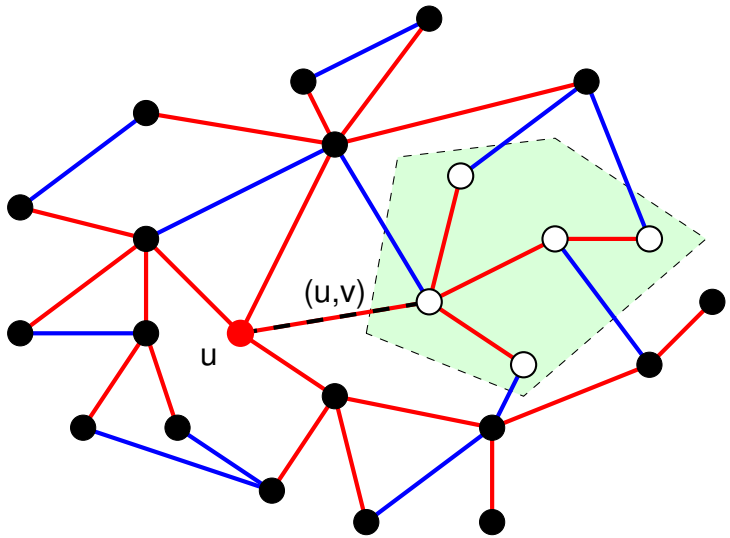
Algorithmen für Routenplanung – Vorlesung 4

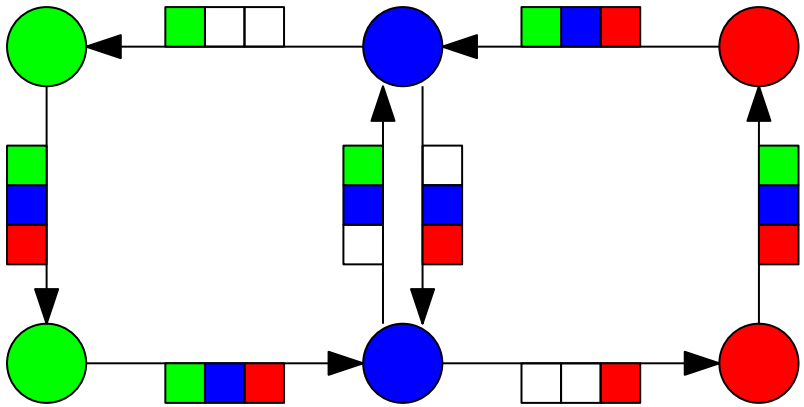
Daniel Delling

Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Letztes Mal

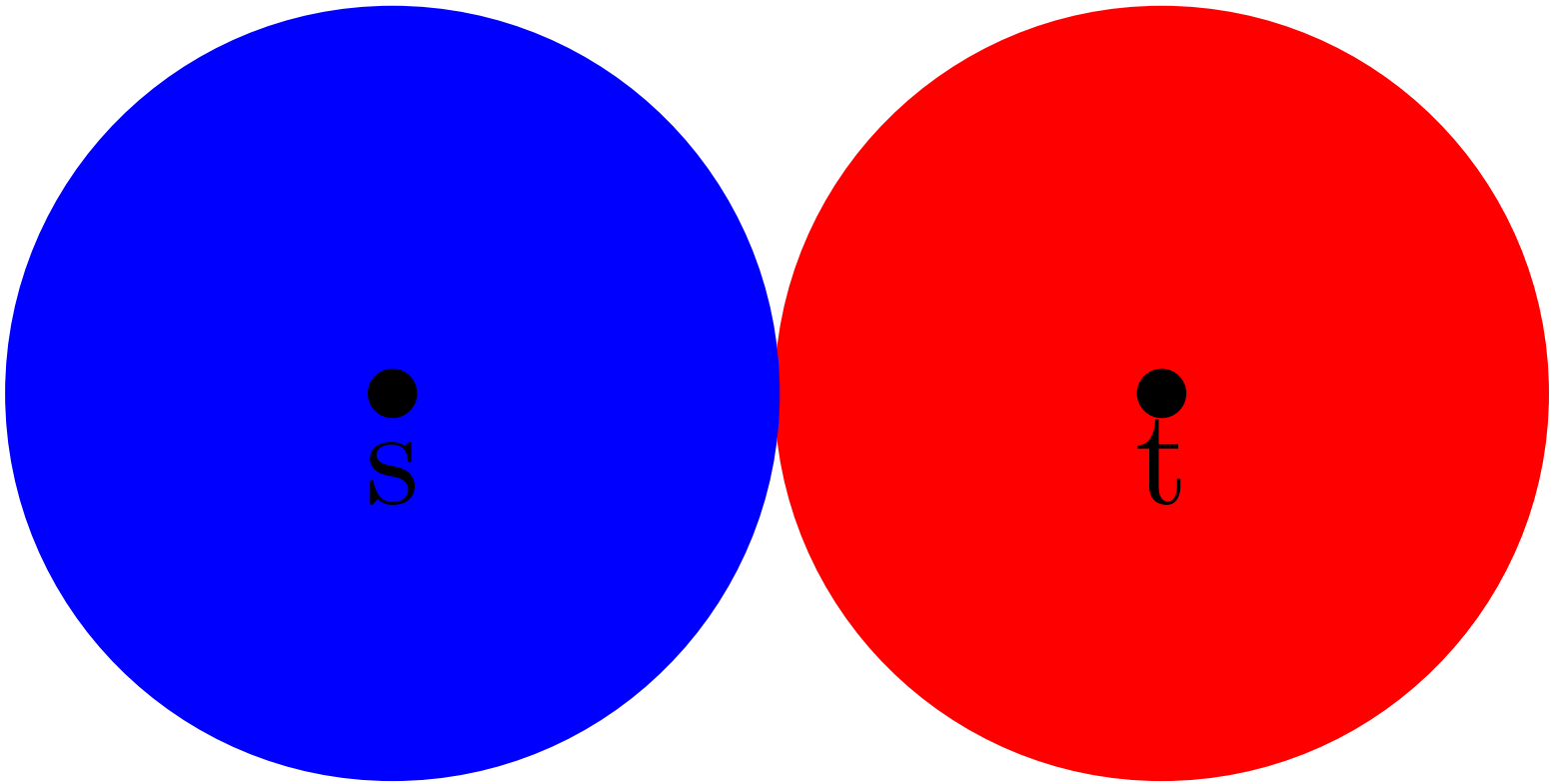
- Geometrische Container
- Arc-Flags





Themen Heute + nächste 1-2

- Hierarchische Techniken



Ideensammlung

Wie Suche hierarchisch machen?

- ≫ identifiziere wichtige Knoten mit Zentralitätsmaß
- ≫ überspringe unwichtige Teile des Graphen

heute ersteres

Zentralitätsmaße

Zentralitätsmaße bewerten wichtigkeit eines Knoten oder einer Kante

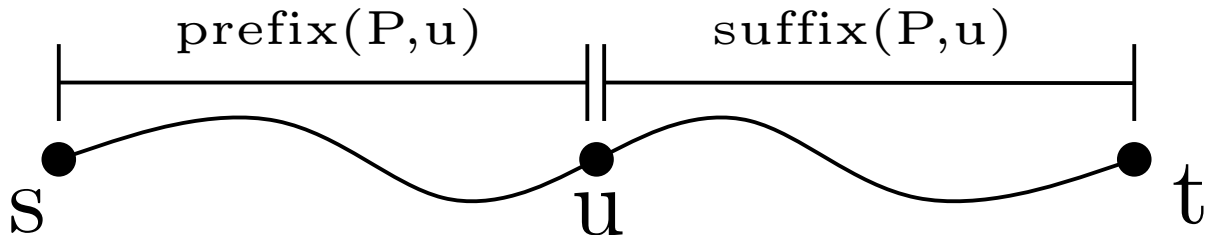
Beispiele:

- betweenness
- centrality

Idee:

- berechne Zentralitätsmaß
- benutze ZM zum prunen von Knoten oder Kanten

Reach



Definition:

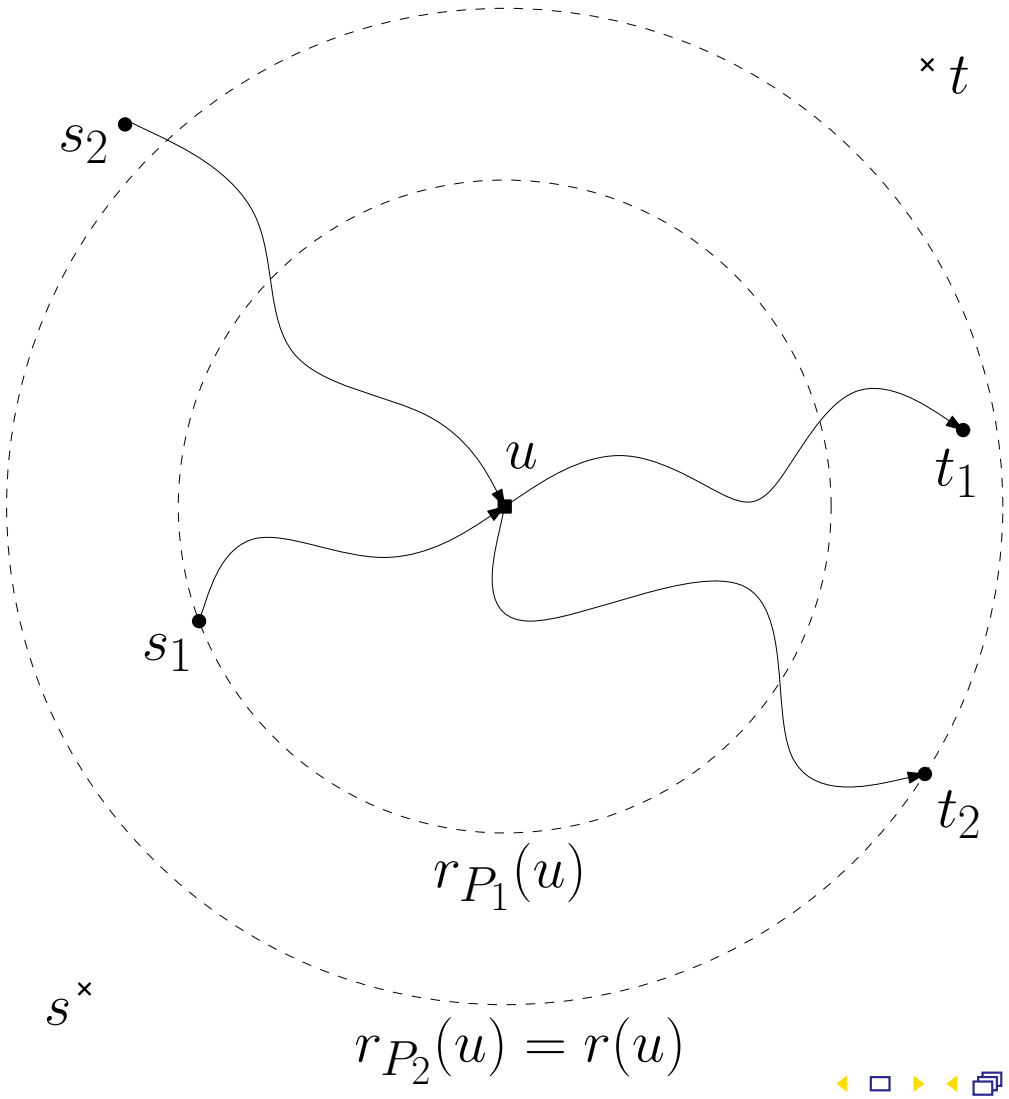
- sei $P = \langle s, \ldots, u, \ldots, t \rangle$ Pfad durch u
- dann reach von u bezüglich P:
  $r_P(u) := \min\{len(P_{su}), len(P_{ut})\}$
- reach von u: maximum seiner reachwerte bezüglich aller kürzester Pfade durch u:

$$r(u) := \max\{r_P(u) \ : \ P \text{ kürzester Weg mit } u \in P\}$$

Reach-Pruning

somit:

- reach $r(u)$ von u gibt suffix oder prefix des längsten kürzesten Weges durch u
- wenn für u während query $r(u) < d(s,u)$ und $r(u) < d(u,t)$ gilt, kann u geprunt werden



Reach Dijkstra - Pseudocode

```
ReachDijkstra(G = (V, E), s, t)
1  d[s] = 0
2  Q.clear(), Q.add(s, 0)
3  while !Q.empty() do
4      u ← Q.deleteMin()
5      if r(v) < d[u] && r(u) < d(u, t) then
6          continue
7      for all edges e = (u, v) ∈ E do
8          if d[u] + len(e) < d[v] then
9              d[v] ← d[u] + len(e)
10             if v ∈ Q then Q.decreaseKey(v, d[v])
11             else Q.insert(v, d[v])
```

Probleme

Probleme:

- Abfrage $r(v) < d(v,t)$

Lösung:

- Knotenpotential $\pi(v)$ gibt untere Schranke zum Ziel wenn $\pi(t) = 0$
- benutze A^* für Abschätzung (euklidisch oder Landmarken)
- gut kombinierbar mit A^*

Potentiale durch Landmarken

Idee:

- wähle Landmarken L aus V (≈ 16)
- berechne Distanzen von und zu allen Landmarken
- dann gilt:

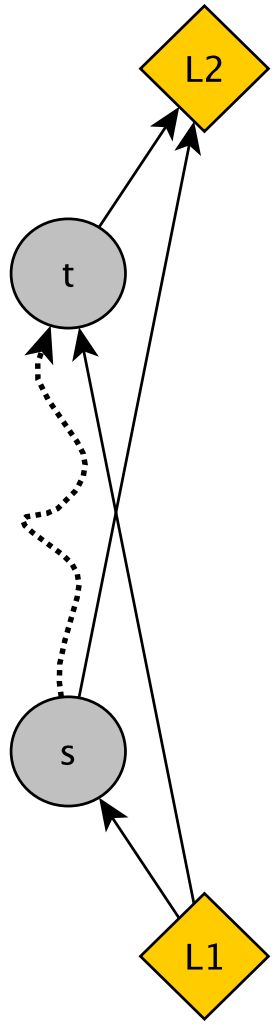
$$\begin{aligned} d(u,t) &\geq d(L_1,t) - d(L_1,u) \\ d(u,t) &\geq d(u,L_2) - d(t,L_2) \end{aligned}$$

für alle $u \in V$.

- somit ist

$$\pi(u) = \max_{\ell \in L}\{\max\{d(\ell,t) - d(\ell,u), d(u,\ell) - d(t,\ell)\}\}$$

ein gültiges Potential



A^* + Reach

$A^*\texttt{Reach}(G = (V, E), s, t)$

```
1  d[s] = 0
2  Q.clear(), Q.add(s, π(s))
3  while !Q.empty() do
4      u ← Q.deleteMin()
5      if r(u) < d[u] && r(u) < π(u) then
6          continue
7      for all edges e = (u, v) ∈ E do
8          if d[u] + len(e) < d[v] then
9              d[v] ← d[u] + len(e)
10             if v ∈ Q then Q.decreaseKey(v, d[v] + π(v))
11             else Q.insert(v, d[v] + π(v))
```

Nachteil A Reach*

Problem:

- Potentiale nicht verfügbar
- Potentiale können schlechte Abschätzung sein $\Rightarrow$ schwaches Pruning

Lösung:

- benutze bidirektionalen Anfragealgo
- zwei Ansätze:
  - self-bounding
  - distance-bounding

Bidirectional Self-Bounding Reach-Dijkstra

Idee (für Vorwärtssuche):

- prune, wenn $d[u] > r(u)$ gilt
- überlasse den Check $d(u,t) > r(u)$ der Rückwärtssuche
- Rückwärtssuche analog
- ändere Stoppkriterium

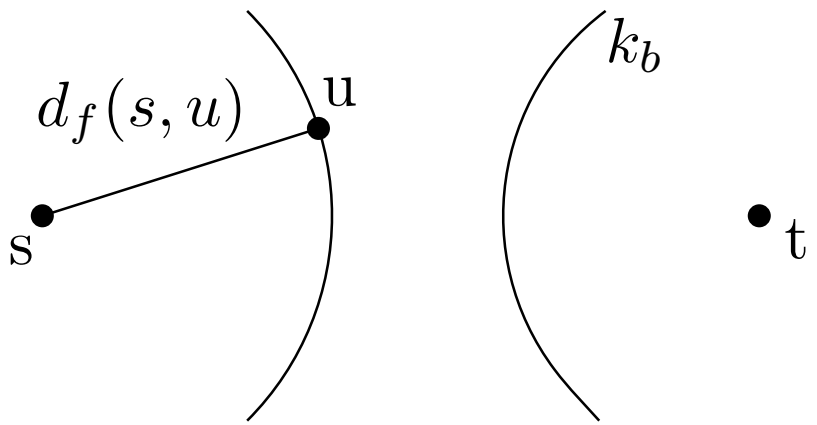
neues Stoppkriterium:

- stoppe Suche in eine Richtung wenn Queue leer oder es gilt: $k_f > \mu/2$
- stoppe Anfrage, wenn beide Suchrichtungen gestoppt haben
- Beweis in Übung

Bidirectional Distance-Bounding Reach-Dijkstra

Idee (für Vorwärtssuche):

- wenn u von Rückwärtssuche noch nicht erreicht, ist k_b eine untere Schranke für $d(u,t)$
- wenn u von Rückwärtssuche abgearbeitet, $d(u,t)$ bekannt



somit:

- prune, wenn $r(u) < d_f[u]$ und $r(u) < \min\{d_b[u], k_b\}$
- Stoppkriterium bleibt erhalten (also $k_f + k_b \geq \mu$
- wenn als Alternierungsstrategie $\min\{k_f, k_b\}$ gewählt, gilt für Vorwärtssuche: $d_f[u] \leq k_b$
- kann zusätzlich mit A^* kombiniert werden

Optimierungen

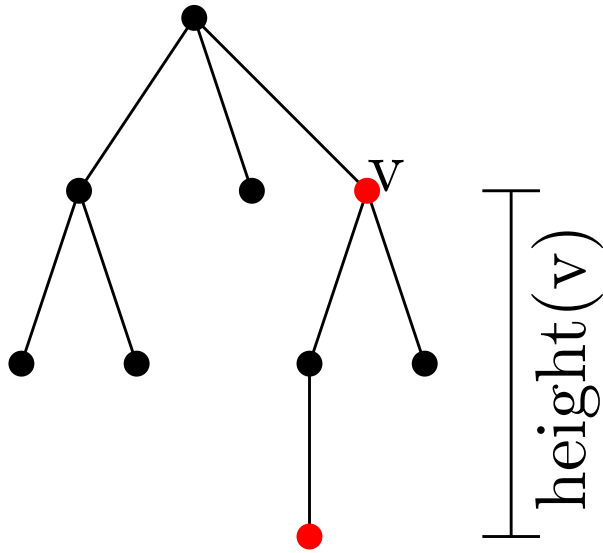
mögliche Verbesserungen:

- Kanten pruning: (u, v) muss nicht relaxiert werden, wenn gilt:
  - $d[u] + len(u, v) > r(v)$
  - und $r(v) < \min\{d_b[u], k_b\}$
- Kanten sortieren:
  - sortiere ausgehende Kanten (u, v_i) absteigende nach $r(v_i)$
  - wenn Kante relaxiert wird mit $r(v_i) < k_b$ und $r(v_i) < d_f[u]$ müssen die restlichen Kanten ausgehend von u nicht relaxiert werden

Vorberechnung: Erster Ansatz

Wie kann man Reach Werte vorberechnen?

- initialisieren $r(u) = 0$ für alle Knoten
- für jeden Knoten u
  - konstruiere kürzeste Wege-Baum
  - höhe von Knoten v: Abstand v zum am weitesten entfernten Nachfolger
  - für jeden Knoten v: $r(v) = \max\{r(v), \min\{d(u,v), \mathit{height}(v)\}\}$



altes Problem:

- Vorberechnung basiert auf all-pair-shortest paths
- somit wieder 500 Jahre Vorberechnung

Approximation

Beobachtung:

- es genügt, für jeden Knoten eine obere Schranke des reach Wertes zu haben

Problem:

- untere Schranken einfach zu finden:
  - breche Konstruktion der Bäume einfach bei bestimmter Größe ab
- aber: untere Schranken sind unbrauchbar
- Berechnung von oberen Schranken deutlich schwieriger

Der RE-Algorithmus

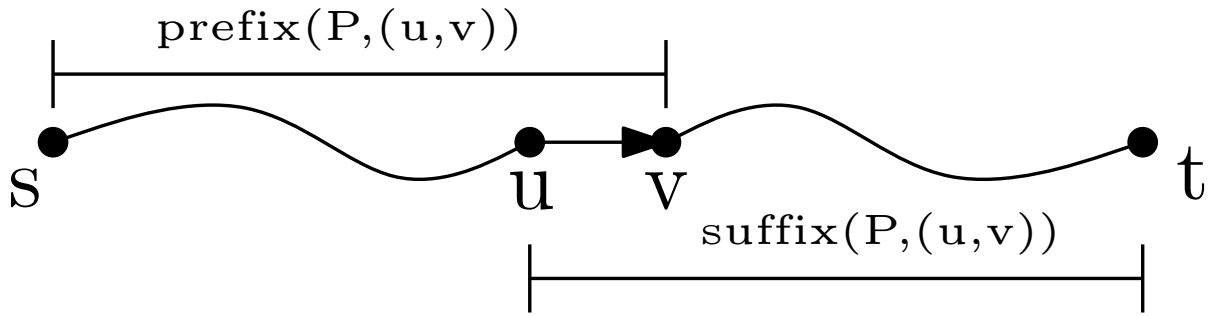
Erweiterungen gegenüber reinem Reach:

- Kanten-Reach für Vorberechnung
- obere Schranken durch iterative Berechnung
- Shortcuts

Ergebnisse:

- schnellere Vorberechnung
- deutlich schnellere Anfragen

Kanten-Reach



Definition:

- sei $P = \langle s, \ldots, u, v, \ldots, t \rangle$ Pfad durch (u, v)
- dann reach von (u, v) bezüglich P:
  $r_P((u, v)) := \min\{len(P_{sv}), len(P_{ut})\}$
- reach von (u, v): maximum seiner reachwerte bezüglich aller kürzester Pfade durch (u, v):

$$r((u, v)) := \max\{r_P((u, v)) \ : \ P \text{ kürzester Weg mit } (u, v) \in P\}$$

Approximative Berechnung von Kanten-Reach

Idee:

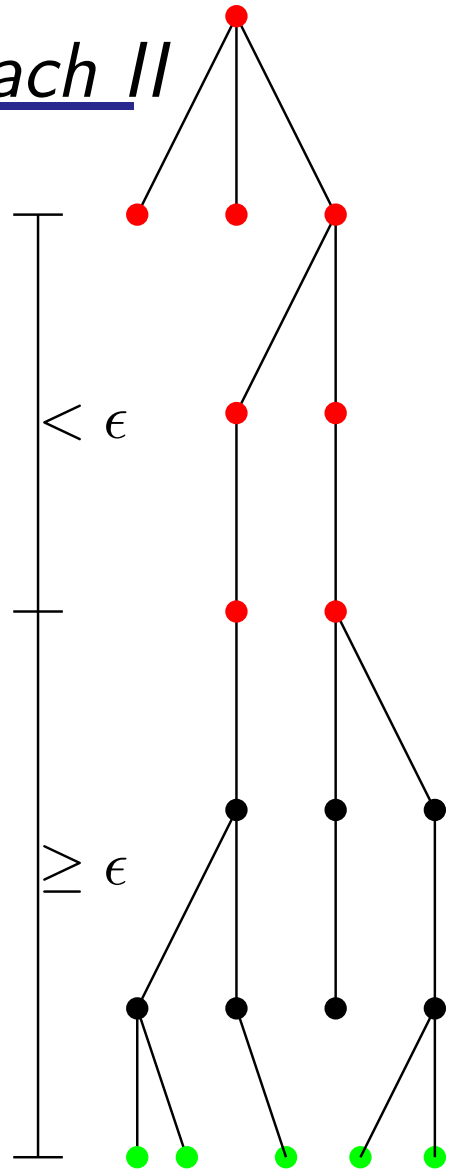
- gegeben: Graph und Schranke ϵ
- gesucht:
  - Gültige obere Reach-Schranken für alle Kanten mir reach kleiner ϵ
  - alle andere Kanten sollen reach von ∞ gesetzt bekommen
  - dabei dürfen Kanten mit reach kleiner ϵ auch den Wert ∞ gesetzt bekommen, also false negatives erlaubt
- Problem: um $r(u,v) < \epsilon$ zu garantieren, müssen alle kürzesten Wege durch (u,v) berücksichtigt werden.

Beobachtung:

- es gibt Pfad $P_{st} = \langle s, s', \ldots, u, v, \ldots, t', t \rangle$ Pfad durch (u,v) mit $r_{P_{s't}}(u,v) < \epsilon$ und $r_{P_{st'}}(u,v) < \epsilon$
- genannt ϵ-minimal bezüglich (u,v)
- es reicht, ϵ-minimale Pfade zu finden (ohne Beweis)

Approximative Berechnung von Kanten-Reach II

- partieller KW-Baum T_w von jedem $w \in V$
- stoppe, wenn garantiert, dass alle ϵ-minimalen Pfade, die bei w starten gefunden worden sind
- u ist innerer Knoten: $u = w$ oder $d(x, u) < \epsilon$ mit x erster Knoten nach w zwischen u und w
- stoppe, wenn Queue leer oder wenn Abstand von alle Knoten in der Queue zu nähestem inneren Knoten bzgl. T_w größer gleich ϵ ist
- für jeden inneren Knoten u
  - berechne Höhe bezüglich T_w
  - reach für (u, v) bezüglich T_w ist $\min\{d(w, u), height_{T_w}(u)\}$
- $r(u, v) = max_{w \in V}\{r_{T_w}(u, v)\}$
- wenn $r(u, v) \geq \epsilon$ setze $r(u, v) = \infty$



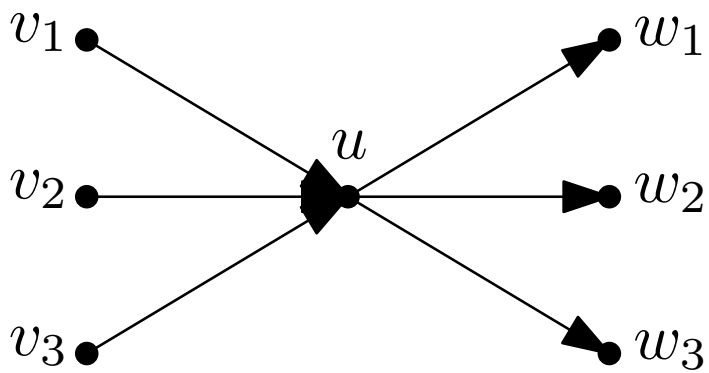
Iterative Berechnung von Knoten-Reach

Berechnen von Reach:

- wähle ϵ frei
- iterativ, solange $E \neq \emptyset$
  - berechne obere Schranken mit vorherigem Verfahren
  - entferne alle Kanten mit reach $< \epsilon$ aus Graphen
  - setze $\epsilon = k \cdot \epsilon$
- wandle Kanten-Reach in Knoten-Reach um

Umrechnung:

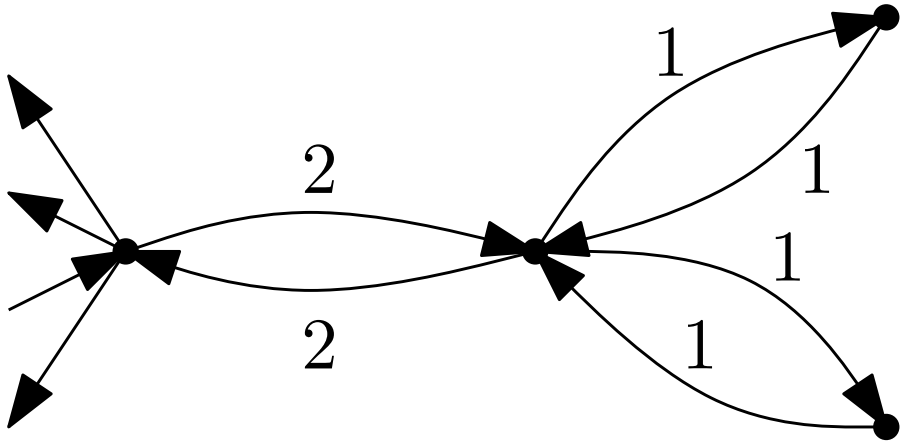
$$r(u) \leq \min \left\{ \max_i\{r(v_i, u)\}, \max_i\{r(u, w_i)\} \right\}$$



Penalties

Problem:

- durch entfernen von Knoten und Kanten ändern sich die Längen der kürzesten Wege



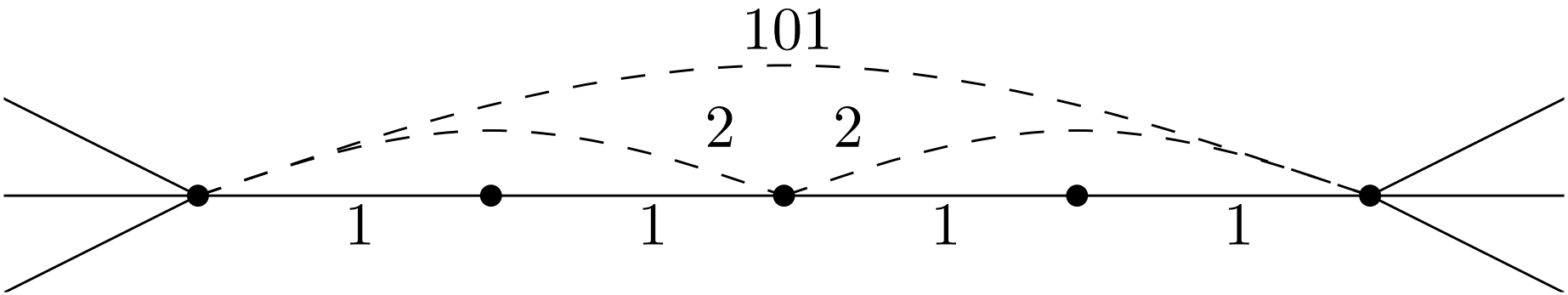
Lösung:

- halte für jeden Knoten zwei Werte vor:
  - $inPenalty(u)$: maximum der reachs entfernter (w, u) Kanten
  - $outPenalty(u)$: maximum der reachs entfernter (u, w) Kanten
- während Vorberechnung:
  - Hänge an jeden Knoten v Pseudoknoten v' mit Abstand $outPenalty(v)$
  - bestimmte neue Höhe $height_p(u)$
  - dann $r_{T_w}(u, v) = \min\{d(u, w) + inPenalty(u), height_p(u)\}$

Shortcuts

Beobachtung:

- lange modellierte Pfade in Straßennetzwerken
- erscheint unnötig, alle diese Knoten anzuschauen



Idee:

- füge iterativ Shortcuts ein
- Shortcuts verkleinern reach Werte

Kontraktion des Graphen

Vorgehen:

- überspringe iterativ Knoten v aus Graphen
- für jede Kante eingehende Kante (u, v) und ausgehende (v, w)
  - füge Shortcut (u, w) mit $len(u, w) = len(u, v) + len(v, w)$ ein
  - wenn Kante schon existiert:
    $len(u, w) = \min\{len(u, w), len(u, v) + len(v, w)\}$
  - $r(u, v) = len(u, v) + outPenalty(v)$,
    $r(v, w) = len(v, w) + inPenalty(v)$
  - entferne (u, v), (v, w) und v aus Graphen

Reihenfolge:

- Reihenfolge wie Knoten entfernt werden, ändert das Ergebnis
- benutze Priority Queue zum Verwalten, wenn als nächstes
- Expansion $c(v) = \deg_{in}(v) \cdot \deg_{out}(v)/(\deg_{in}(v) + \deg_{out}(v))$
- stoppe wenn $c(v) > C$, (meist $C = 1.5$ gewählt)

Verfeinerung der Reach Werte

Problem:

- Approximation schaukelt sich auf
- hohe Werte werden massiv überschätzt
- diese Knoten genau die wichtige für die Anfragen

Idee:

- nach voller Vorberechnung
- extrahiere δ Knoten mit höchstem Reach
- berechne exakten Reach mit APSP für diesen Subgraphen (mit Penalties)
- wenn neuer Reach wert kleiner, aktualsiere

RE

Vorberechnung:

- wähle ϵ frei
- iterativ, solange $E \neq \emptyset$
  - Kontrahiere Graphen
  - berechne obere Schranken mit vorherigem Verfahren
  - entferne alle Kanten mit reach $< \epsilon$ aus Graphen
  - setze $\epsilon = k \cdot \epsilon$
- wandle Kanten-Reach in Knoten-Reach um
- verfeinere Knoten-Reach Werte

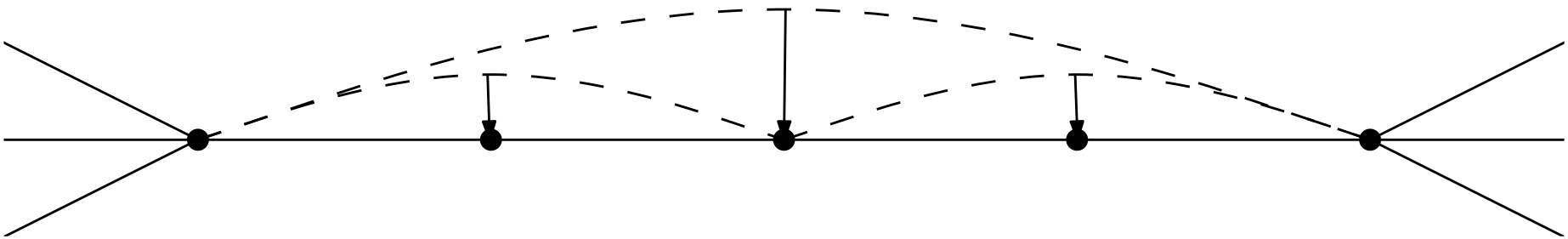
Anfrage:

- Bidirectional Distance-Bounding Reach-Dijkstra auf Graphen mit Shortcuts

Entpacken von Shortcuts

Problem:

- Anfrage auf Graphen mit Shortcuts
- Distanzen bleiben erhalten
- kürzester Weg enthält jetzt viele Shortcuts
- schlecht, wenn wir den gesamten Pfad haben wollen



Lösung

- jeder Shortcut überspringt genau einen Knoten
- speicher Mittelknoten für jeden Shortcut
- entpacke rekursiv bei Bedarf

Kombination mit ALT

Beobachtung:

- RE-algorithmus hierarchisch
- nicht zielgerichtet
- gut kombinierbar mit ALT

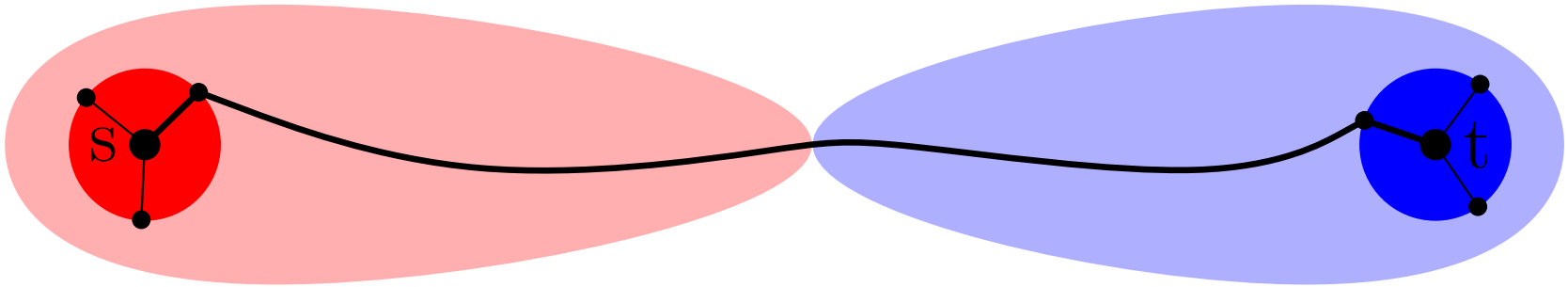
REAL-Algorithmus

- RE + ALT
- Vorberechnung unabhängig voneinander
- Anfragen mit Bidirectional Distance-Bounding ALT-Reach-Dijkstra auf Graphen mit Shortcuts

Partielle Landmarken

Beobachtung:

- Landmarken brauchen viel Speicher
- während Anfragen werden meist Knoten mit hohem reach betrachtet



Idee:

- speichere Landmarken-Informationen nur für Knoten mit reach $> R$
- 2-Phase Anfrage Algorithmus
  - reiner Reach, relaxiere keine Kanten ausgehend von Knoten mit Reach $> R$
  - wenn kürzester Weg gefunden, fertig
  - sonst starte REAL Query von allen Knoten mit $> R$

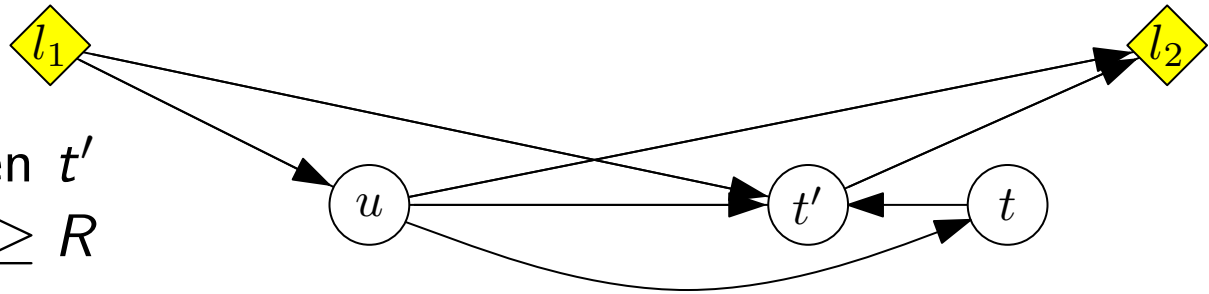
Proxy Knoten

Problem:

- REAL braucht Potential von jedem Knoten zu t und s
- $r(s)$ und/oder $r(t)$ könnten kleiner R sein
- keine Abstandswerte von den Landmakren zu s und t

Lösung:

- bestimme Proxy-Knoten t' für t (s analog), $r(t') \geq R$
- neue Ungleichungen:



$$d(u,t) \leq d(u,l_2) - d(t',l_2) - d(t,t')$$
$$d(u,t) \leq d(l_1,t') - d(l_1,u) - d(t,t')$$

Experimente

Eingaben:

- Straßennetzwerke
  - Europa: 18 Mio. Knoten, 42 Mio. Kanten
  - USA: 22 Mio. Knoten, 56 Mio. Kanten

Evaluation:

- Vorberechnung in Minuten und zusätzliche Bytes pro Knoten
- durchschnittlicher Suchraum (#abgearbeitete Knoten) und Suchzeiten (in *ms*) von 10 000 Zufallsanfragen

Zufallsanfragen: RE mit/ohne Shortcuts

Eingabe: BayArea: 330k Knoten

		Vorb.	Anfrage	
shortcuts	reach	time [min]	#settled	time
nein	approx	52	13369	6.44
nein	exakt	966	11194	6.05
ja	approx	3	1590	1.17
ja	exakt	980	1383	0.97

Beobachtung:

- shortcuts reduzieren Vorberechnungszeit
- beschleunigen Anfragen
- approximative Vorberechnung deutlich schneller
- Verlust in Anfragen nicht sehr groß

RE: Verfeinerung

USA	Vorb.	Anfrage	
δ	time [min]	#settled	time
0	32	2 555	2.00
12 235	36	2 448	1.84
24 469	44	2 317	1.81
48 938	67	2 159	1.70
97 876	148	2 086	1.66

Beobachtung:

- Verfeinerung bringt bis zu 15%
- aber zu hohe Werte erhöhen Vorberechnungszeit zu massiv

Zufallsanfragen: RE und REAL

	Vorberechnung		Anfrage		
landmarks	sparsity	Zeit [min]	Platz [byte/n]	Such raum	Zeit [ms]
0	–	44	15	2 317	1.81
16	1	64	104	675	1.06
16	4	64	39	689	1.34
16	16	64	22	730	1.31
16	64	64	17	888	1.38
64	4	121	113	493	1.02
64	16	121	43	540	0.98
64	64	121	25	743	1.02

Beobachtungen:

- Zuschalten von Landmarken zahlt sich aus
- Partielle Landmarkeninformationen reduziert Platzverbrauch

Übersicht: bisherige Techniken

	Vorberechnung		Anfrage		
	Zeit [h:m]	Platz [byte/n]	Such raum	Zeit [ms]	Beschl.
Dijkstra	0:00	0	9 114 385	5 591.6	1
ALT-16	1:25	128	74 669	53.6	104
ALT-64	1:08	512	25 324	19.6	285
Arc-Flags (128)	17:08	10	2 764	0.8	6 988
RE	1:22	13	4 643	3.5	1 597
REAL-(16,1)	1:36	81	814	1.2	4 588
REAL-(64,16)	2:20	35	679	1.1	5 037

Beobachtung:

- REAL ähnliche Performance wie Arc-Flags
- deutlich kürzere Vorberechnungszeiten
- höherer Speicherverbrauch

Zusammenfassung Reach

- prunen von Knoten
- auf Basis von Zentralitätswerten
- benötigt Abstand von s und t
- Abstand zu t durch Potential
- und/oder mit bidirektionalem Algorithmus
- Problem: Vorberechnung basiert auf APSP

Zusammenfassung RE/REAL

- Erweiterung von Reach
- Kanten-Reach
- Shortcuts
- iteratives Berechnen von Reach
- Verfeinerung
- REAL: Reach + ALT
- partielle Landmarken

Literatur

Reach:

- Gutman 2004 (ursprüngliche Idee, schlecht verständlich)
- Goldberg et al. 2006 (Erweiterungen)

Anmerkung:

- wird auf der Homepage verlinkt