Algorithmen für Routenplanung – Vorlesung 3

Daniel Delling

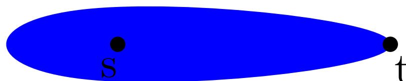
Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Letztes Mal

- bidirektionale Suche
- A^* Suche
- A^* mit Landmarken
- bidirektionale A^* Suche

Themen Heute

- Geometrische Container
- Arc-Flags



Ideensammlung

Wie Suche zielgerichtet machen?

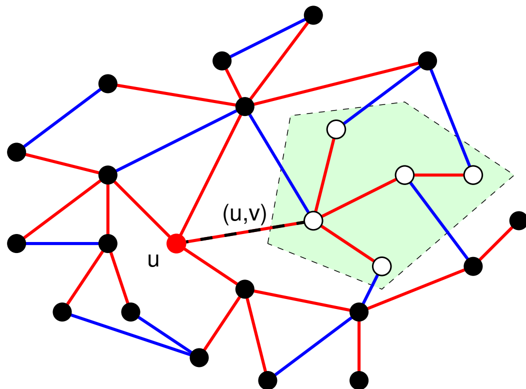
- prunen von Kanten, Knoten die in die “falsche” Richtung liegen
- Reihenfolge in der Knoten besucht werden ändern

heute ersteres

Geometrische Container

Beobachtung:

- subpfade von kürzesten Wege sind auch kürzeste Wege
- nicht jede Kante ist wichtig für ein bestimmtes Ziel



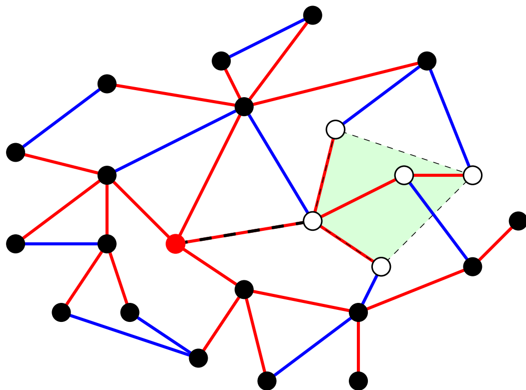
Idee:

- speicher geometrisches Objekt für jede Kante, das alle Knoten des Unterbaums beinhaltet
- relaxiere während Anfrage nur Kanten, für die Ziel t im Objekt ist

Pruning Dijkstra - Pseudocode

```
PruningDijkstra(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      for all edges e = (u, v) ∈ E do
6          if t ≠ C(e) then continue
7          if d[u] + len(e) < d[v] then
8              d[v] ← d[u] + len(e)
9              if v ∈ Q then Q.decreaseKey(v, d[v])
10             else Q.insert(v, d[v])
```

Formen



Viele Formen möglich:

- Winkel
- Winkel + Distanz
- Umgebenes Rechteck
- Konvexe Hülle

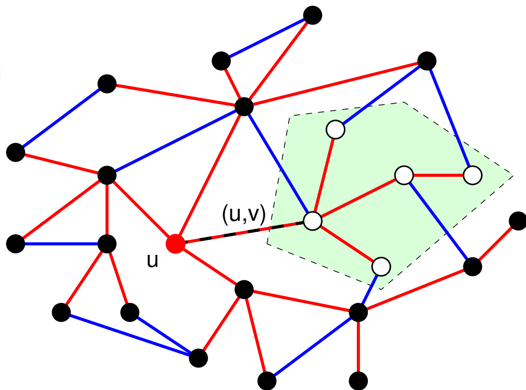
Trade-Off:

- Speicherplatz pro Kante
- Overhead zur Bestimmung ob t im Container liegt
- Umgebenes Rechteck scheint am besten zu sein

Vorberechnung

Vorgehen:

- für jeden Knoten einen kürzeste Wege Baum berechnen
- dann für jede Kante den minimalen Container berechnen
- speicher Container an der Kante



Zeit- und Platz-Bedarf:

- m Container, Größe abhängig von Komplexität des Containers
- n Dijkstras + m Berechnungen der Container

Diskussion

Vorteile:

- einfacher Anfrage-Algorithmus
- Beschleunigung um einen Faktor von bis zu Faktor 40
- Vorberechnung basiert auf Dijkstra Läufen

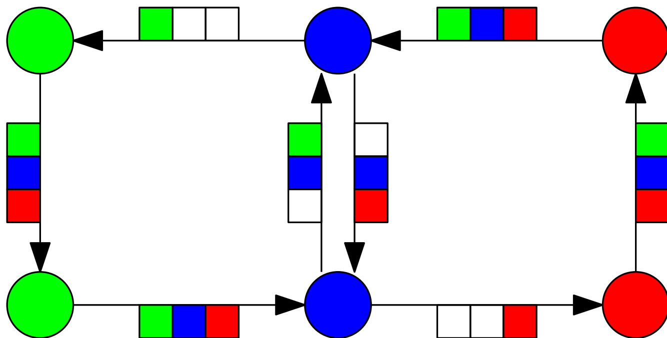
Nachteile:

- Laufzeit der Vorberechnung $O(\underbrace{m \cdot |C|}_{\text{Container}} + \underbrace{n(m + n \log n)}_{\text{all pair shortest path}})$
- daher nicht berechenbar auf sehr großen Netzen (ungefähr 500 Jahre für das Straßennetzwerk von Europa)
- Vorberechnungsplatz: m Container (Box: 2 Punkte)
- Einbettung der Graphen nötig
- Container können sehr ungenau sein

Arc-Flags

Idee:

- invertiere die Idee der Geometrischen Container
- partitioniere Graphen in k Zellen
- hänge Label mit k Bits an jede Kante
- gibt an, ob e für Ziele in Zielzelle T benötigt wird



Vorberechnung

Zwei Schritte:

- Partitionierung
- setze korrekte Flaggen

Partitionierung

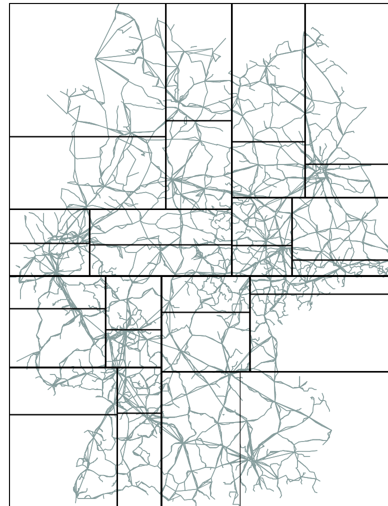
Anforderungen:

- balanciert

mögliche Partitionen:

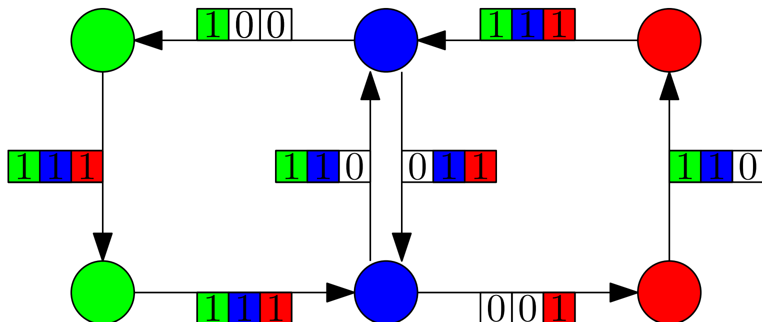
- Gitter
- Quad-Baum
  - iterativ in 4 Zellen unterteilen
- kd-Baum
  - Verallgemeinerung von Quad-Baum

weitere Anforderungen?



Flaggenberechnung: Erster Versuch

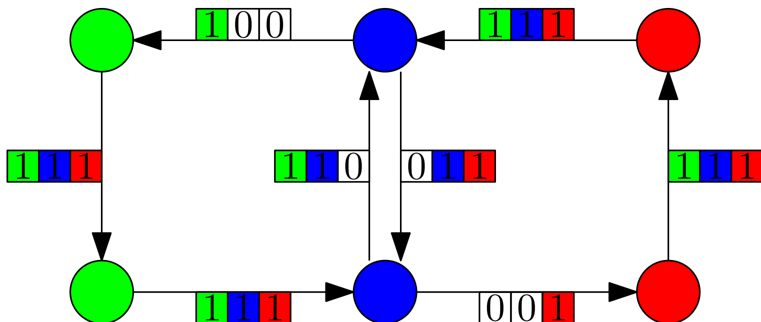
- von jedem Knoten
- konstruiere Rückwärts KW-Baum
- setze Flagge der Region der Wurzel für jede Baumkante
- wieder APSP und somit > 400 Jahre



Flaggenberechnung: Randknotenbäume

Beobachtung: Man muss durch Randknoten in die Zelle

- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn e Baumkante des Baums von b ist



Flaggenberechnung: zentralisierte Bäume

verallgemeinere Dijkstra:

- für jede Region r (mit Randknotenmenge B_r)
- hänge Label $l(u)$ der Größe $|B_r|$ an jeden Knoten u
- speichert den Abstand $d(u, b)$ für alle $b \in B_r$
- triviale Intialisierung der Randknoten, füge alle in Queue ein
- nehme Knoten u aus der Queue (key: $\min\{l(u)\}$)
- relaxiere Kanten (u, v):
  - erzeuge Label l durch $l(u) + len(u, v)$
  - checke ob l das Label $l(v)$ verbessert
- breche ab, wenn keine Verbesserungen mehr möglich sind
- setze Flagge auf `true` wenn $d(u, b) + len(u, v) = d(v, b)$ gilt

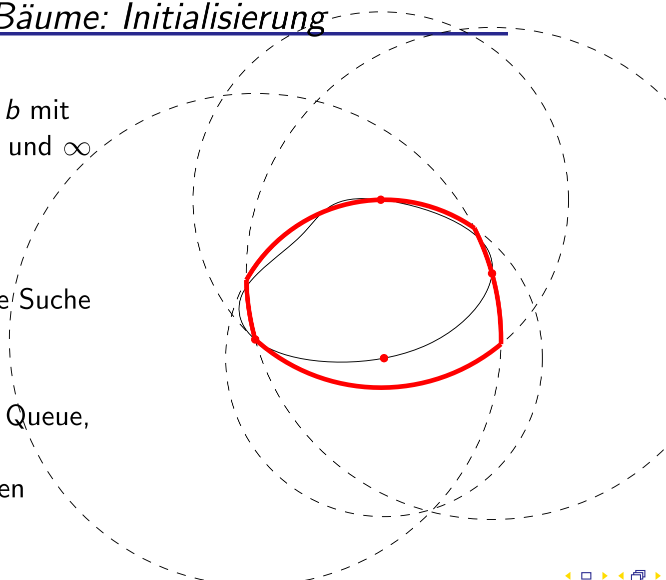
Zentralisierte Bäume: Initialisierung

trivial:

- jeder Randknoten b mit Abstand 0 zu sich und ∞ zu allen anderen

besser:

- von jedem b lokale Suche zu allen anderen Randknoten
- alle Knoten in die Queue, die auf Rand der Schnittmenge liegen



Zentralisierte Bäume

Bemerkungen:

- Knoten können mehrfach besucht werden (label-correcting)
- Prioritäten der Knoten beliebig wählbar
  - z.B. Minimum über alle Einträge von u oder auch nur der vorläufigen
  - hängt von Eingabe ab
- hoher Speicherverbrauch durch n Label der Größe $|B_r|$
  - 18 Mio. Knoten und 1000 Randknoten $\Rightarrow$ 72 GB
  - teile Arbeit in mehrere Schritte mit maximal 100 Randknoten
  - meist Beschleunigung um Faktor 4 gegenüber Randknotenansatz

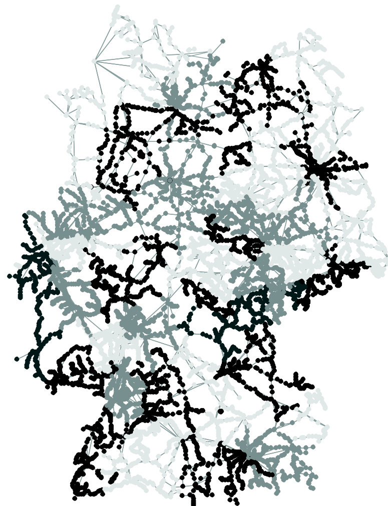
Partitionierung

Anforderungen:

- ausbalanciert
- wenige Randknoten
- zusammenhängend

Multi-Way Arc-Separator:

- benutzt keine Einbettung
- teilt rekursiv Graphen in k Teile mit kleinem Schnitt
- prädestiniert für Arc-Flags



Anfragen

```
ArcFlagDijkstra(G = (V, E), s, t)
1  T = region(t)
2  d[s] = 0
3  Q.clear(), Q.add(s, 0)
4  while !Q.empty() do
5      u ← Q.deleteMin()
6      for all edges e = (u, v) ∈ E do
7          if AF_T(e) = false then continue ;     // checking Arc-Flag
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])
```

Korrektheit

Korrektheit

`ArcFlagsDijsktra` ist korrekt

zu zeigen: für jeden kürzesten s–t Weg $P = (s = u_1, \ldots, u_k = t)$ haben alle Kanten (u_i, u_{i+1}) die Flagge $AF_T(u_i, u_{i+1})$ gesetzt.

- für alle (u_i, u_{i+1}) in T trivial
- sei u_j der letzte Knoten auf P in T
- u_j also Randknoten und somit Rückwärtsbaum gebaut von u_i (einzeln oder zentralisiert)
- $P' = (u_1, \ldots, u_j)$ ist Subpfad von P und somit ein kürzester Weg von u_1 nach u_j
- somit Baumkanten mit gesetzten Flaggen

Suchraum

Diskussion

Vorteile:

- einfacher Anfrage Algorithmus
- Vorberechnung basiert nicht auf APSP

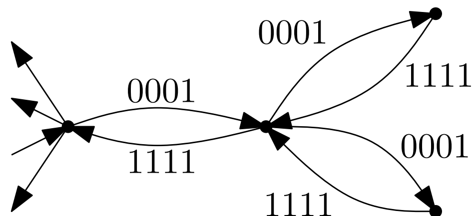
Nachteile:

- Randknotenansatz: lange Vorberechnung, ca. 3 Tage für Europa
- zentralisierter Ansatz:
  - hoher Speicherverbrauch währende Vorberechnung
  - moderate Dauer (aber immer noch ca. 18 Stunden für Europa)
- mehr Flaggen gesetzt, wenn nah an der Zielzelle
- alle Flaggen in Zielzelle gesetzt
- Anfragen in einer Zelle ohne Beschleunigung

Automatisches Setzen von Flaggen

Beobachtung:

- Für manche Kanten kann man die Flaggen automatisch setzen



Angehangene Bäume:

- Kanten zur Wurzel hin haben alle Flaggen gesetzt
- Kanten von Wurzel weg haben nur eine Flagge gesetzt
- also können die Bäume vor der Vorberechnung vom Graphen entfernt werden
- Knotenzahl verringert sich um 1 Drittel

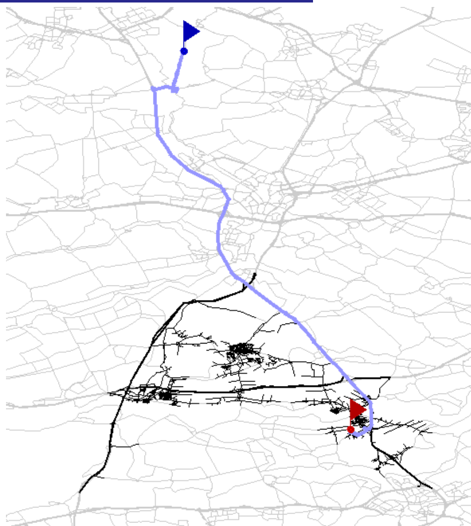
Coning-Effekt

Beobachtung:

- lange Zeit nur eine Kante wichtig
- daher immer nur ein Knoten in der Queue
- aber: je näher an der Zelle, desto mehr Kanten werden wichtig
- Suche fächert sich auf
- in Zelle werden dann alle Kanten relaxiert

Zwei Ansätze:

- bidirektionale Flags
- multi-level Flags



Bidirektionale Arc-Flags

Vorberechung:

- Vorwärts- und Rückwärtsflaggen
- Rückwärtsflaggen können analog für eingehende Kanten berechnet werden
- Vorberechungszeit in gerichteten Graphen erhöht sich um Faktor 2

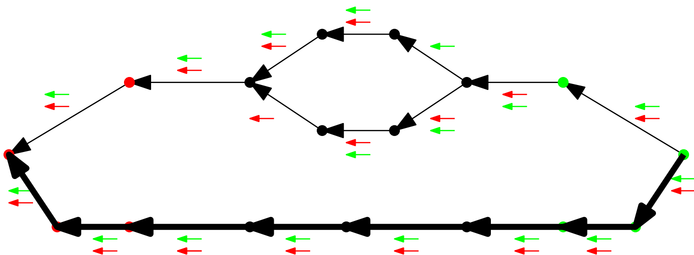
Anfrage:

- bidirektional:
  - Vorwärtssuche relaxiert nur Kanten mit Flagge für T
  - Rückwärtssuche nur Kanten mit Flaggen für S
- normales Stopp-Kriterium von bidirektionalem Dijkstra

Bidirektionale Arc-Flags

Problem:

- Eindeutigkeit der Wege
- eventuell nicht korrekt



Lösung:

- kommt in Straßengraphen kaum vor
- daher öffne Flaggen für alle möglichen Wege

Suchraum

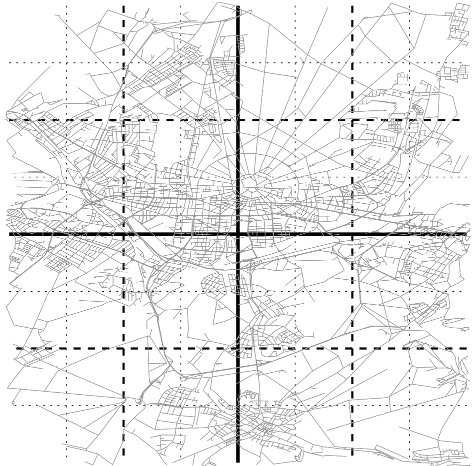
Multi-Level Arc-Flags

Problem:

- ≫ Anfragen in einer Zelle ohne Beschleunigung
- ≫ viele Real-Welt Anfragen sind lokal

Multi-Level Arc-Flags:

- ≫ Multi-Level Partition
- ≫ berechne partielle Flaggen



Multi-Level Arc-Flags

Vorberechnung:

- oberster Level wie gehabt
- auf unteren Leveln:
  - von jedem Randknoten führe Dijkstra aus, bis alle Knoten der Superzelle abgearbeitet worden sind
  - setze Flaggen nur in Superzelle
  - Hinweis: es reicht nicht, nur den Subgraphen der Superzelle zu betrachten (Übungsaufgabe)

Anfragen:

- bestimme gemeinsamen Level l von u und t
- werte Flaggen auf dem Level l aus

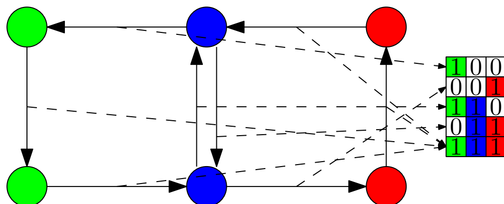
Korrektheit: siehe Übung

Suchraum

Datenstruktur

Effizient Flaggen speichern?

- ≫ pro Kante eine Flagge
- ≫ Beobachtung: Anzahl Kombinationen begrenzt



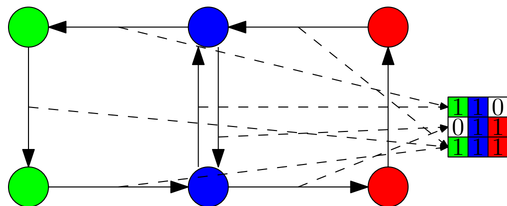
Idee:

- ≫ speicher Flaggen in Matrix
- ≫ Zeiger von Kanten auf die Matrix
- ≫ verringert Speicherverbrauch um einen Faktor 5

Flaggenkompression

Beobachtung:

- kippen eines Bits von 1 auf 0 verboten
- kippen eines Bits von 0 auf 1 erlaubt (weiterhin korrekt, eventuell langsamer)



Idee:

- verringere Anzahl eindeutiger Arc-Flags durch kippen
- dadurch Kompression der Matrix
- finde "gutes" Mapping (Studienarbeit WS 08/09)

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: Anzahl Regionen

	Prepro		Query		
regions	time [min]	space [B/n]	# settled nodes	time [ms]	spd up
0	0	0	9 114 385	5 591.6	1.0
200	1 028	19	2 369	1.6	3 494.8
400	1 366	20	1 868	1.2	4 659.7
600	1 723	21	1 700	1.1	5 083.3
800	1 892	23	1 642	1.4	3 994.0
1000	2 156	25	1 593	1.1	5 083.3

Beobachtungen:

- lange Vorberechnung
- hohe Beschleunigung
- geringer Speicherverbrauch
- mehr als 200 Regionen lohnt sich nicht

Dijkstra-Rank

Problem:

- Zufallsanfragen geben wenig Informationen
- Wie ist die Varianz?
- Werden nahe oder ferne Anfragen beschleunigt?

Idee:

- Dijkstra definiert für gegebenen Startknoten Ordnung für auf den Knoten
- Dijkstra-Rang $r_s(u)$ eines Knoten u für gegebenes s
- wähle 1000 Startknoten und analysiere jeweils die Suchzeiten um die Knoten mit Rang $2^1, \ldots, 2^{\log n}$ zu finden
- zeichne Plot

Lokale Anfragen Arc-Flags I

Dijkstra
Arc–Flags (128 regions)
bidirectional Arc–Flags (128 regions)

Query Time [ms]

10000
1000
100
10
3
1
0.1
0.01

2^{0} 2^{1} 2^{2} 2^{3} 2^{4} 2^{5} 2^{6} 2^{7} 2^{8} 2^{9} 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17} 2^{18} 2^{19} 2^{20} 2^{21} 2^{22} 2^{23} 2^{24}

Dijkstra Rank

Lokale Anfragen Arc-Flags I

bidirectional Dijkstra
bidirectional Arc–Flags (16 regions)
bidirectional Arc–Flags (128 regions)

Query Time [ms]

10000 1000 100 10 3 1 0.1 0.01

2^{0} 2^{1} 2^{2} 2^{3} 2^{4} 2^{5} 2^{6} 2^{7} 2^{8} 2^{9} 2^{10} 2^{11} 2^{12} 2^{13} 2^{14} 2^{15} 2^{16} 2^{17} 2^{18} 2^{19} 2^{20} 2^{21} 2^{22} 2^{23} 2^{24}

Dijkstra Rank

Beobachtungen

- birektionale Arc-Flags deutlich schneller als unidirektionale
- gegenüber Dijkstra nur Beschleunigung für weite Anfragen
- 128 Regionen deutlich besser 16 (bei 2^{24} nahezu gleich auf)

Flaggenkompression (Multi-Level, unidirektional)

Europagraph, Kostenfkt., Häufigkeitsfakt. 0,5

Laufzeit

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5

— 1_2_4_8_16_32
— 1_2_4_8_32_64
— 1_2_4_8_64_128

3.9(
0.518118

0e+00 2e+05 4e+05 6e+05 8e+05

#Entfernte Flags

Beobachtungen

- kaum Verlust bis zu 60% entfernte Flaggen
- geringer Verlust bis zu 80% entfernte Flaggen
- kippen von niedrig-leveligen Flaggen billiger

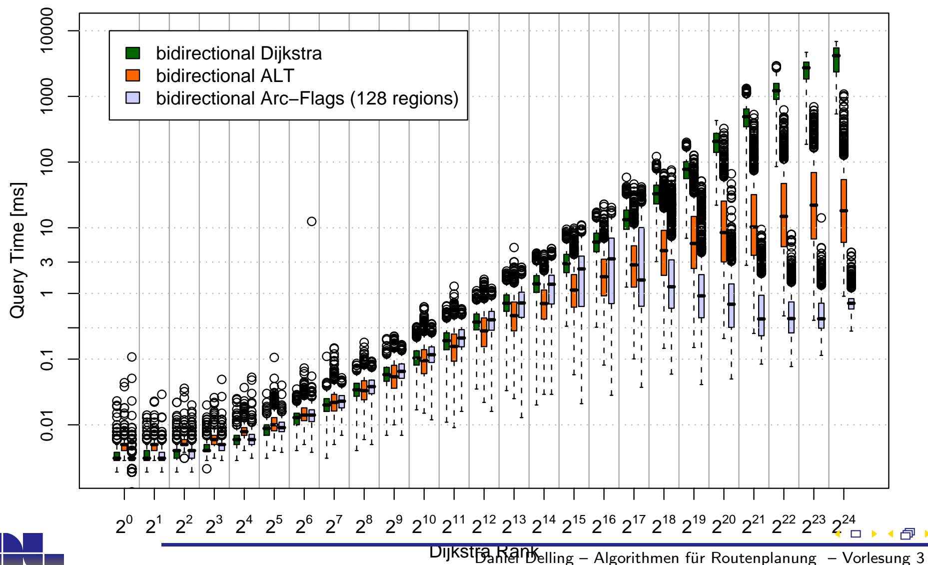
Übersicht: bisherige Techniken

	Vorberechnung		Anfrage		
	Zeit [h:m]	Platz [byte/n]	Such raum	Zeit [ms]	Beschl.
Dijkstra	0:00	0	9 114 385	5 591.6	1.0
Bi-Dijkstra	0:00	0	4 764 110	2 713.2	2.1
Uni-ALT-16	1:25	128	815 639	327.6	17.1
Uni-ALT-64	1:08	512	604 968	288.5	19.4
Bi-ALT-16	1:25	128	74 669	53.6	104.3
Bi-ALT-64	1:08	512	25 324	19.6	285.3
Uni Arc-Flags (128)	8:34	20	92 545	31.9	175.3
Bi Arc-Flags (128)	17:08	10	2 764	0.8	6 988.1

Beobachtung:

- ALT deutlich überlegen bei Anfragen und Platzverbrauch
- deutlich längere Vorberechnungszeiten

Lokale Anfragen Vergleich



Zusammenfassung Geometrische Container

- speichere pro Kante einen Container ab
- beinhaltet alle Knoten, die über diese Kante erreicht werden können
- einfacher Anfrage-Algorithmus
- Vorberechung basiert auf Dijkstra-Bäumen
- Beschleunigung von ca. 40
- Vorberechnung basiert auf APSP
- knapp 500 Jahre Vorberechnung für Europa
- nicht praktikabel

Zusammenfassung Arc-Flags

- invertiert Geometrische Container
- teile Graphen in k Regionen
- Flaggen zeigen an, ob Kante wichtig für Zielregion ist
- einfacher Anfrage-Algorithmus
- Vorberechnung
  - basiert nicht auf APSP
  - Dijkstra Baum von jedem Randknoten
  - manche Flaggen können automatisch gesetzt werden
  - daher schneller als bei Geometrischen Container
- bidirektional und multi-level Erweiterungen
- Beschleunigung von bis zu 7000
- nahe Anfragen nicht schneller als Dijkstra
- Vorberechnung dauert allerdings immer noch ca. 1 Tag

Literatur

Geometrische Container:

- Wagner, Willhalm, Zaroliagis 2005

Arc-Flags:

- Hilger, Köhler, Möhring, Schilling 2009

Anmerkung:

- wird auf der Homepage verlinkt