distanzen von s zu *allen* anderen Knoten

Lösung:

- Dijkstra [Dij59]

Fakten:

- $O(m + n \log n)$ mit Fibonacci Heaps [FT87]
- **linear** (mit kleinen Konstanten) in Praxis [Gol01]
- Ausnutzung von moderner Hardware schwierig

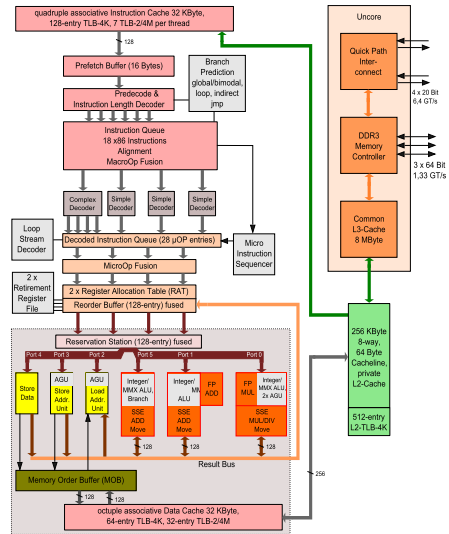


Moderne CPU Architektur

Einige Fakten:

- viele Kerne
- mehr Kerne als Speichercontroller
- Hyperthreading
- Multi-Sockel-Systeme
- steile Speicherhierarchie
- Cache coherency

Intel Nehalem microarchitecture



GT/s: gigatransfers per second

Moderne CPU Architektur

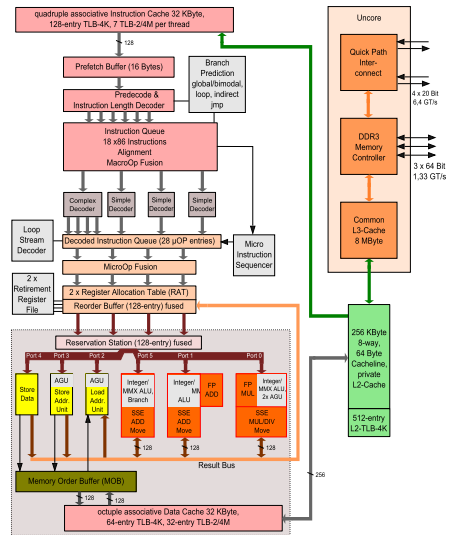
Einige Fakten:

- viele Kerne
- mehr Kerne als Speichercontroller
- Hyperthreading
- Multi-Sockel-Systeme
- steile Speicherhierarchie
- Cache coherency

Haupt Herausforderungen:

- Parallelisierung
- Speicherzugriff

Intel Nehalem microarchitecture



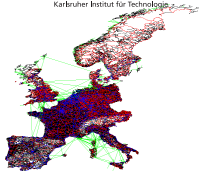
GT/s: gigatransfers per second

Ausnutzen von moderner Hardware

Datenlokalität

- Eingabe: West Europa
- 18M Knoten, 23M Straßen

Dijkstra: ≈ 3.0 s \Rightarrow nicht real-time



Core-i7 workstation (2.66 GHz)

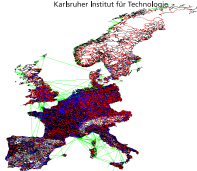
Ausnutzen von moderner Hardware

Datenlokalität

- Eingabe: West Europa
- 18M Knoten, 23M Straßen

Dijkstra: ≈ 3.0 s \Rightarrow nicht real-time

$n + m$ clock cycles: ≈ 15 ms \Rightarrow viel schneller



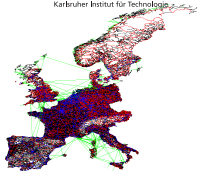
Core-i7 workstation (2.66 GHz)

Ausnutzen von moderner Hardware

Datenlokalität

- Eingabe: West Europa
- 18M Knoten, 23M Straßen
 - Dijkstra: ≈ 3.0 s \Rightarrow nicht real-time
 - $n + m$ clock cycles: ≈ 15 ms \Rightarrow viel schneller
 - BFS: ≈ 2.0 s
- Verlangsamung kommt nicht durch Priorityqueue allein

Core-i7 workstation (2.66 GHz)

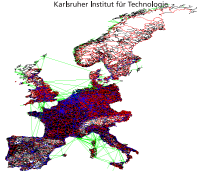


Ausnutzen von moderner Hardware

Datenlokalität

- Eingabe: West Europa
- 18M Knoten, 23M Straßen
 - Dijkstra: ≈ 3.0 s \Rightarrow nicht real-time
 - $n + m$ clock cycles: ≈ 15 ms \Rightarrow viel schneller
 - BFS: ≈ 2.0 s
- Verlangsamung kommt nicht durch Priorityqueue allein

Core-i7 workstation (2.66 GHz)



Parallelisierung:

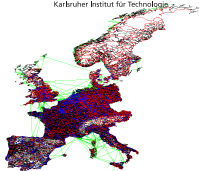
- Spekulation
- Δ -stepping [MS03]
- mehr Operationen als Dijkstra
- keine große Beschleunigung auf dünnen Graphen

Ausnutzen von moderner Hardware

Datenlokalität

- Eingabe: West Europa
- 18M Knoten, 23M Straßen
 - Dijkstra: ≈ 3.0 s \Rightarrow nicht real-time
 - $n + m$ clock cycles: ≈ 15 ms \Rightarrow viel schneller
 - BFS: ≈ 2.0 s
- Verlangsamung kommt nicht durch Priorityqueue allein

Core-i7 workstation (2.66 GHz)



Parallelisierung:

- Spekulation
- Δ -stepping [MS03]
- mehr Operationen als Dijkstra
- keine große Beschleunigung auf dünnen Graphen
- Berechnen von mehreren Bäumen ist einfach

Ansatz 1

Idee:

- Umordnen der Knoten im Graphen

Idee:

- Umordnen der Knoten im Graphen

algorithm	details	time per tree [ms]		
		random	input	DFS
Dijkstra	binary heap	11159	5859	5180
	Dial	7767	3538	2908
	smart queue	7991	3556	2826
BFS	—	6060	2445	2068

(**Achtung:** one-to-all & älterer Rechner)

Idee:

- Umordnen der Knoten im Graphen

algorithm	details	time per tree [ms]		
		random	input	DFS
Dijkstra	binary heap	11159	5859	5180
	Dial	7767	3538	2908
	smart queue	7991	3556	2826
BFS	—	6060	2445	2068

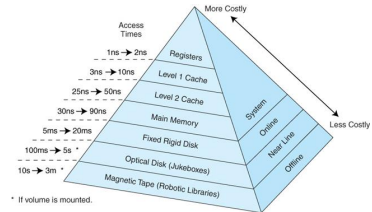
(**Achtung:** one-to-all & älterer Rechner)

⇒ keine große Beschleunigung

Ansatz 2

Dijkstras Algorithmus:

- moderne Hardware nicht ausgenutzt
- Hauptprobleme:
 - Datenlokalität
 - Parallelisierung



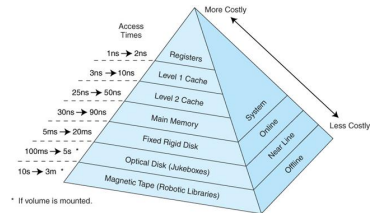
Ansatz 2

Dijkstras Algorithmus:

- moderne Hardware nicht ausgenutzt
- Hauptprobleme:
 - Datenlokalität
 - Parallelisierung

Fragen:

- hilft Vorbereitung?
- wie?
- Ansatzpunkt?



Ansatz 2

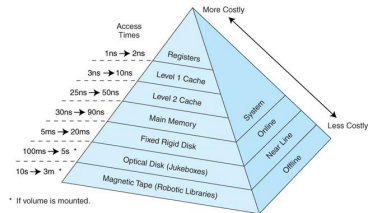
Dijkstras Algorithmus:

- moderne Hardware nicht ausgenutzt
- Hauptprobleme:
 - Datenlokalität
 - Parallelisierung

Fragen:

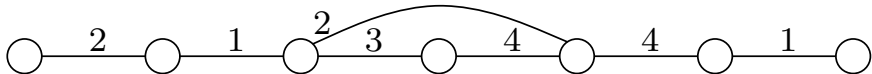
- hilft Vorbereitung?
- wie?
- Ansatzpunkt?

PHAST: Hardware-Accelerated Shortest path Trees



Contraction Hierarchies

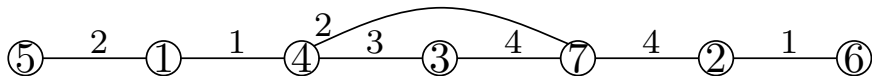
Vorbereitung:



Contraction Hierarchies

Vorbereitung:

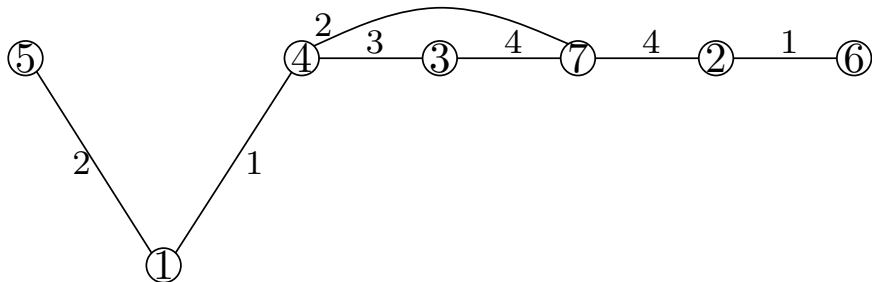
- ordne Knoten nach Wichtigkeit



Contraction Hierarchies

Vorbereitung:

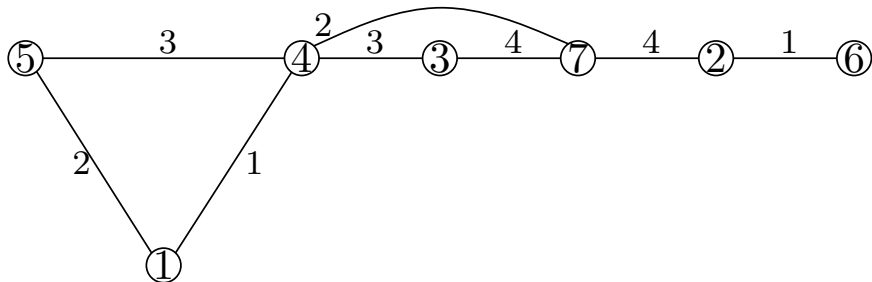
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

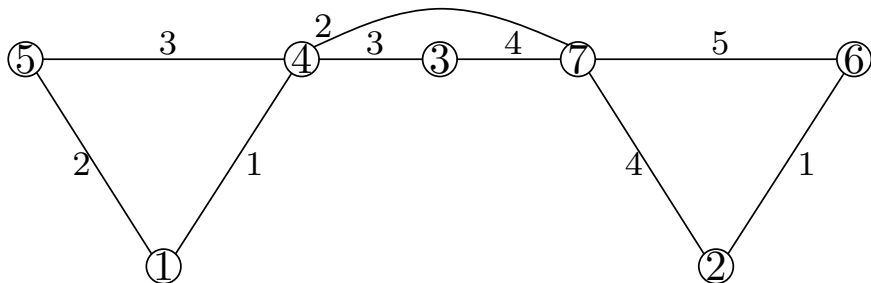
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

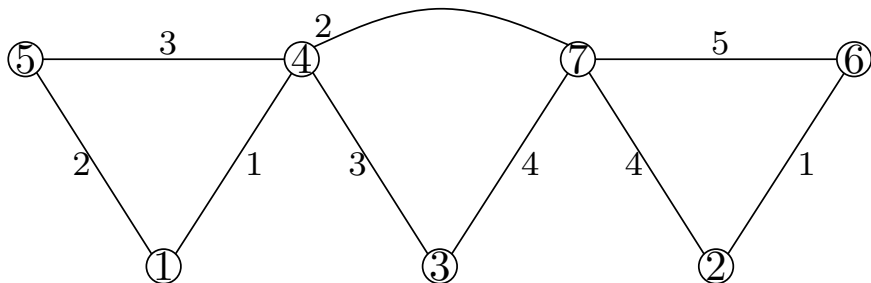
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

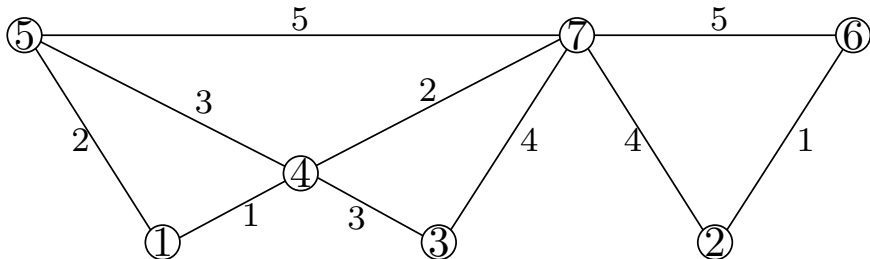
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

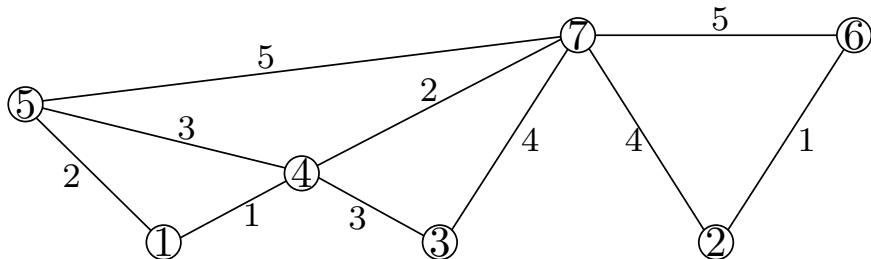
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

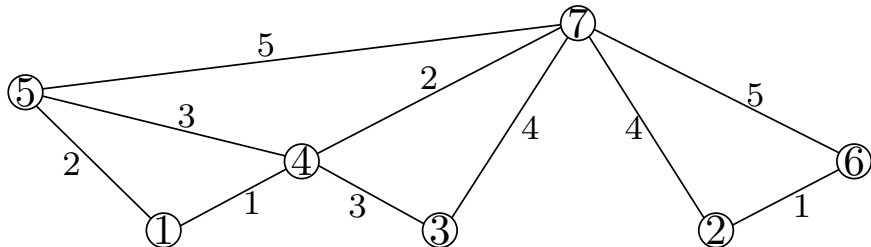
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

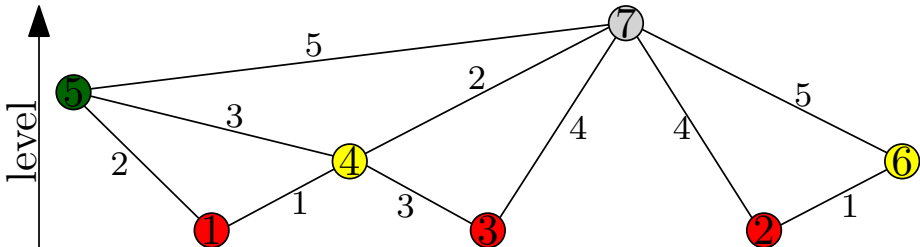
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu



Contraction Hierarchies

Vorbereitung:

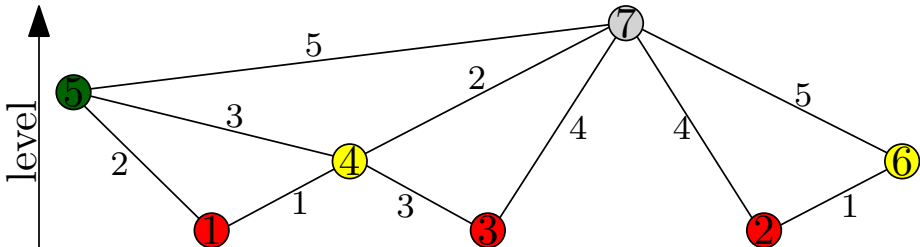
- ordne Knoten nach Wichtigkeit
- bearbeite in der Reihenfolge
- füge Shortcuts hinzu
- Levelzuordnung (ca. 150 in Straßennetzwerken)



Contraction Hierarchies

Punkt-zu-Punkt-Anfragen

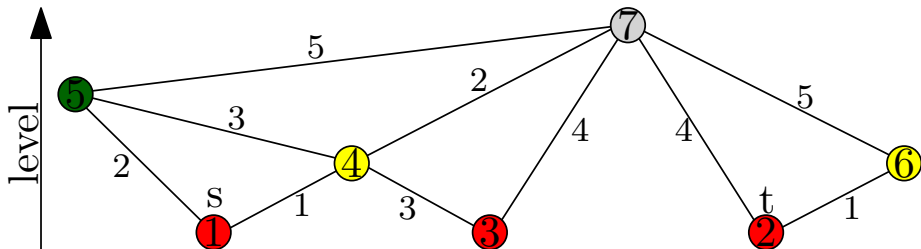
- modifizierter **bidirektionaler** Dijkstra
- folgt nur Kanten zu wichtigeren Knoten
- besucht nur 500 Knoten



Contraction Hierarchies

Punkt-zu-Punkt-Anfragen

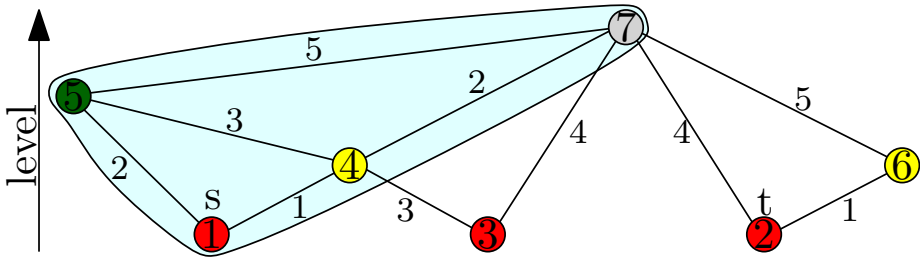
- modifizierter **bidirektionaler** Dijkstra
- folgt nur Kanten zu wichtigeren Knoten
- besucht nur 500 Knoten



Contraction Hierarchies

Punkt-zu-Punkt-Anfragen

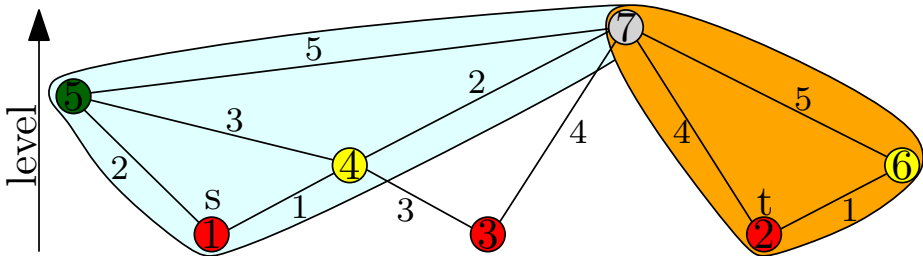
- modifizierter **bidirektionaler** Dijkstra
- folgt nur Kanten zu wichtigeren Knoten
- besucht nur 500 Knoten



Contraction Hierarchies

Punkt-zu-Punkt-Anfragen

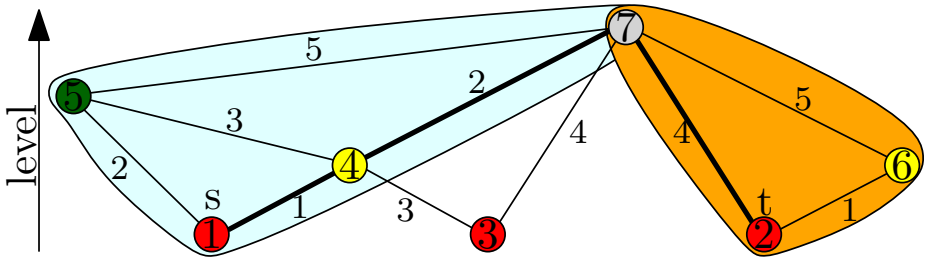
- modifizierter **bidirektionaler** Dijkstra
- folgt nur Kanten zu wichtigeren Knoten
- besucht nur 500 Knoten



Contraction Hierarchies

Punkt-zu-Punkt-Anfragen

- modifizierter **bidirektionaler** Dijkstra
- folgt nur Kanten zu wichtigeren Knoten
- besucht nur 500 Knoten



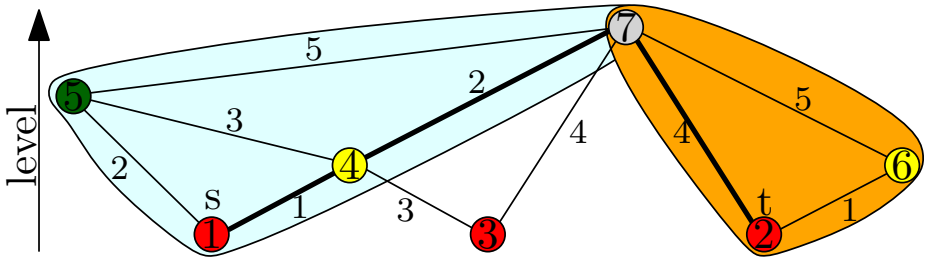
Contraction Hierarchies

Punkt-zu-Punkt-Anfragen

- modifizierter **bidirektionaler** Dijkstra
- folgt nur Kanten zu wichtigeren Knoten
- besucht nur 500 Knoten

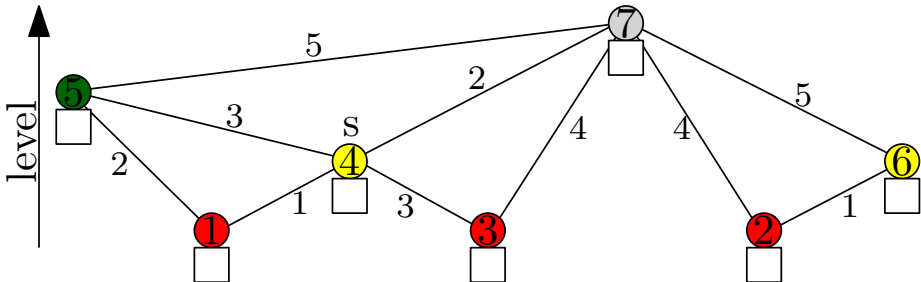
Korrektheit:

- es gibt einen wichtigsten Knoten auf dem Pfad
- dieser wird von Vorwärts- und Rückwärtssuche gescannt



Neuer Anfragealgorithmus

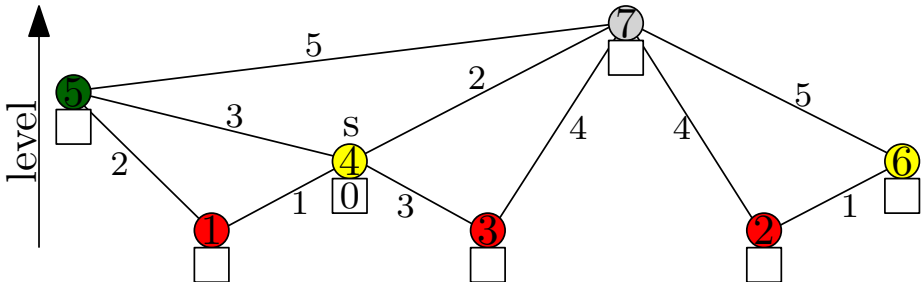
One-to-all-Suche von s :



Neuer Anfragealgorithmus

One-to-all-Suche von s :

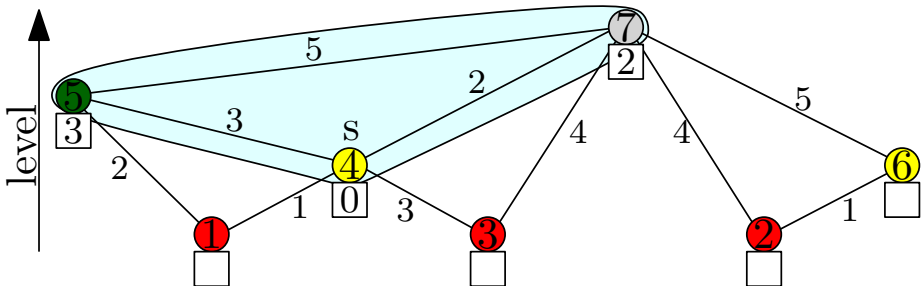
- vorwärts CH-Suche von s (≈ 0.05 ms)



Neuer Anfragealgorithmus

One-to-all-Suche von s :

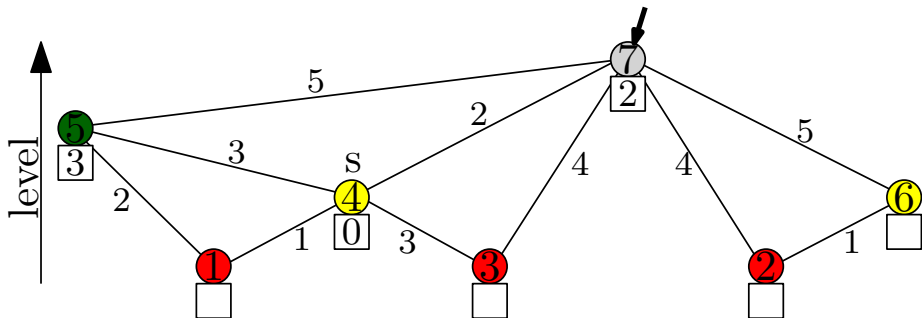
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten



Neuer Anfragealgorithmus

One-to-all-Suche von s :

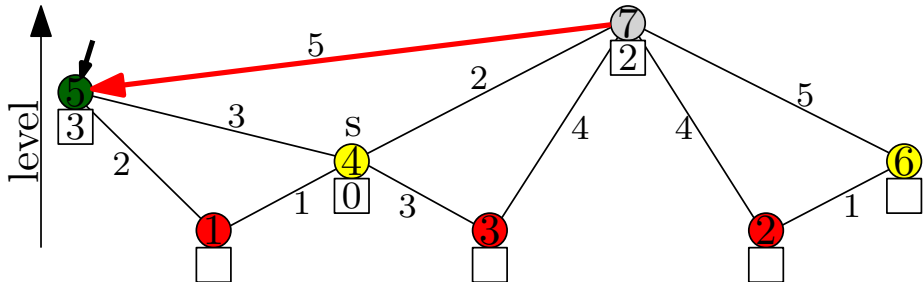
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

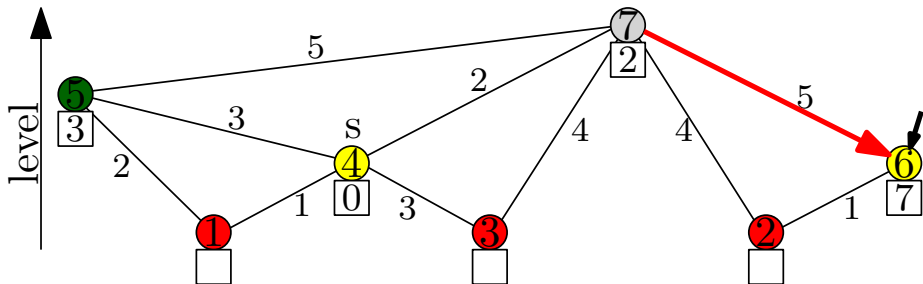
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

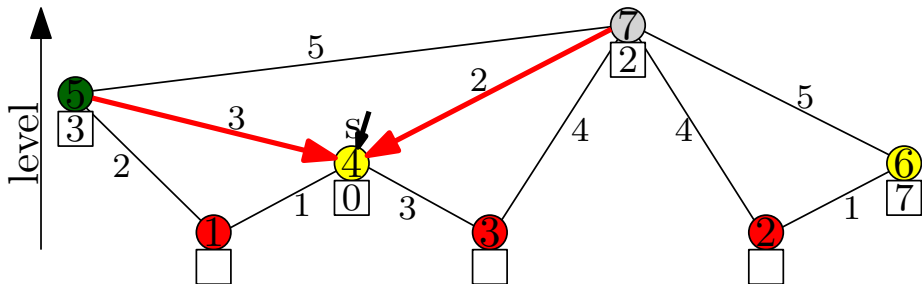
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

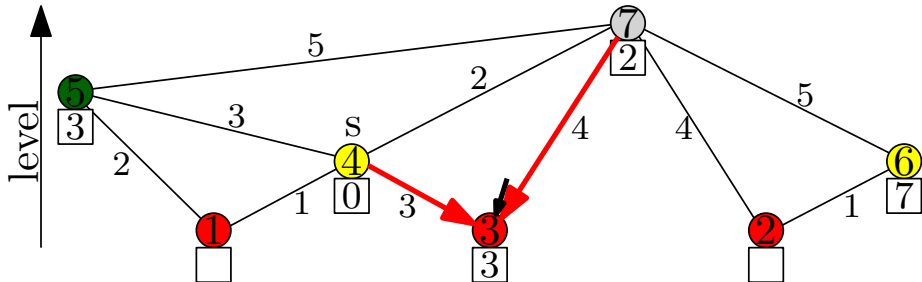
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

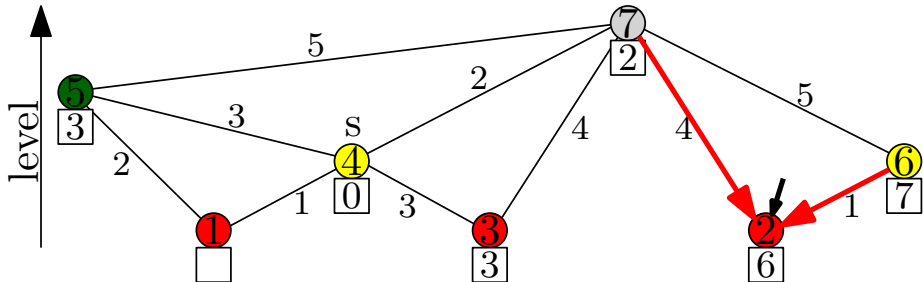
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

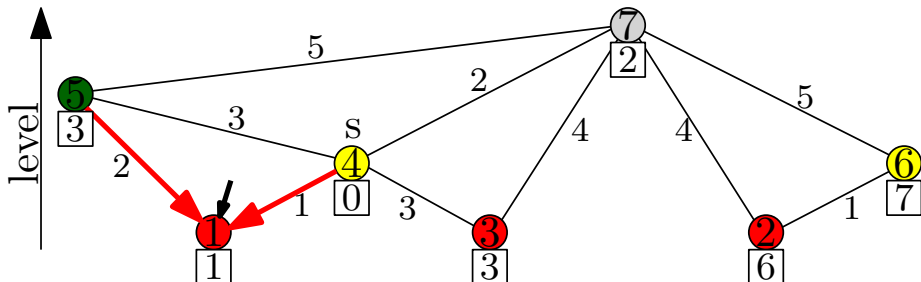
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

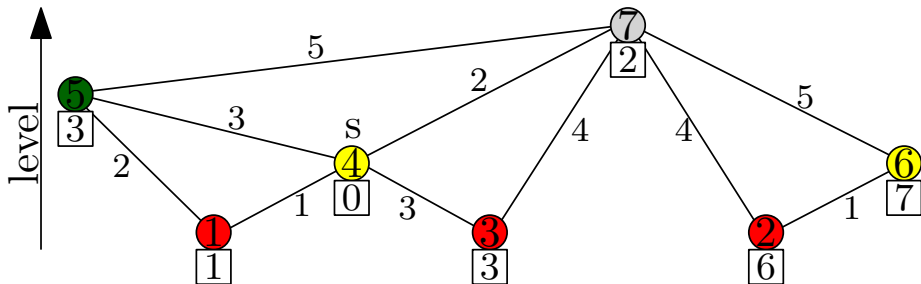
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Neuer Anfragealgorithmus

One-to-all-Suche von s :

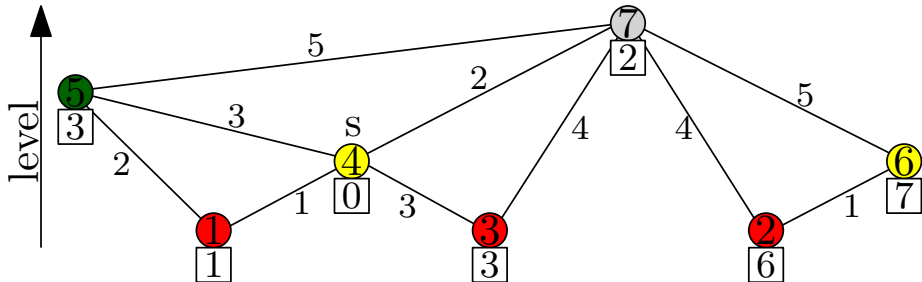
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$
- **top-down** Bearbeitung ohne Priority Queue (ca.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



Neuer Anfragealgorithmus

One-to-all-Suche von s :

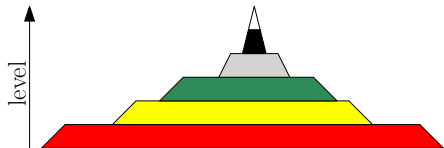
- vorwärts CH-Suche von s (≈ 0.05 ms)
- setze Distanzen d für alle erreichten Knoten
- bearbeite alle Knoten u in **inverser** Levelordnung:
 - checke **eingehende** Kanten (v, u) mit $lev(v) > lev(u)$
 - setze $d(u) = \min\{d(u), d(v) + w(v, u)\}$
- **top-down** Bearbeitung ohne Priority Queue (ca.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



Analyse

Beobachtung:

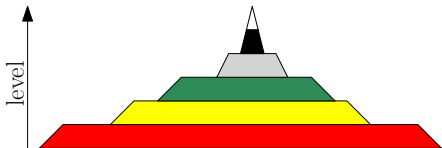
- top-down Prozess ist der Flaschenhals



Analyse

Beobachtung:

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

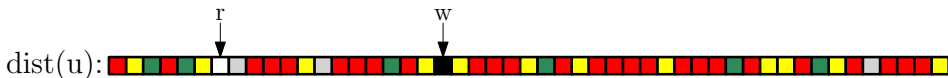
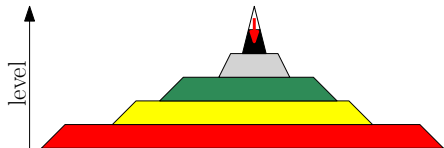


dist(u): 

Analyse

Beobachtung:

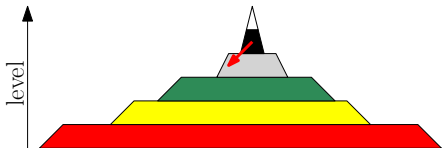
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**



Analyse

Beobachtung:

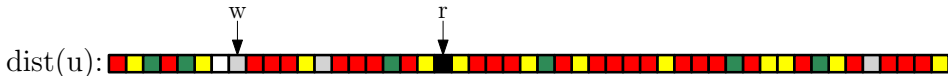
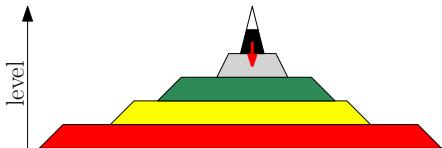
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**



Analyse

Beobachtung:

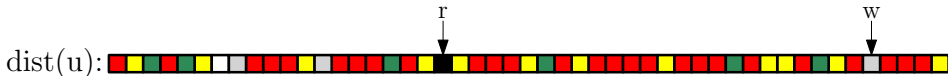
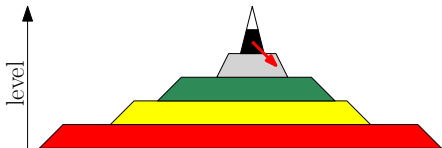
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**



Analyse

Beobachtung:

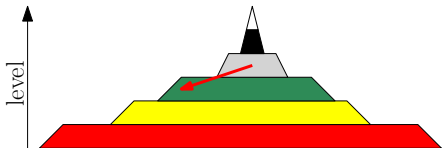
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**



Analyse

Beobachtung:

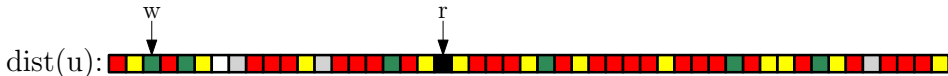
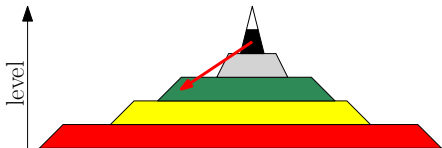
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**



Analyse

Beobachtung:

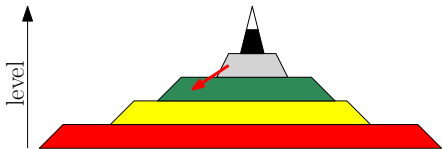
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**



Analyse

Beobachtung:

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



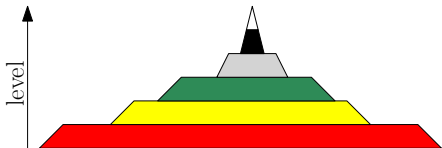
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



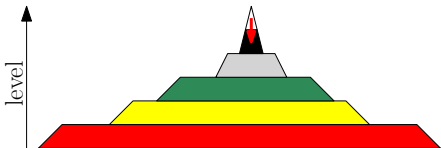
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



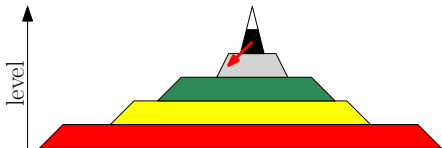
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



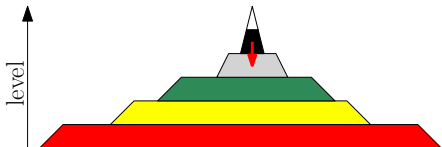
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



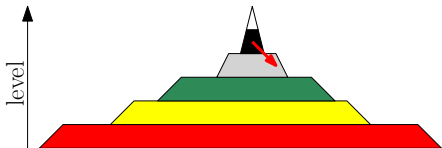
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



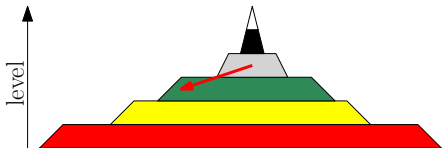
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



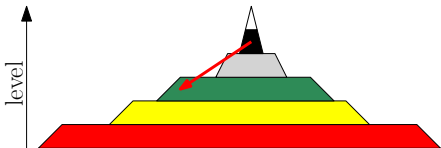
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**



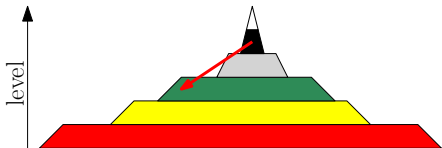
Analyse

Beobachtung:

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

Idee:

- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten
und schreiben der Distanzen
wird zu einem **sequenziellen Sweep**
- ⇒ 172 ms pro Baum



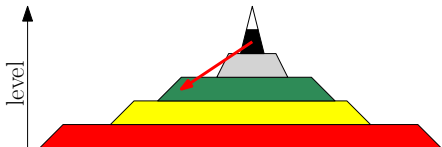
Analyse

Beobachtung:

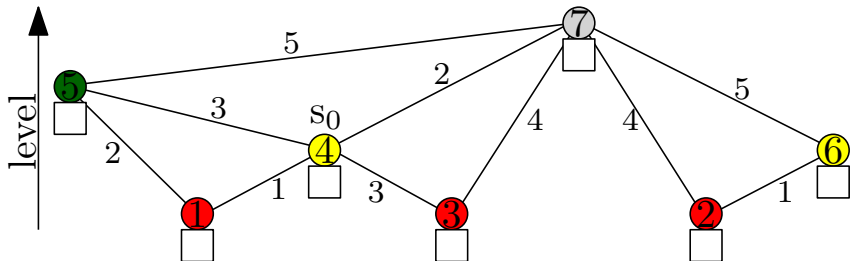
- top-down Prozess ist der Flaschenhals
- Zugriff auf die Daten ist immer noch **ineffizient**
- Zugriffsmuster sind **unabhängig** von s

Idee:

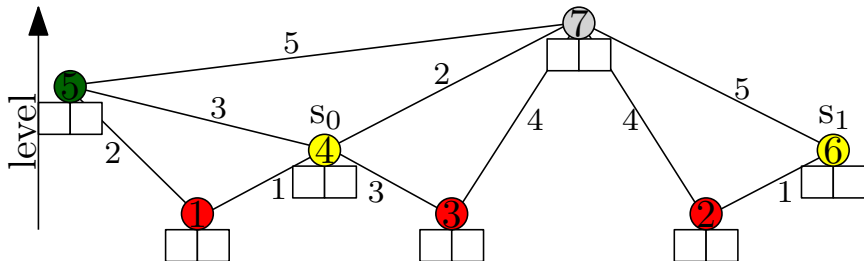
- speichere G_{\uparrow} und G_{\downarrow} separat
 - **Umordnung** der Knoten, Kanten, und Distanzlabel nach Level
- ⇒ lesen der Kanten und schreiben der Distanzen wird zu einem **sequenziellen Sweep**
- ⇒ 172 ms pro Baum
- aber lesen der Distanzen immer noch **ineffizient**



Szenario: Multiple Startknoten

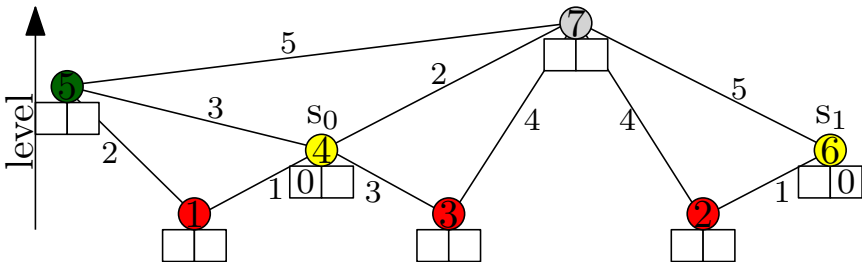


Szenario: Multiple Startknoten



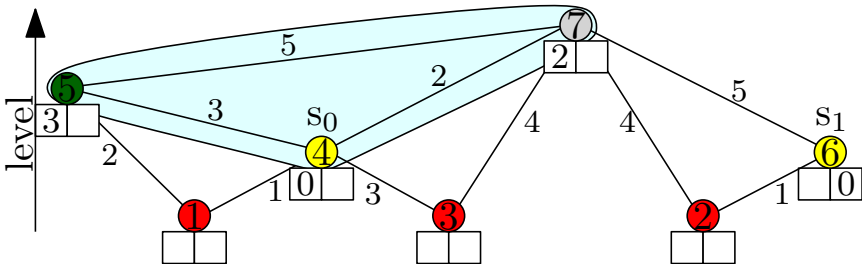
Szenario: Multiple Startknoten

- k Vorwärtssuchen



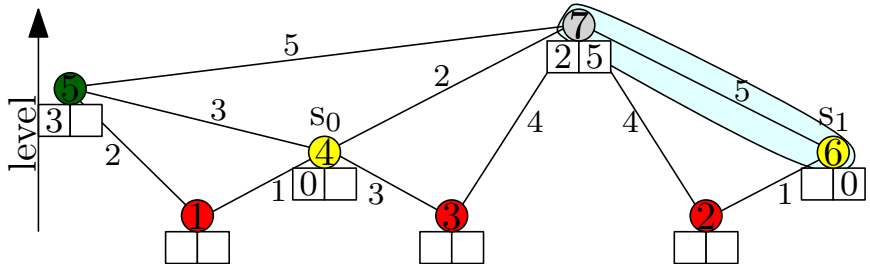
Szenario: Multiple Startknoten

- k Vorwärtssuchen



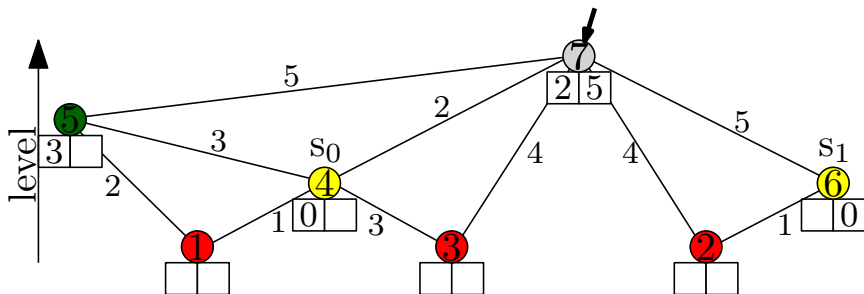
Szenario: Multiple Startknoten

- k Vorwärtssuchen



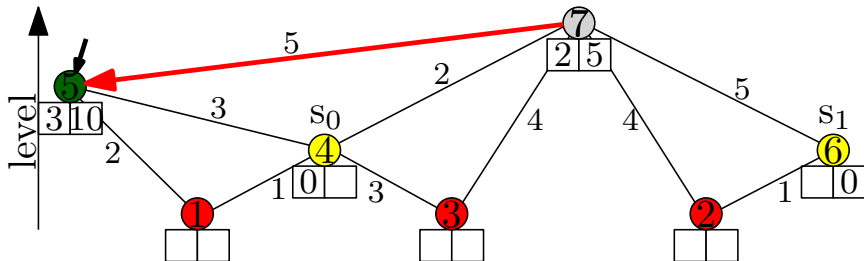
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



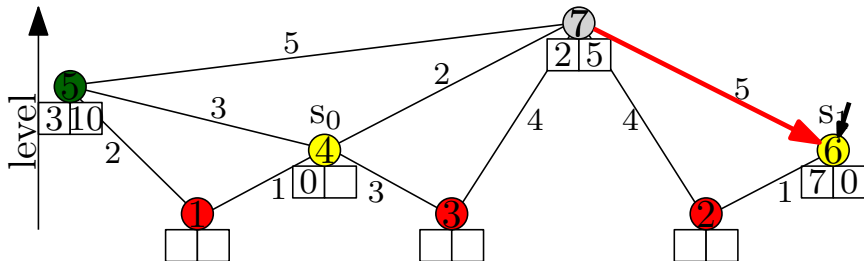
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



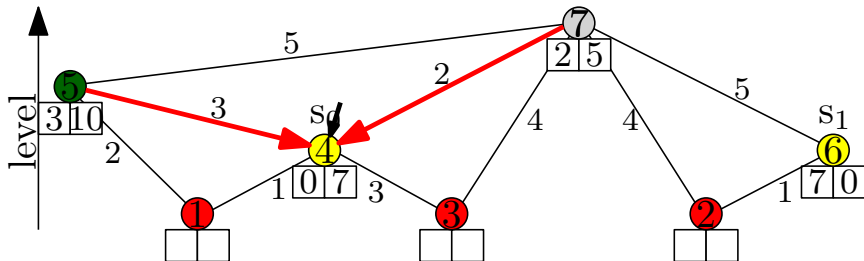
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



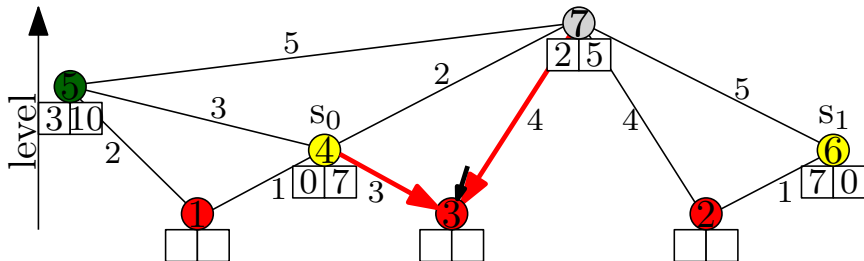
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



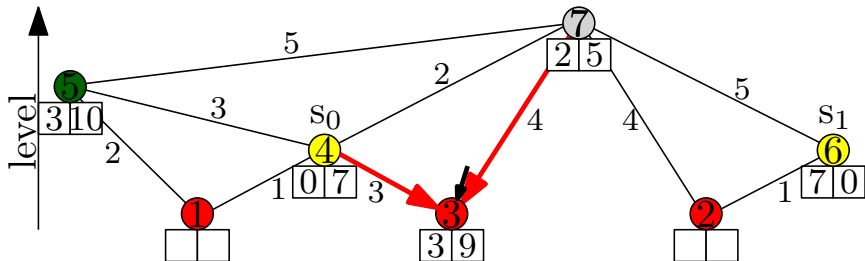
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



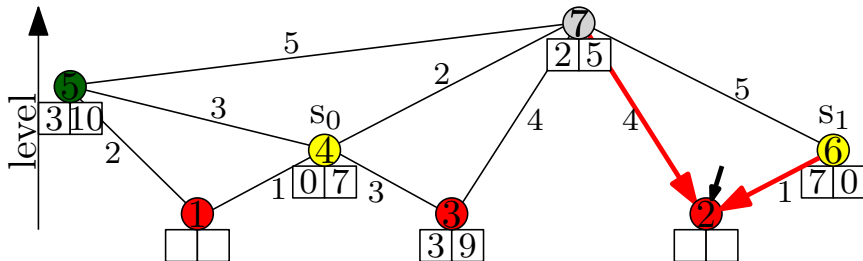
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



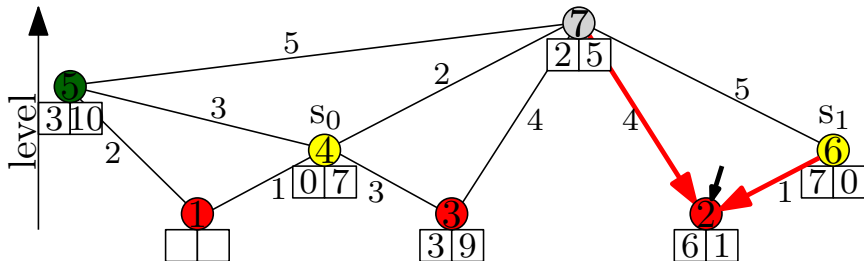
Szenario: Multiple Startknoten

- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten



Szenario: Multiple Startknoten

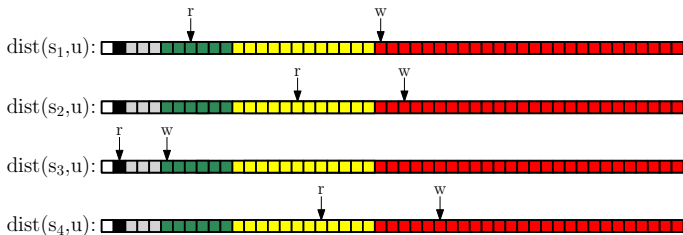
- k Vorwärtssuchen
- ein sweep (update aller k Werte)
- speicher Distanzlabel pro Knoten
- Vorteil: Mehr Daten pro Lesezugriff
- 96.8 ms pro Baum ($k = 16$)



- Bei mehreren Bäumen: PHAST-Operationen können mit SSE umgesetzt werden
- scanne 4 Distanzlabel auf einmal
- 37.1 ms pro Baum ($k = 16$)

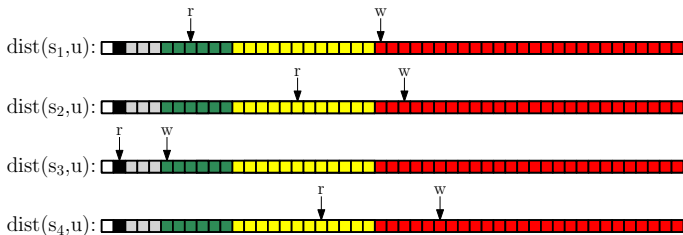
Parallelisierung

- Jeder Thread arbeitet unterschiedliche Startknoten ab



Parallelisierung

- Jeder Thread arbeitet unterschiedliche Startknoten ab

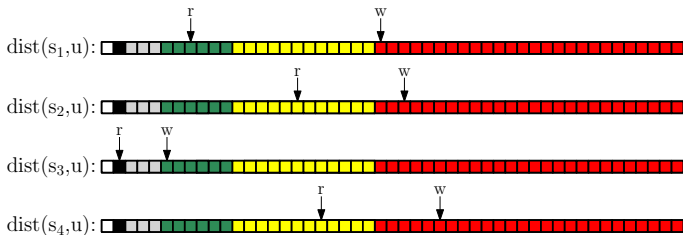


Ergebnisse:

- 16 Startknoten pro Sweep (updates via SSE)
- multi-core nach Startknoten \Rightarrow 64 Startknoten parallel (4 cores)
- 18.8 ms per Baum

Parallelisierung

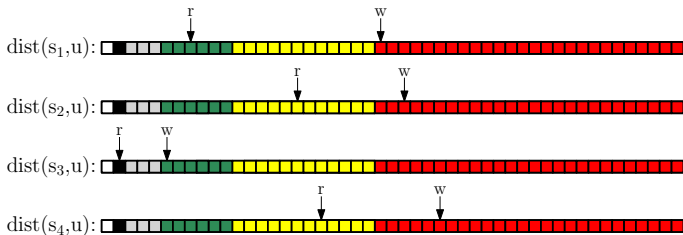
- Jeder Thread arbeitet unterschiedliche Startknoten ab



Ergebnisse:

- 16 Startknoten pro Sweep (updates via SSE)
- multi-core nach Startknoten \Rightarrow 64 Startknoten parallel (4 cores)
- 18.8 ms per Baum
- Warum kein perfekter Speedup?

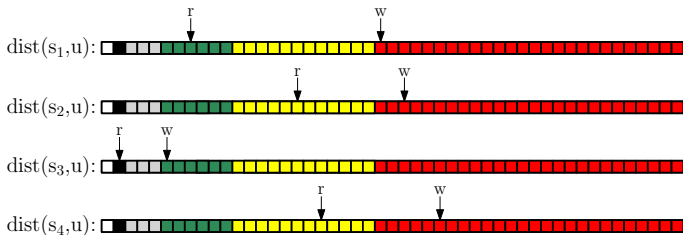
- Jeder Thread arbeitet unterschiedliche Startknoten ab



Ergebnisse:

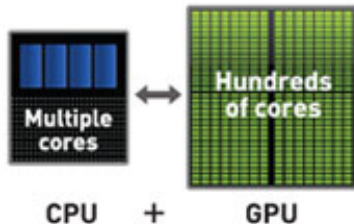
- 16 Startknoten pro Sweep (updates via SSE)
- multi-core nach Startknoten \Rightarrow 64 Startknoten parallel (4 cores)
- 18.8 ms per Baum
- Warum kein perfekter Speedup?
- lower bound tests zeigen: nah an Speicherbandbreite

- Jeder Thread arbeitet unterschiedliche Startknoten ab



Ergebnisse:

- 16 Startknoten pro Sweep (updates via SSE)
- multi-core nach Startknoten \Rightarrow 64 Startknoten parallel (4 cores)
- 18.8 ms per Baum
- Warum kein perfekter Speedup?
- lower bound tests zeigen: nah an Speicherbandbreite
- Kann eine GPU helfen?



Intel Xeon X5680:

- 3.33 GHz, oft ≥ 10 GB RAM
- 32 GB/s Speicherbandbreite
- 6 Kerne
- SIMD/SSE mit 4 floats/ints per Vektor

NVIDIA GTX 580:

- 772 MHz, 1.5 GB RAM
- 192 GB/s Speicherbandbreite
- 16 Kerne, 32 parallele Threads pro Kern \Rightarrow 512 parallele Threads
- eingeschränkte Berechnungen

Beobachtungen:

- Aufwärtsuche ist schnell
- Flaschenhals ist der lineare Sweep
- Speicherbandbreite das Problem

Beobachtungen:

- Aufwärtsuche ist schnell
- Flaschenhals ist der lineare Sweep
- Speicherbandbreite das Problem

Idee:

- speicher CH und Distanzarray auf der GPU
- Aufwärtssuche auf der CPU
- kopiere besuchte Knoten zur GPU (weniger als 2 kB)
- **Wichtig:** Nur die besuchten kopieren, nicht alle
- linearen Sweep auf der GPU

Beobachtungen:

- Aufwärtsuche ist schnell
- Flaschenhals ist der lineare Sweep
- Speicherbandbreite das Problem

Idee:

- speicher CH und Distanzarray auf der GPU
- Aufwärtssuche auf der CPU
- kopiere besuchte Knoten zur GPU (weniger als 2 kB)
- **Wichtig:** Nur die besuchten kopieren, nicht alle
- linearen Sweep auf der GPU

Problem:

- nicht genug Speicher auf GPU, um Tausende von Bäumen parallel zu bearbeiten
- wir müssen eine einzelne Baumberechnung parallelisieren

Paralleler Linearer Sweep

Beobachtung:

- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten



Paralleler Linearer Sweep

Beobachtung:

- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten



Paralleler Linearer Sweep

Beobachtung:

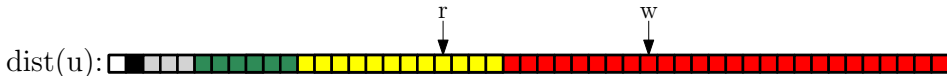
- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten



Paralleler Linearer Sweep

Beobachtung:

- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten



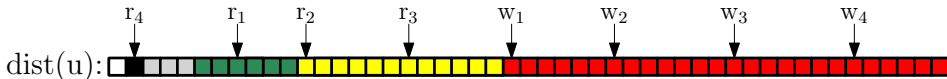
Paralleler Linearer Sweep

Beobachtung:

- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten

Idee:

- scanne alle Knoten auf Level i **parallel**
- Synchronisation nach jedem Level
- ein Thread pro Knoten



Paralleler Linearer Sweep

Beobachtung:

- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten

Idee:

- scanne alle Knoten auf Level i **parallel**
- Synchronisation nach jedem Level
- ein Thread pro Knoten



Paralleler Linearer Sweep

Beobachtung:

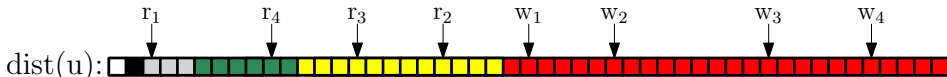
- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten

Idee:

- scanne alle Knoten auf Level i **parallel**
- Synchronisation nach jedem Level
- ein Thread pro Knoten

results:

- 5.5 ms auf NVIDIA GTX 580
- Beschleunigung von 511 gegenüber Dijkstra



Paralleler Linearer Sweep

Beobachtung:

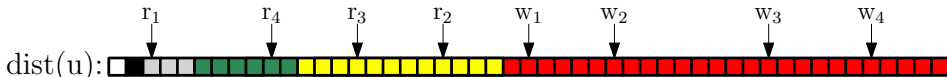
- beim Scannen von Level i :
 - nur eingehende Kanten von level $> i$ wichtig
 - Schreiben von Distanzlabeln in Level i , Lesen von Level $> i$
 - Distanzlabel für Level $> i$ sind korrekt
- scannen eines Level- i Knoten ist **unabhängig** von anderen Level- i Knoten

Idee:

- scanne alle Knoten auf Level i **parallel**
- Synchronisation nach jedem Level
- ein Thread pro Knoten

results:

- 5.5 ms auf NVIDIA GTX 580
- Beschleunigung von 511 gegenüber Dijkstra
- (mehrere Bäume: 2.2 ms)



PHAST auf 4-Kern Workstation (Core-i7 920)

sources/ sweep	time per tree [ms]					
	1 core		2 cores		4 cores	
1	171.9		86.7		47.1	
4	121.8	(67.6)	61.5	(35.5)	32.5	(24.4)
8	105.5	(51.2)	53.5	(28.0)	28.3	(20.8)
16	96.8	(37.1)	49.4	(22.1)	25.9	(18.8)

Werte in Klammern mit SSE aktiviert

PHAST auf Nvidia GTX 580

trees / sweep	memory [MB]	time [ms]
1	395	5.53
2	464	3.93
4	605	3.02
8	886	2.52
16	1448	2.21

algorithm	device	Europe		USA	
		time	distance	time	distance
Dijkstra	4-core workstation	947.72	609.19	1269.12	947.75
	12-core server	288.81	177.58	380.40	280.17
	48-core server	168.49	108.58	229.00	167.77
PHAST	4-core workstation	18.81	22.25	27.11	28.81
	12-core server	7.20	8.27	10.42	10.71
	48-core server	4.03	5.03	6.18	6.58
GPHAST	GTX 580	2.21	3.88	3.41	4.65

Beobachtung:

- Beschleunigung für Distanzmetrik geringer

All-Pairs Shortest Paths

Eingabe: Europa mit Reisezeiten

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	
	12-core server	60d	
	48-core server	35d	
PHAST	4-core workstation	94h	
	12-core server	36h	
	48-core server	20h	
GPHAST	GTX 580		

All-Pairs Shortest Paths

Eingabe: Europa mit Reisezeiten

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	
	12-core server	60d	
	48-core server	35d	
PHAST	4-core workstation	94h	
	12-core server	36h	
	48-core server	20h	
GPHAST	GTX 580	11h	

All-Pairs Shortest Paths

Eingabe: Europa mit Reisezeiten

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	
	12-core server	60d	
	48-core server	35d	
PHAST	4-core workstation	94h	
	12-core server	36h	
	48-core server	20h	
GPHAST	GTX 580	11h	

4-core workstation without GPU: 163 watts

4-core workstation with GPU: 375 watts

12-core server: 332 watts

48-core server: 747 watts

All-Pairs Shortest Paths

Eingabe: Europa mit Reisezeiten

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	2780.6
	12-core server	60d	1725.9
	48-core server	35d	2265.5
PHAST	4-core workstation	94h	55.2
	12-core server	36h	43.0
	48-core server	20h	54.2
GPHAST	GTX 580	11h	14.9

4-core workstation without GPU: 163 watts

4-core workstation with GPU: 375 watts

12-core server: 332 watts

48-core server: 747 watts

bis jetzt:

- nur Distanzen berechnet, nicht Bäume

bis jetzt:

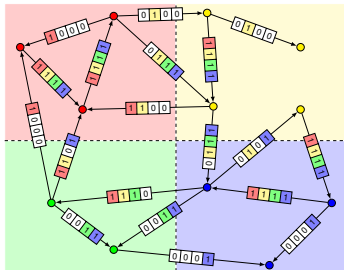
- nur Distanzen berechnet, nicht Bäume

Idee:

- Iteration über alle Kanten (1 Thread pro Kante)
- Wenn $d(v) + \text{len}(v, u) = d(u)$ dann ist v der Vorgänger von u (sofern kürzeste Wege eindeutig sind)

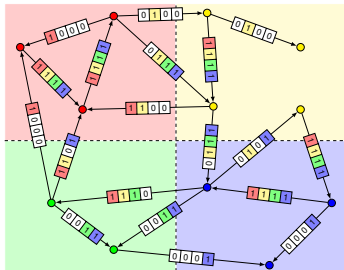
Idee:

- benutze GPHAST zum Berechnen der Bäume von den Randknoten aus



Idee:

- benutze GPHAST zum Berechnen der Bäume von den Randknoten aus
- setze Flaggen durch zusätzlichen Sweep auf GPU
Wichtig, weil alle Bäume "auf CPU kopieren" teuer ist
- Vorberechnung sinkt von 17 Stunden auf 3 Minuten



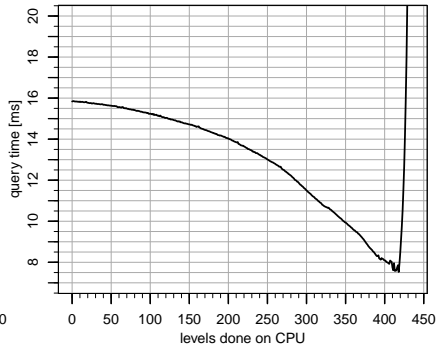
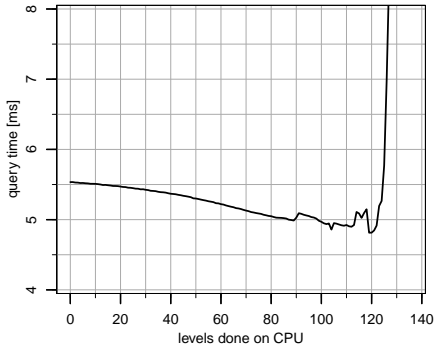
Beobachtung:

- Synchronization des Level kostet Zeit auf der GPU ($5 \mu s$ pro Level)
- obere Level sind klein

Idee:

- beginne linearen Sweep auf CPU (bis Level k)
- kopiere Suchraum und alle Distanzlabel für Knoten oberhalb k zur GPU
- restlicher Scan auf der GPU

Reisezeiten vs. Reisedistanzen



Es lohnt sich, ein paar Level auf der CPU zu berechnen.

Das One-to-Many Problem

Problemdefinition:

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

Problemdefinition:

- Eingabe: ein 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)
 - ⇒ Performance stark abhängig von $|T|$ und Verteilung von T

Problemdefinition:

- Eingabe: ein 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)
 - ⇒ Performance stark abhängig von $|T|$ und **Verteilung** von T
- $|T|$ p2p Anfragen (z.B. CH)
 - ⇒ Performance stark abhängig von $|T|$

Problemdefinition:

- Eingabe: ein 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)
 - ⇒ Performance stark abhängig von $|T|$ und Verteilung von T
- $|T|$ p2p Anfragen (z.B. CH)
 - ⇒ Performance stark abhängig von $|T|$
- benutze PHAST (kein Stoppkriterium!)
 - ⇒ Overkill (vor allem für kleine T)

Erste Ideen

Vorschläge?

Definition:

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

$$\forall (s, t_1, t_2) \in V^3 : \vec{\sigma}(s, t_1) = \vec{\sigma}(s, t_2) \quad \text{und}$$

$$\forall (s_1, s_2, t) \in V^3 : \overleftarrow{\sigma}(s_1, t) = \overleftarrow{\sigma}(s_2, t)$$

Definition:

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

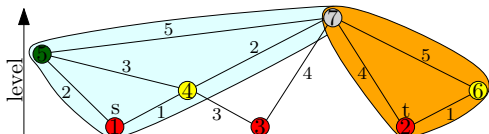
$$\forall (s, t_1, t_2) \in V^3 : \vec{\sigma}(s, t_1) = \vec{\sigma}(s, t_2) \quad \text{und}$$
$$\forall (s_1, s_2, t) \in V^3 : \overleftarrow{\sigma}(s_1, t) = \overleftarrow{\sigma}(s_2, t)$$

Beispiele:

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

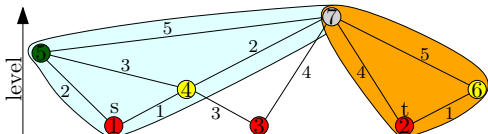
Beobachtung:

- suchen nur aufwärts
- sind nicht zielgerichtet



Beobachtung:

- suchen nur aufwärts
- sind nicht zielgerichtet

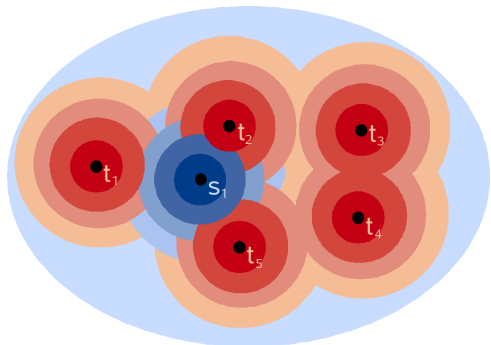


somit:

- Bidirektionaler Dijkstra
- Reach
- (Customizable) Contraction Hierarchies
- ohne Stoppkriterium, lass laufen bis Queues leer sind

Idee:

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



Problem:

- Verwalten der Suchräume?

während Rückwärtssuchen (target selection phase):

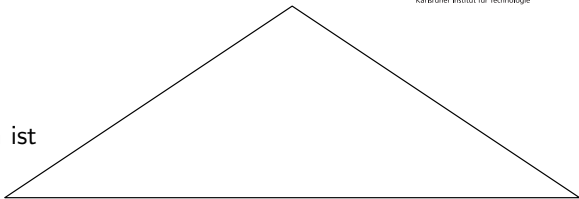
- je $t \in T$:
- starte Suche im Rückwärts-DAG, breche nicht ab
- für jedes erreichte u :
- speichere einen Bucket $\beta(u)$ mit $(t, d(u, t))$
Alternative Sichtweise: füge gewichtete Abwärtskanten zu erreichbaren Zielknoten ein

während Vorwärtssuche (query phase):

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

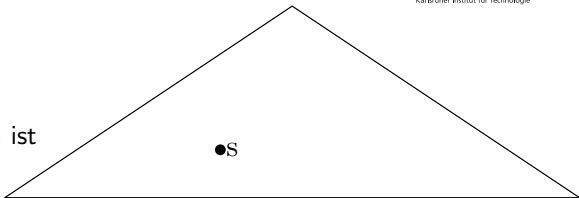
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



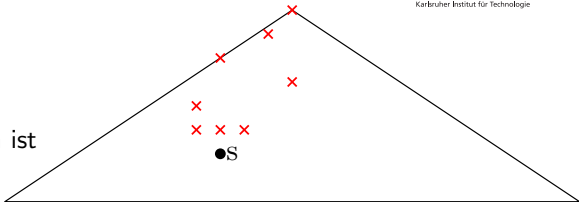
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



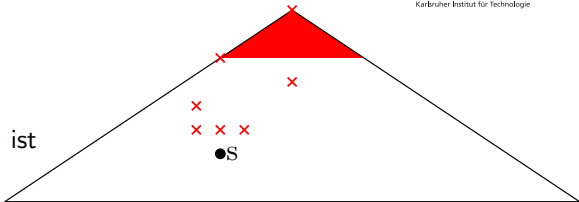
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



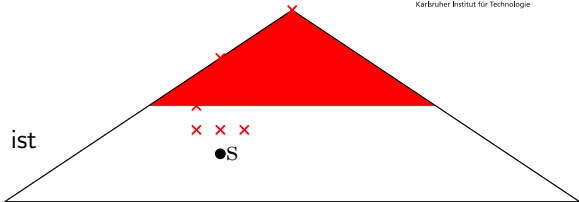
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



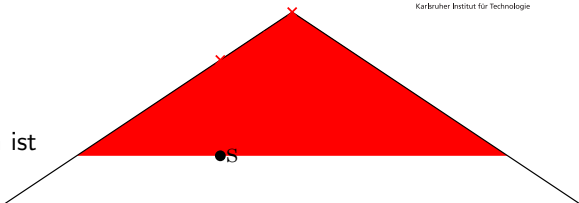
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



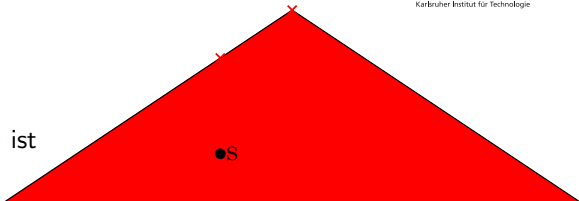
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



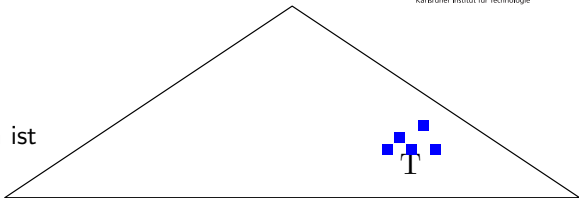
Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

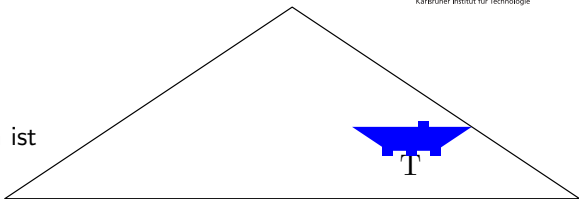


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

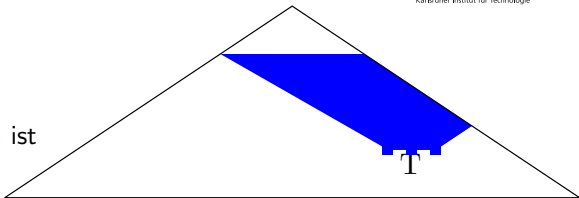


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

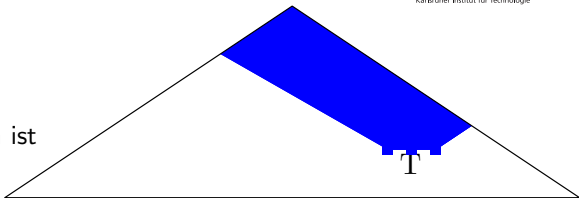


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

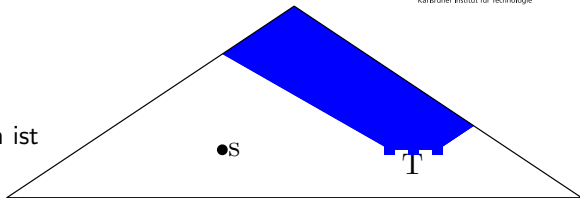


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

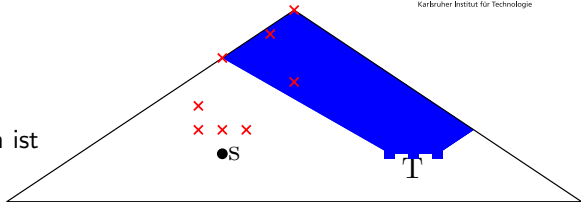


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)
- Aufwärtssuche im vollen Graphen

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

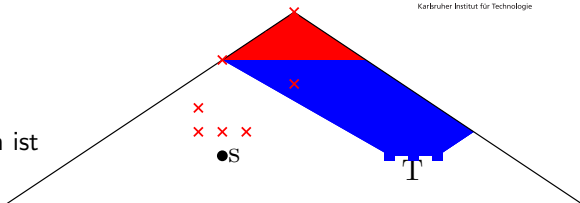


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)
- Aufwärtssuche im vollen Graphen

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

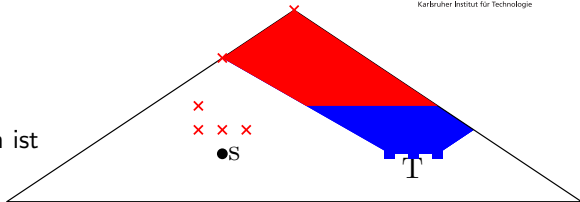


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

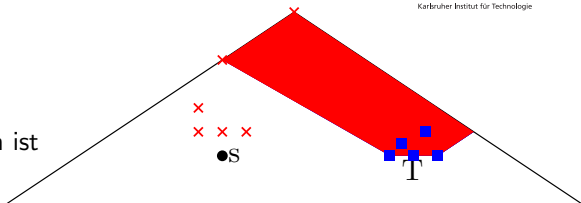


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals

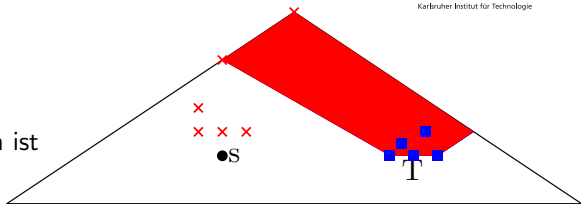


Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

Beobachtung PHAST:

- Sweep über den Graphen ist der Flaschenhals



Idee:

- **extrahiere** relevanten Teil des Graphen (Zielselektion)
- Aufwärtssuche im vollen Graphen
- Sweep auf extrahiertem Graphen

⇒

- Startknoten kann im ganzen Graphen liegen
- Größe des extrahierten Graphen hängt von Verteilung und Anzahl T ab
- kann wie PHAST parallelisiert werden
- GPU-Implementierung möglich

Problem:

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

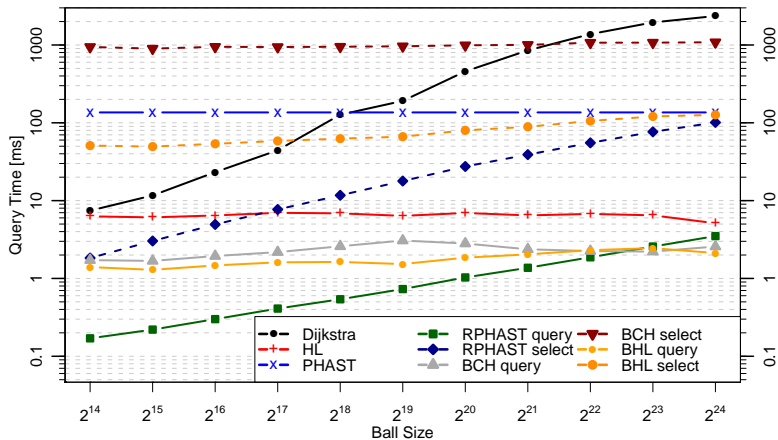
Setup:

- 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

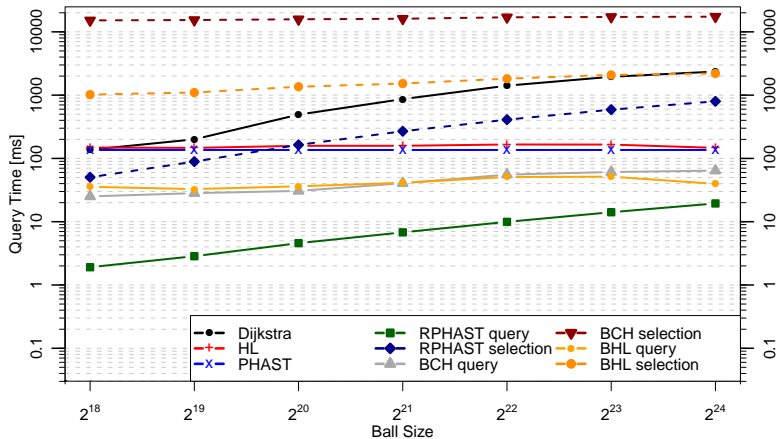
Experimente I

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



Experimente II

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



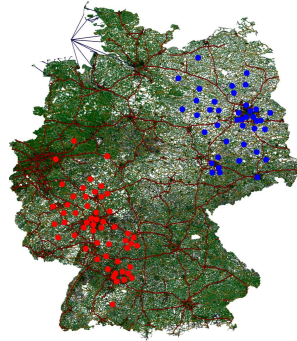
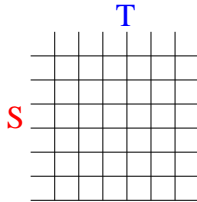
Many-to-Many Kürzeste Wege

Gegeben:

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

Gesucht:

- Distanzmatrix D



Many-to-Many Kürzeste Wege

Gegeben:

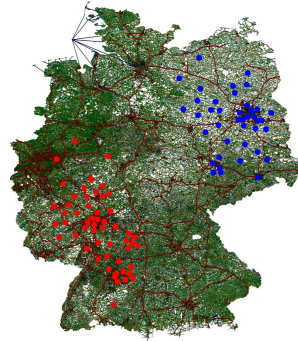
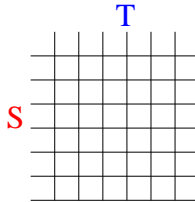
- Graph
- Knotenmengen $S, T \in V$

Gesucht:

- Distanzmatrix D

Anwendungen:

- vehicle routing
- traveling salesman



Many-to-Many Kürzeste Wege

Gegeben:

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

Gesucht:

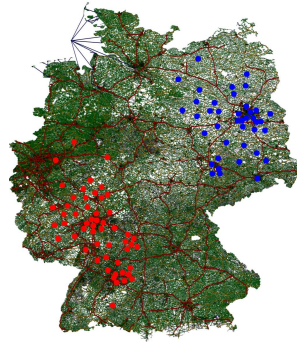
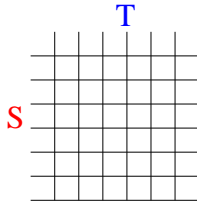
- Distanzmatrix D

Anwendungen:

- vehicle routing
- traveling salesman

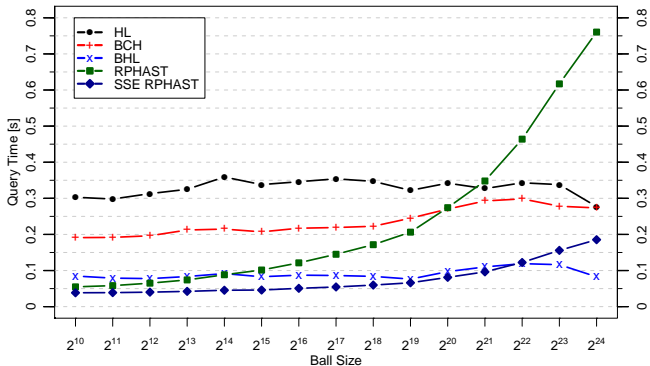
Lösung:

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



Experimente I

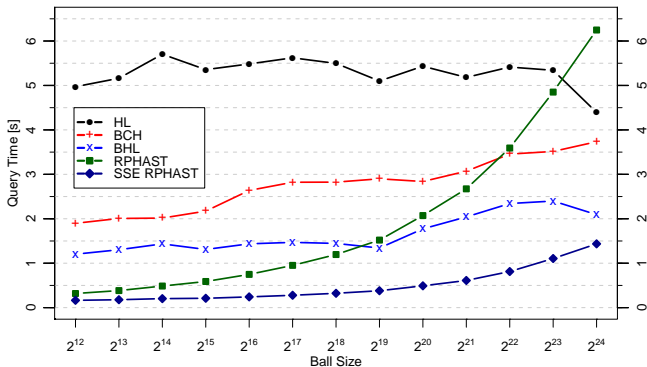
input: Westeuropa (18M Knoten), $|S| = |T| = 2^{10}$



Beobachtung: alle Techniken unter einer Sekunde

Experimente II

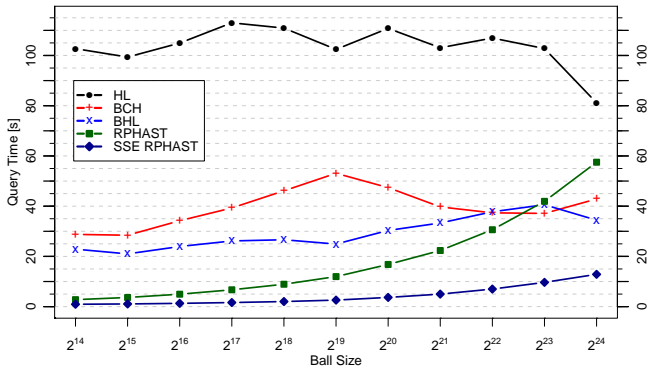
input: Westeuropa (18M Knoten), $|S| = |T| = 2^{12}$



Beobachtung: SSE PHAST am schnellsten

Experimente III

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



Beobachtung: SSE PHAST am schnellsten

 Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck.

HLDB: Location-Based Services in Databases.

In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pages 339–348. ACM Press, 2012.

Best Paper Award.

 Richard Bellman.





On a Routing Problem.





Quarterly of Applied Mathematics, 16:87–90, 1958.

 George B. Dantzig.

Linear Programming and Extensions.

Princeton University Press, 1962.


-  Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck.
PHAST: Hardware-accelerated shortest path trees.
Journal of Parallel and Distributed Computing, 73(7):940–952, 2013.
-  Daniel Delling, Andrew V. Goldberg, and 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)*, volume 20 of *OpenAccess Series in Informatics (OASICS)*, pages 52–63, 2011.
-  Edsger W. Dijkstra.
A Note on Two Problems in Connexion with Graphs.
Numerische Mathematik, 1(1):269–271, 1959.
-  Lester R. Ford, Jr. and Delbert R. Fulkerson.
Flows in Networks.
Princeton University Press, 1962.

-  Robert W. Floyd.
Algorithm 97: Shortest path.
Communications of the ACM, 5(6):345, 1962.
-  Michael L. Fredman and Robert Tarjan.
Fibonacci heaps and their uses in improved network optimization algorithms.
Journal of the ACM, 1987.
-  Andrew V. Goldberg.
A Simple Shortest Path Algorithm with Linear Average Time.
In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA'01)*,
volume 2161 of *Lecture Notes in Computer Science*, pages 230–241, 2001.
-  Donald B. Johnson.
Efficient Algorithms for Shortest Paths in Sparse Networks.
Journal of the ACM, 24(1):1–13, January 1977.

 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. SIAM, 2007.

 Ulrich Meyer and Peter Sanders.

Δ -Stepping: A Parallelizable Shortest Path Algorithm.

Journal of Algorithms, 49(1):114–152, 2003.

 Stephen Warshall.

A Theorem on Boolean Matrices.

Journal of the ACM, 9(1):11–12, 1962.