Karlsruher Institut für Technologie

Algorithmen für Routenplanung

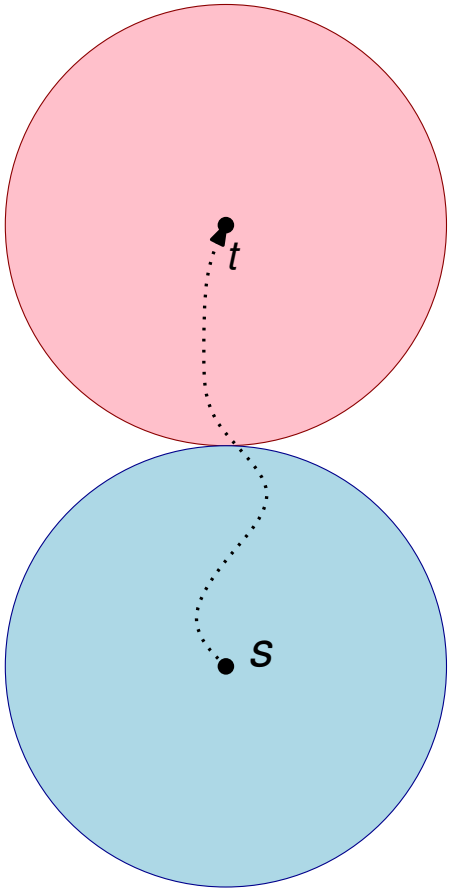
3. Sitzung, Sommersemester 2010

Thomas Pajor | 30. April 2010

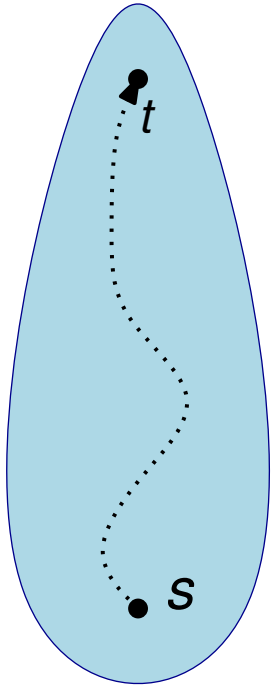
Institut für Theoretische Informatik · Algorithmik I · Prof. Dr. Dorothea Wagner

Letztes Mal

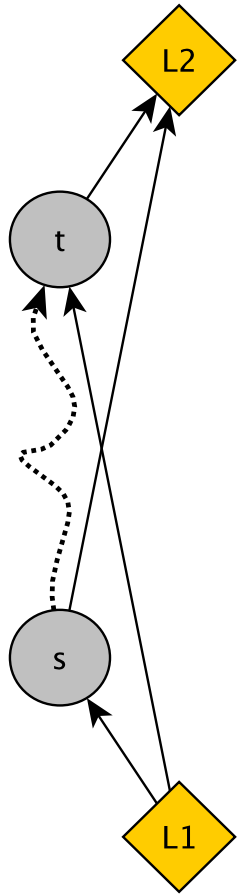
Bidirektionale Suche



A*



Landmarken



Wiederholung

A*-Algorithmus

- Benutze Potentialfunktion $\pi : V \to \mathbb{R}$ mit

$$\text{len}(u, v) - \pi(u) + \pi(v) \geq 0 \qquad \forall (u, v) \in E$$

- Keys in der Priority-Queue sind $d[u] + \pi(u)$ für alle $u \in V$
- Wenn $\pi(t) = 0$, dann $\pi(u) \leq d(u, t)$
- Je besser untere Schranke für $d(u, t)$ desto besser die Zielrichtung

A* $\leftrightarrow$ Dijkstra

- Definiere $G_\pi = (V, E, \text{len}_\pi)$ mit

$$\text{len}_\pi(u, v) := \text{len}(u, v) - \pi(u) + \pi(v)$$

- Dann: Dijkstra auf G_π entspricht A* auf G

Wiederholung

A*-Algorithmus

- Benutze Potentialfunktion $\pi : V \to \mathbb{R}$ mit

$$\text{len}(u, v) - \pi(u) + \pi(v) \geq 0 \qquad \forall (u, v) \in E$$

- Keys in der Priority-Queue sind $d[u] + \pi(u)$ für alle $u \in V$
- Wenn $\pi(t) = 0$, dann $\pi(u) \leq d(u, t)$
- Je besser untere Schranke für $d(u, t)$ desto besser die Zielrichtung

A* $\leftrightarrow$ Dijkstra

- Definiere $G_\pi = (V, E, \text{len}_\pi)$ mit

$$\text{len}_\pi(u, v) := \text{len}(u, v) - \pi(u) + \pi(v)$$

- Dann: Dijkstra auf G_π entspricht A* auf G

Wiederholung

Frage: Was ist eine gute Potentialfunktion für A*?

Idee: Benutze Landmarken und die Dreiecksungleichung

Vorberechnung:

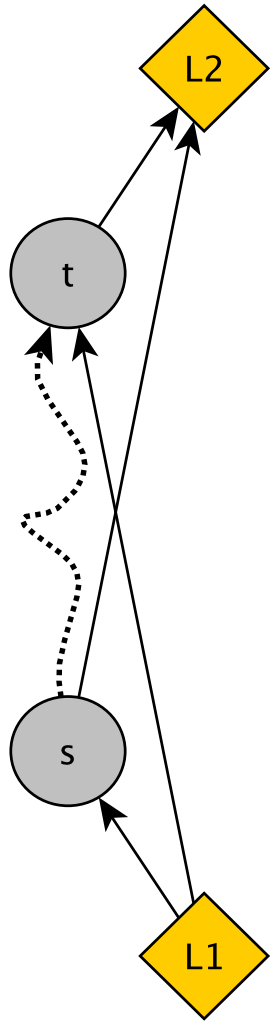
- Wähle $L \subseteq V$ ($L \approx 16$) Landmarken
- Berechne Distanz von und zu allen Landmarken

Anfrage:

- Benutze

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

als gültiges Potential



Wiederholung

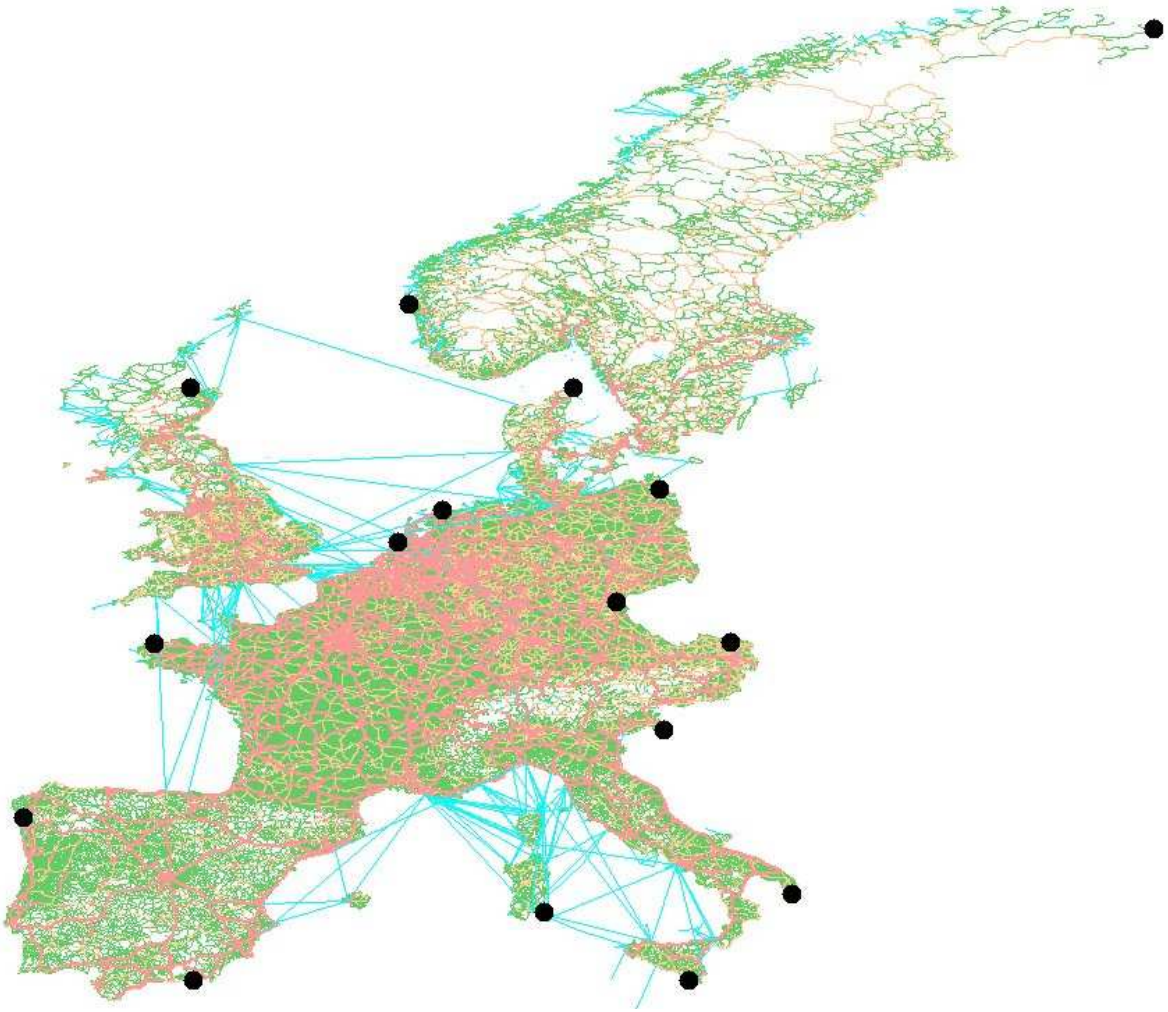
Frage: Was sind gute Landmarken?

Beobachtungen:
Reduzierte Kosten $\text{len}_\pi(u, v) = 0$ für Kanten, die

1. auf gemeinsamen kürzesten Weg zu t und zu Landmarke ℓ
2. auf gemeinsamen kürzesten Weg zu t und von Landmarke ℓ zu t

Intuition: Landmarken möglichst am “Rand” von G

Beispiel gute Landmarken



Berechnung Landmarken

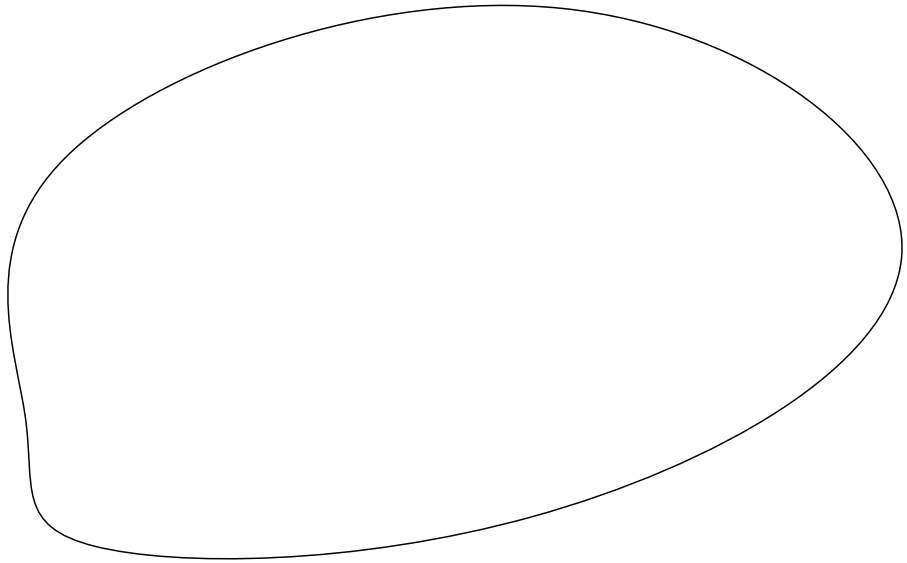
mehrere Ansätze:

- brute force: $\mathcal{O}(n^{|L|} \cdot \underbrace{n(m + n \log n)}_{\text{all pairs shortest path}})$
  - $+$ höchste Beschleunigung
  - $-$ zu lange Vorberechnung
- wähle zufällig
  - $+$ schnellste Vorberechnung
  - $-$ schlechte Beschleunigung
- mehrere Heuristiken, die versuchen den Rand zu finden
  - planar
  - farthest
  - avoid
  - lokale Optimierung (maxCover)

Planar-Landmarken

Vorgehen:

- suche Mittelpunkt c des Graphen
- teile Graphen in k Teile
- in jedem Teil wähle Knoten mit maximalen Abstand zu c als Landmarke



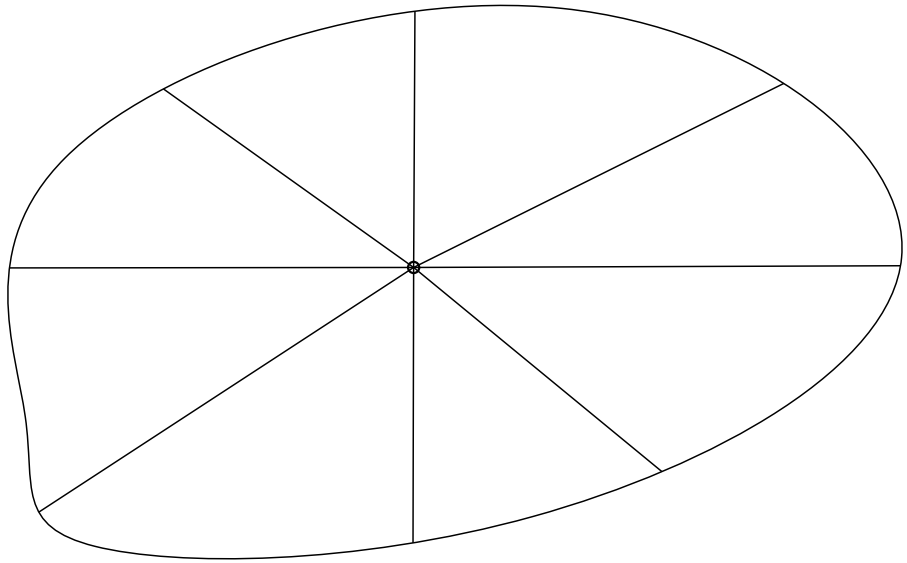
Anmerkungen:

- benötigt planare Einbettung
- liefert erstaunlich schlechte Ergebnisse

Planar-Landmarken

Vorgehen:

- suche Mittelpunkt *c* des Graphen
- teile Graphen in *k* Teile
- in jedem Teil wähle Knoten mit maximalen Abstand zu *c* als Landmarke



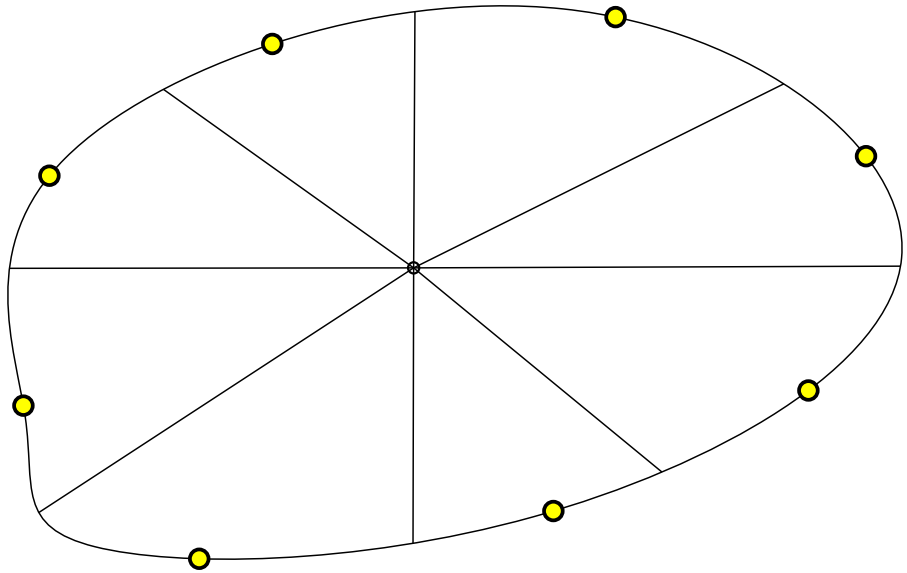
Anmerkungen:

- benötigt planare Einbettung
- liefert erstaunlich schlechte Ergebnisse

Planar-Landmarken

Vorgehen:

- suche Mittelpunkt *c* des Graphen
- teile Graphen in *k* Teile
- in jedem Teil wähle Knoten mit maximalen Abstand zu *c* als Landmarke



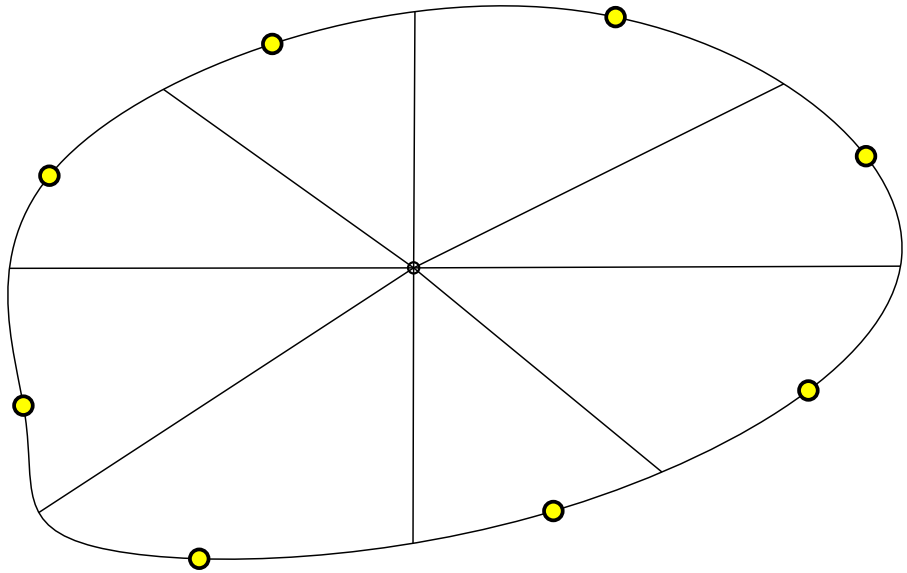
Anmerkungen:

- benötigt planare Einbettung
- liefert erstaunlich schlechte Ergebnisse

Planar-Landmarken

Vorgehen:

- suche Mittelpunkt *c* des Graphen
- teile Graphen in *k* Teile
- in jedem Teil wähle Knoten mit maximalen Abstand zu *c* als Landmarke



Anmerkungen:

- benötigt planare Einbettung
- liefert erstaunlich schlechte Ergebnisse

Farthest-Landmarken

FARTHEST-LANDMARKS(G,k)

```
1 L ← ∅
2 while |L| < k do
3     if |L| = 0 then DIJKSTRA(G,RANDOMNODE)
4     else DIJKSTRA(G,L)
5     u ← last settled node
6     L ← L ∪ {u}
```

Anmerkungen:

- Multi-Startknoten Dijkstra
- schlecht für kleine k
- erste Landmarke schlecht
- weitere Landmarken massiv abhängig von erster

Avoid-Landmarken

Vorgehen:

- berechne kürzeste Wege Baum T_r von einem zufälligen Knoten r
- weight$(u) = d(u, r) - \underline{d(u, r)}$ wobei $\underline{d(u, r)}$ durch L induziert
- size(u) Summe der Gewichte (weight) seiner Nachfolger in T_r
- size$(u) = 0$ wenn Teilbaum T_u von T_r eine Landmarke enthält
- w sei der Knoten mit maximaler Größe (size)
- traversiere T_r startend von w, folge immer dem Knoten mit maximaler Größe
- das erreichte Blatt wird zu L hinzugefügt

Anmerkungen:

- Identifiziere Regionen die schlecht von aktuellem L überdeckt
- Verfeinerung von Farthest Strategie

Optimierung von Landmarken

Problem:

- konstruktive Heuristik
- anfangs gewähle Landmarken eventuell suboptimal

Idee:

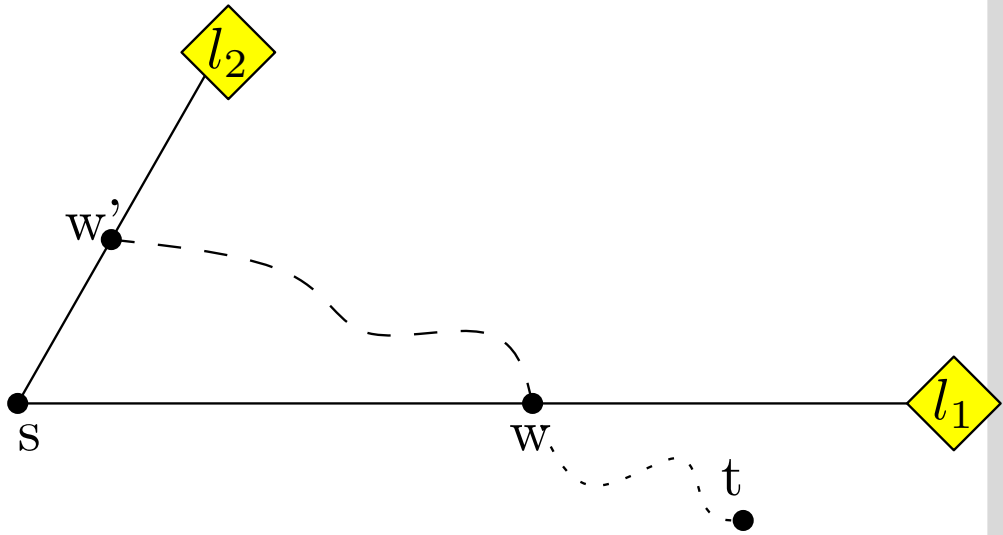
- lokale Optimierung
- berechne mehr Landmarken als nötig ($\approx 4\,000$)
- wähle beste durch Optimierungsfunktion, z. B.
  (1) maximiere Anzahl überdeckter Kanten,
  d. h. $\text{len}(u, v) - \pi(u) + \pi(v) = 0$
  (2) maximiere $\pi(s)$ für 1 Mio. s-t-Paare (simuliert Anfragen)

Avoid mit Funktion (1) wird *maxCover* genannt.

Aktive Landmarken

Problem:

- viele Kanten die zu einer Landmarke führen haben reduziertes Gewicht 0
- daher: Landmarken können Suche in die falsche Richtung ziehen



Lösung:

- wähle während Initialisierung eine Teilmenge von L als aktiv
- diese, für die $\pi(s)$ maximal sind
- die Landmarken liefern die besten Schranken und sind (hoffentlich) die besten
- Aktualisiere die aktiven Landmarken während der Suche

Datenstruktur

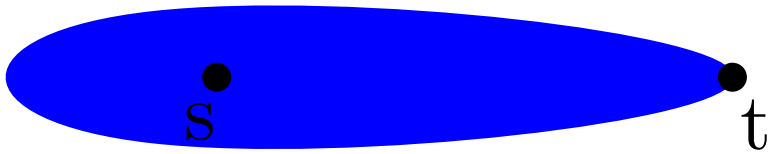
Erster Ansatz:

- $2 \cdot |L|$ 32-bit Vektoren der Größe n
- Problem: viele Cache-Misses

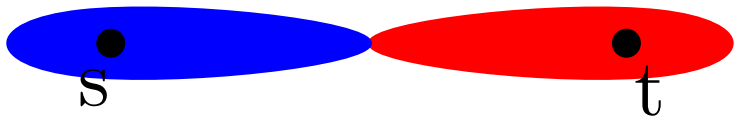
Besser:

- 1 64-bit Vektor der Größe $|L| \cdot n$
- speicher Distanz von und zu Landmarke in einem 64-bit Integer
- Zugriff auf Knoten mit id k und Landmarken-Nummer l in Segment $l \cdot k + l$
- dadurch deutlich erhöhte Lokalität
- beschleunigt die Anfragezeit um ca. einen Faktor von 4

Bidirektionaler A*



Bidirektionaler A*



Bidirektionaler A*

Erster Ansatz:

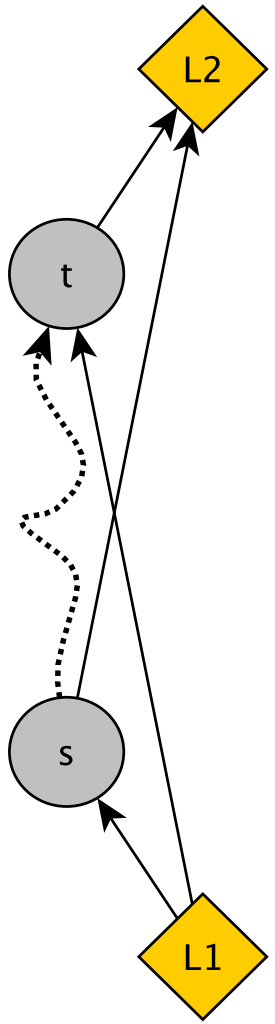
- benutze Vorwärtspot. π_f und Rückwärtspot. π_b

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

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

Problem:

- Suchen operieren auf unterschiedlichen Längenfunktionen
- konservatives Abbruchkriterium: Stoppe erst, wenn $\text{minKey}(\overrightarrow{Q}) > \mu$ oder $\text{minKey}(\overleftarrow{Q}) > \mu$



Bidirektionaler A*

Erster Ansatz:

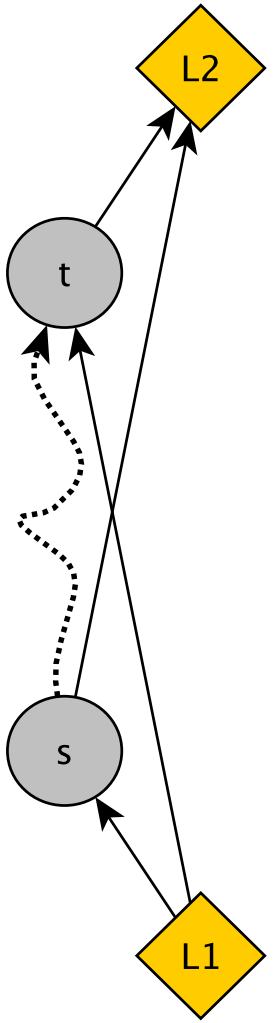
- benutze Vorwärtspot. π_f und Rückwärtspot. π_b

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

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

Problem:

- Suchen operieren auf unterschiedlichen Längenfunktionen
- konservatives Abbruchkriterium: Stoppe erst, wenn minKey($\overrightarrow{Q}$) $> \mu$ oder minKey($\overleftarrow{Q}$) $> \mu$



Bidirektionaler A*

Zweiter Ansatz:

- Wann operieren Suchen auf dem gleichen Graphen?
- wenn

$$\begin{aligned} \text{len}_{\pi_b}(v,u) &= \text{len}\,\pi_f(u,v) \\ \text{len}(u,v) - \pi_b(v) + \pi_b(u) &= \text{len}(u,v) - \pi_f(u) + \pi_f(v) \\ -\pi_b(v) + \pi_b(u) &= -\pi_f(u) + \pi_f(v) \\ \pi_b + \pi_f &= \text{const.} \end{aligned}$$

Bidirektionaler A*

Idee:

- nehme Kombination aus π_f und π_b

$$p_f = \frac{\pi_f - \pi_b}{2} \qquad p_b = \frac{\pi_b - \pi_f}{2} = -p_f$$

Somit

- wie bidirektionaler Dijkstra
- aber mit $d(s,u) + p_f(u)$ und $d(v,t) + p_b(v)$ als Keys
- stoppe wenn

$$\text{minKey}(\overrightarrow{Q}) + \text{minKey}(\overleftarrow{Q}) > \mu_{p_f} = \mu + p_f(s) - p_f(t)$$

- dadurch bidirektional zielgerichtet

ALT [Goldberg et al. 04,05]

- **A***, **L**andmarken, **T**riangle inequality (Dreiecksungleichung)
- bidirektionaler Landmarken-A* (2. Ansatz)
- aktive Landmarken
- Pruning
- meist 16 Landmarken
- meist avoid oder maxCover Landmarken

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 ALT

		Prepro		Query Unidir.			Query Bidir.		
	algorithm	time [min]	space [B/n]	# settled nodes	time [ms]	spd up	# settled nodes	time [ms]	spd up
EUR	Dijkstra	0	0	9 114 385	5 591.6	1.0	4 764 110	2 713.2	2.1
	ALT-4	12.1	32	1 289 070	469.1	11.9	355 442	254.1	22.0
	ALT-8	26.1	64	1 019 843	391.6	14.3	163 776	127.8	43.8
	ALT-16	851	128	815 639	327.6	17.1	74 669	53.6	104.3
	ALT-24	145.2	192	742 958	303.7	18.4	56 338	44.2	126.5
	ALT-32	27.1	256	683 566	301.4	18.6	40 945	29.4	190.2
	ALT-64	68.2	512	604 968	288.5	19.4	25 324	19.6	285.3
USA	Dijkstra	0	0	11 847 523	6 780.7	1.0	7 345 846	3 751.4	1.8
	ALT-8	44.5	64	922 897	329.8	20.6	328 140	219.6	30.9
	ALT-16	103.2	128	762 390	308.6	22.0	180 804	129.3	52.4
	ALT-32	35.8	256	628 841	291.6	23.3	109 727	79.5	85.3
	ALT-64	92.9	512	520 710	268.8	25.2	68 861	48.9	138.7

Beobachtungen

- unidirektionaler ALT: mehr als 16 Landmarken nicht sinnvoll
- bidirektionaler ALT: verdoppelung der Landmarken halbiert den Suchraum (ungefähr)
- 16 Landmarken: Beschleunigung $\approx$ 100 für Europa
- 64 Landmarken: Beschleunigung $\approx$ 300 für Europa
- hoher Speicherverbrauch
  - Graph-Datenstruktur: 424 MB
  - pro Landmarke: 144 MB in Europa
- USA schlechter als Europa (Vermutung: schlechtere Hierarchie)

Dijkstra-Rank

Problem:

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

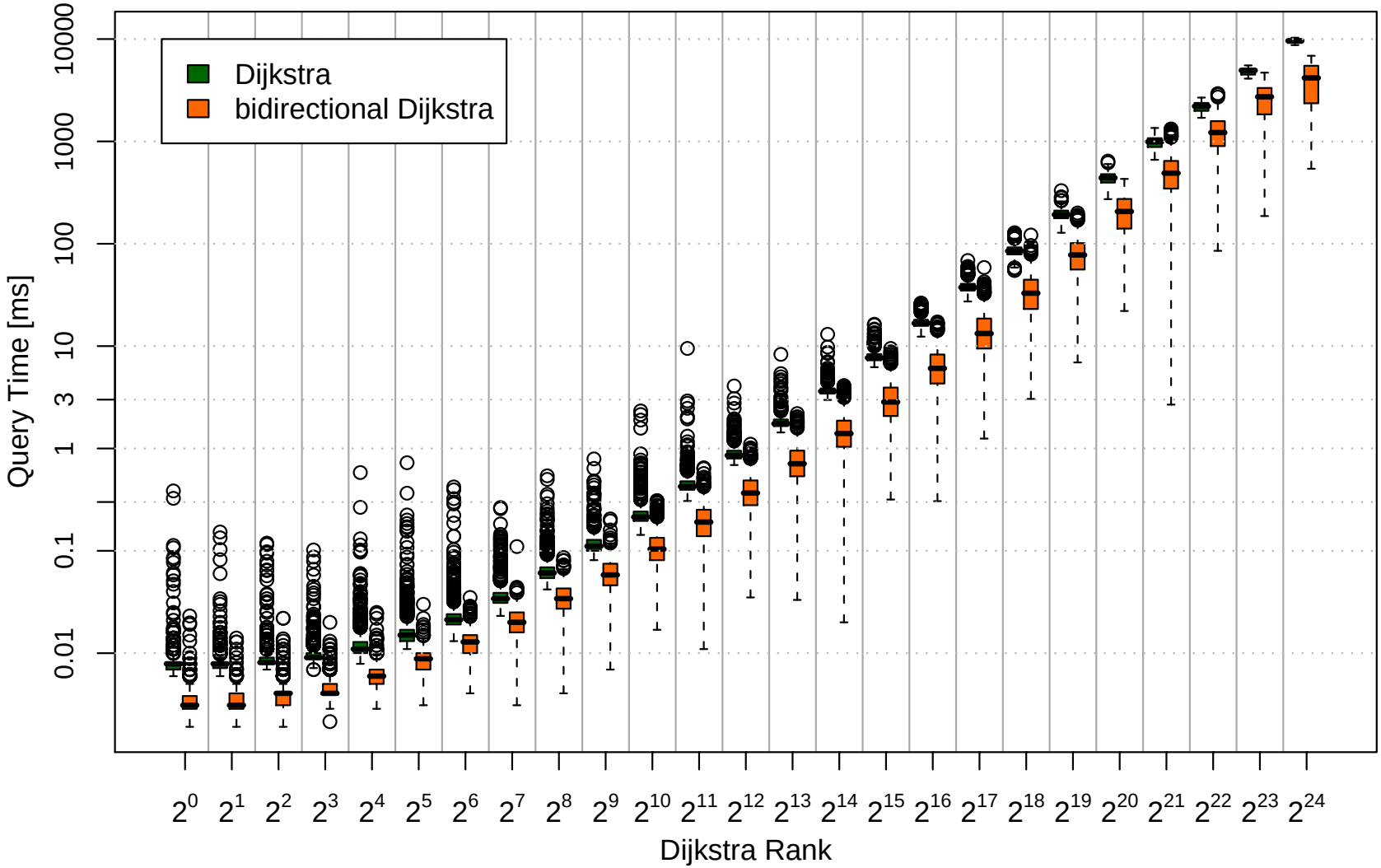
Idee:

- DIJKSTRA definiert für gegebenen Startknoten s Ordnung $\prec_s$ auf den Knoten durch

$$u \prec_s v :\Leftrightarrow d[u] \leq d[v]$$

- DIJKSTRA-Rang $r_s(u)$ eines Knoten u für gegebenes s: Index von u bezüglich $\prec_s$
- wähle 1000 Startknoten und analysiere jeweils die Suchzeiten um die Knoten mit Rang $2^1, \ldots, 2^{\log n}$ zu finden

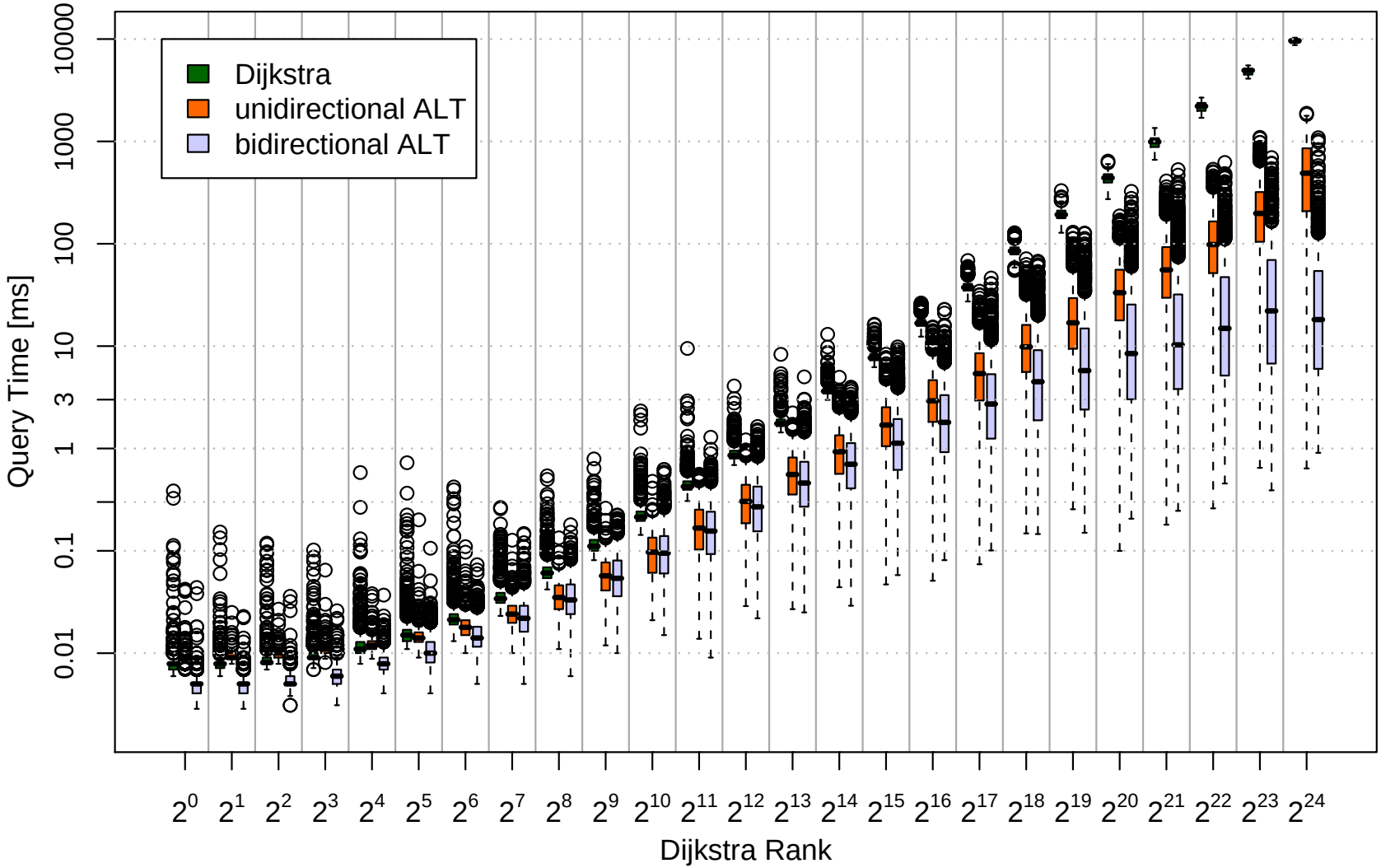
Lokale Anfragen: Bidir. Suche



Beobachtungen

- Ausreißer bei nahen Anfragen (vor allem unidirektional)
- Beschleunigung unabhängig vom Rang (immer ca. Faktor 2)
- Varianz etwas höher als bei unidirektionaler Suche
- manche Anfragen sind sehr schnell

Lokale Anfragen: ALT



Beobachtungen

- Beschleunigung steigt mit Rang
- kaum Beschleunigung für nahe Anfragen
- hohe Varianz für ALT
- Ausreißer bis zu Faktor 100 langsamer als Median

Ideensammlung

Wie Suche zielgerichtet machen?

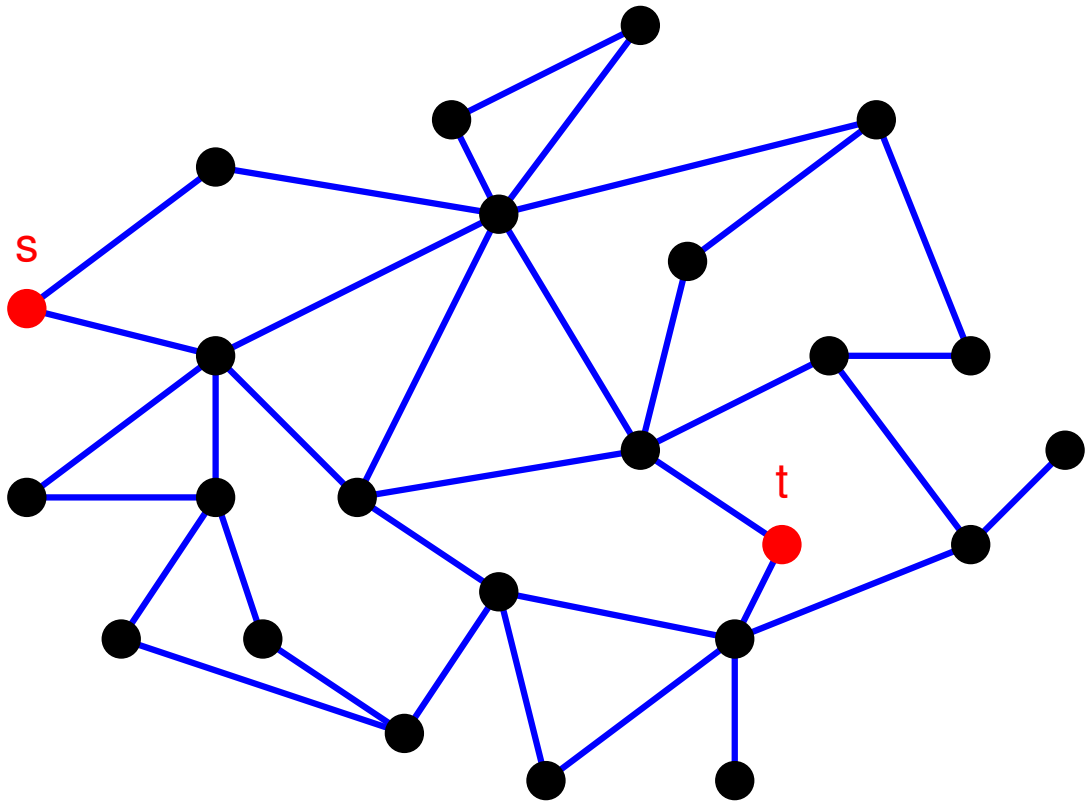
- prunen von Kanten, Knoten die in die “falsche” Richtung liegen
- Reihenfolge in der Knoten besucht werden ändern

Jetzt: ersteres

Geometrische Container

Beobachtung:

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



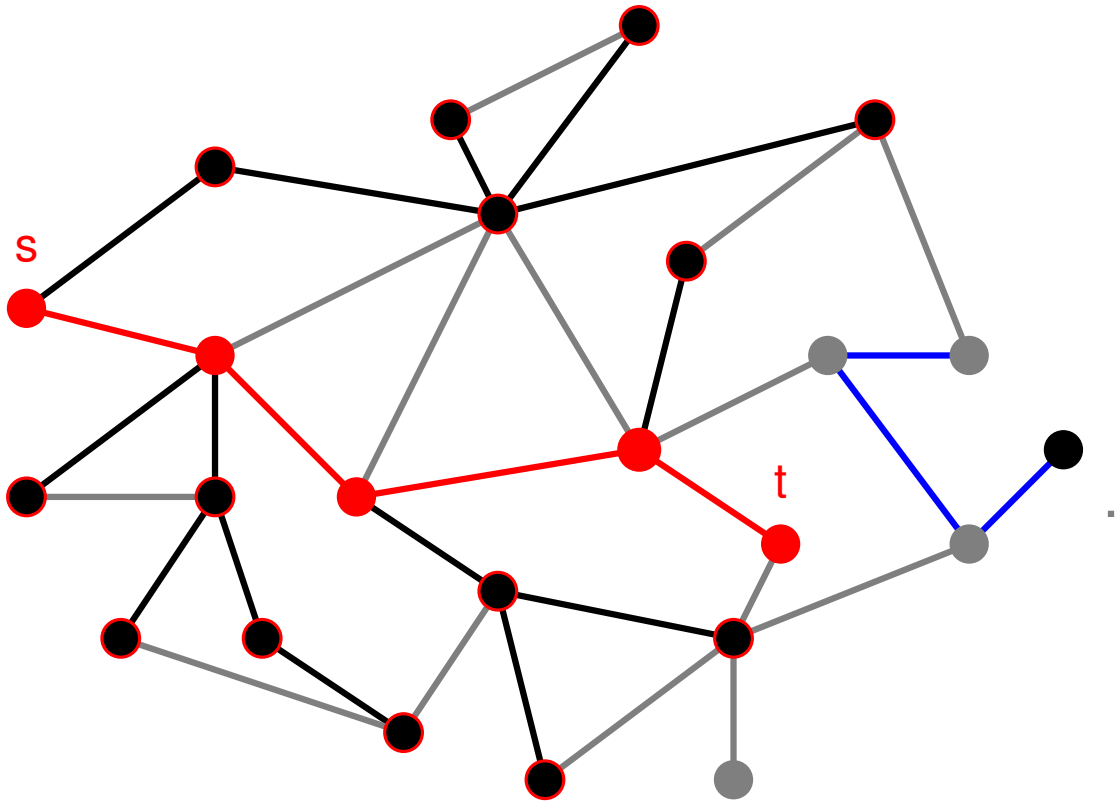
Idee:

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

Geometrische Container

Beobachtung:

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



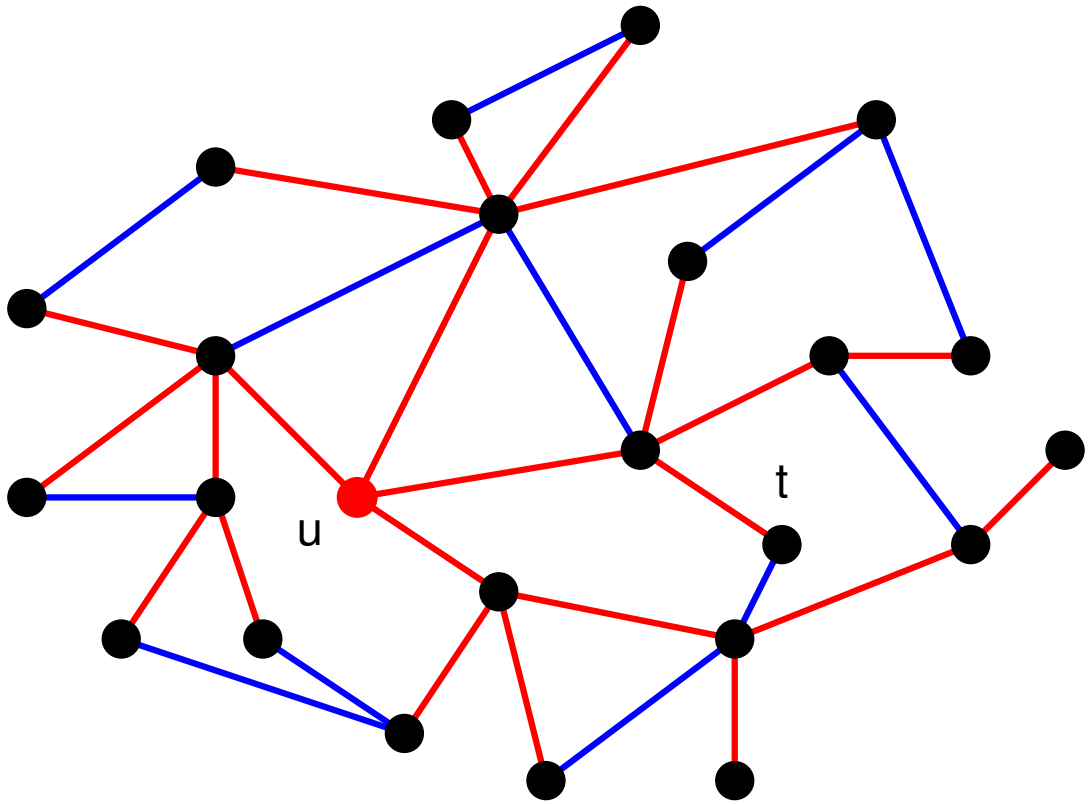
Idee:

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

Geometrische Container

Beobachtung:

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



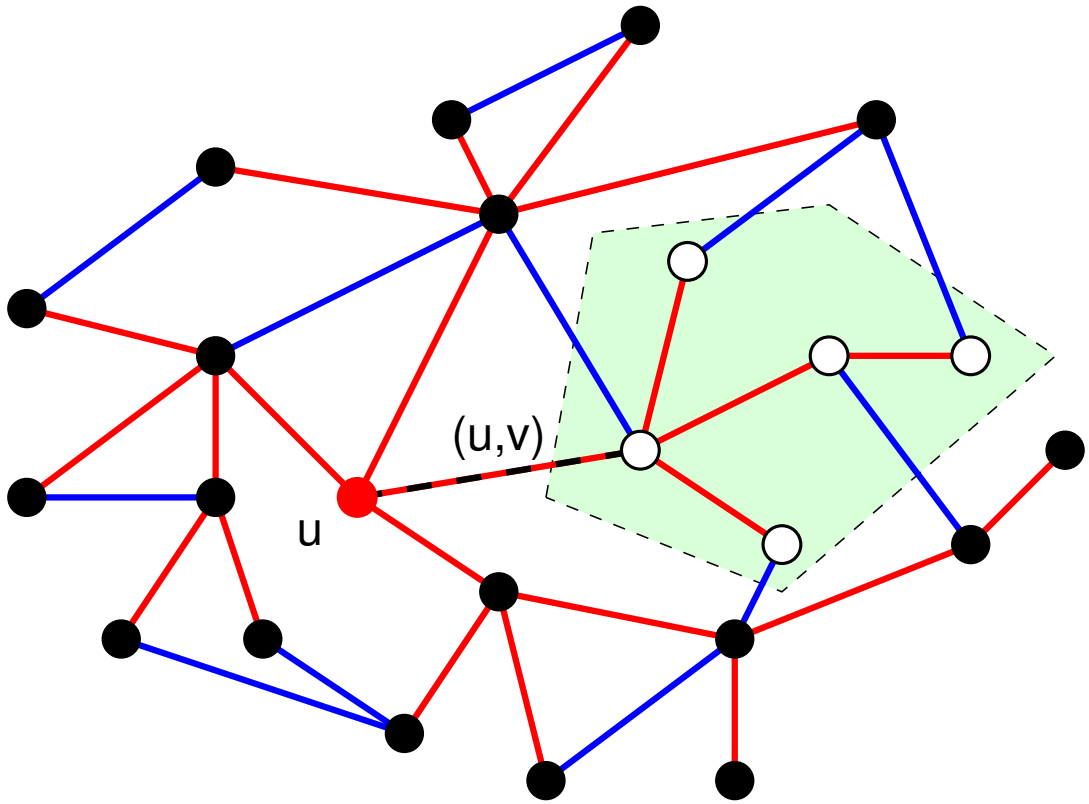
Idee:

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

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

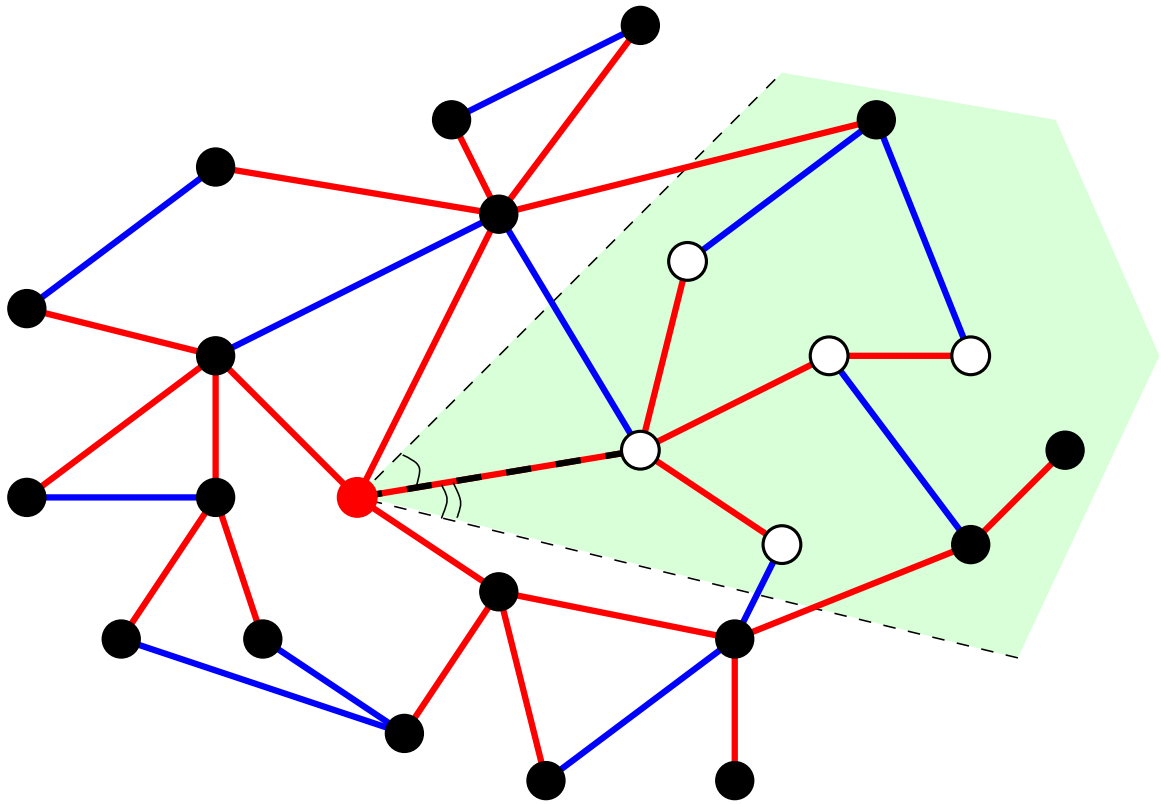
DIJKSTRA($G = (V, E)$, s)

```
1  d[s] = 0
2  Q.clear(), Q.add(s, 0)
3  while !Q.empty() do
4      u ← Q.deleteMin()
5      forall 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



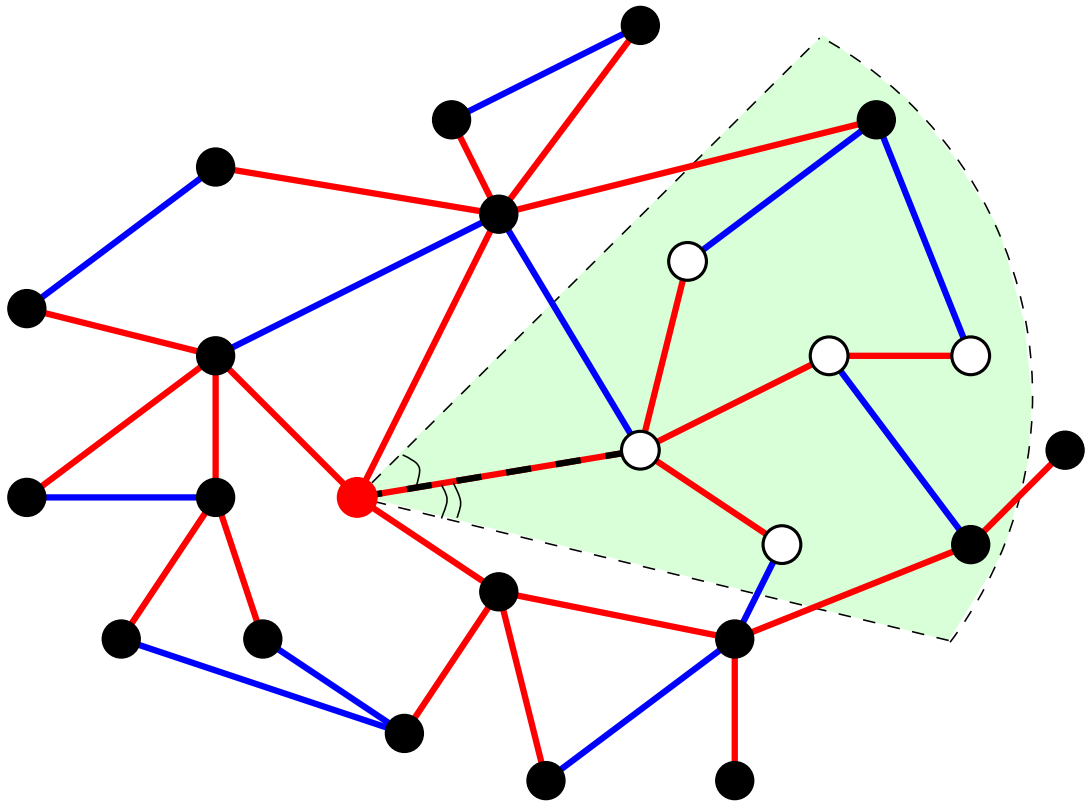
Trade-Off:

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

Formen

Viele Formen möglich:

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



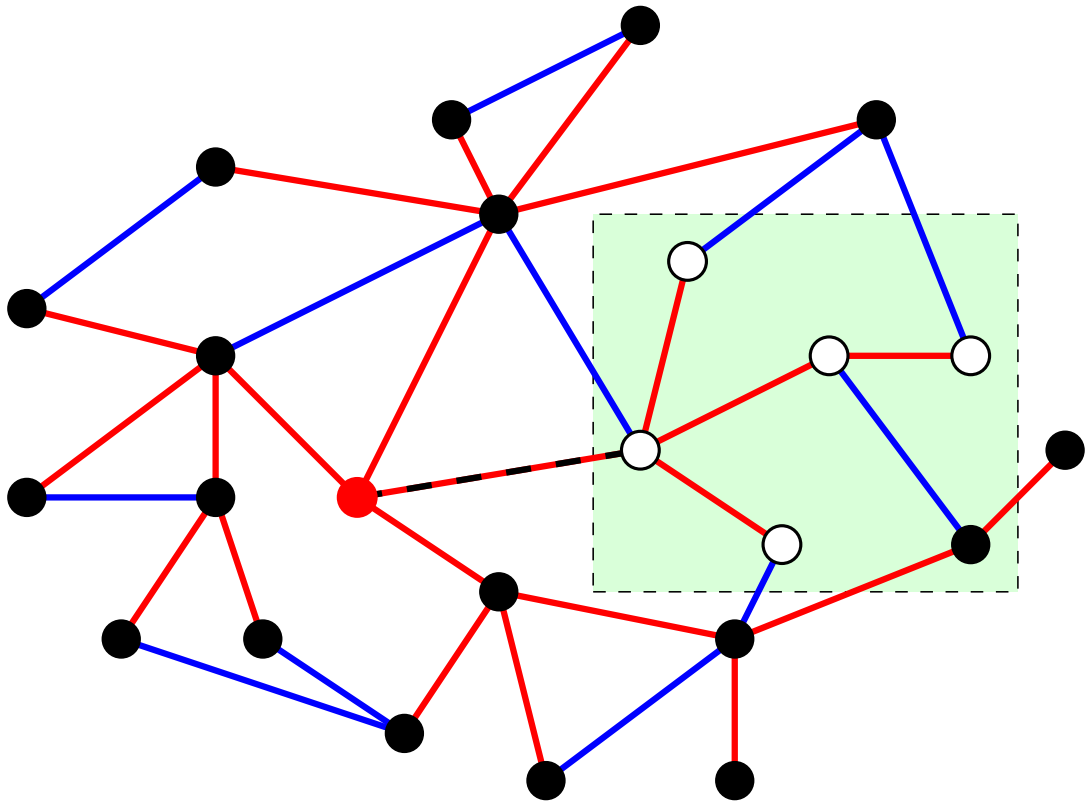
Trade-Off:

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

Formen

Viele Formen möglich:

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



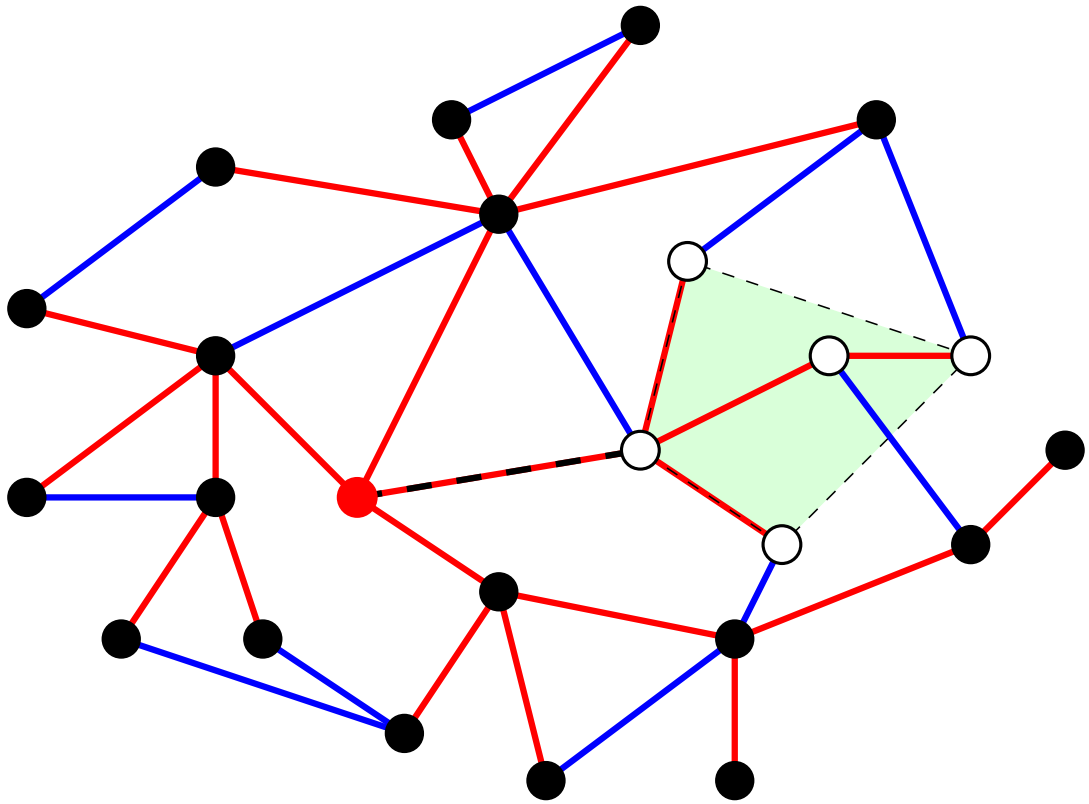
Trade-Off:

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

Formen

Viele Formen möglich:

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



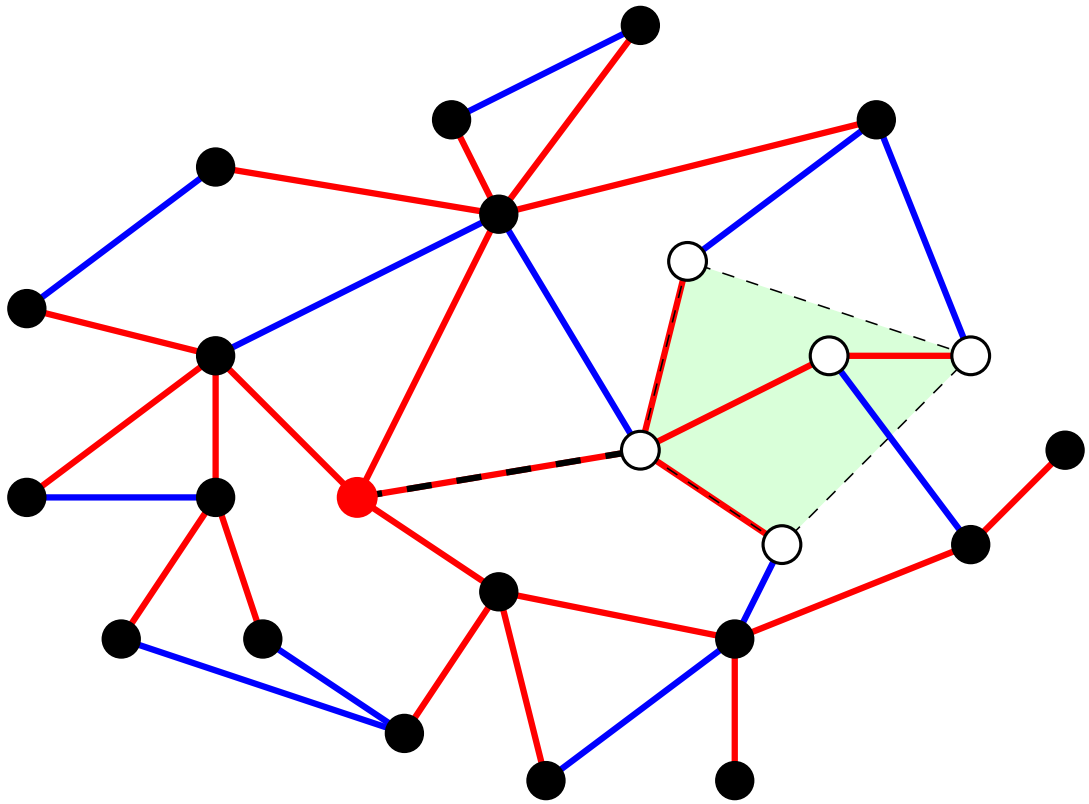
Trade-Off:

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

Formen

Viele Formen möglich:

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



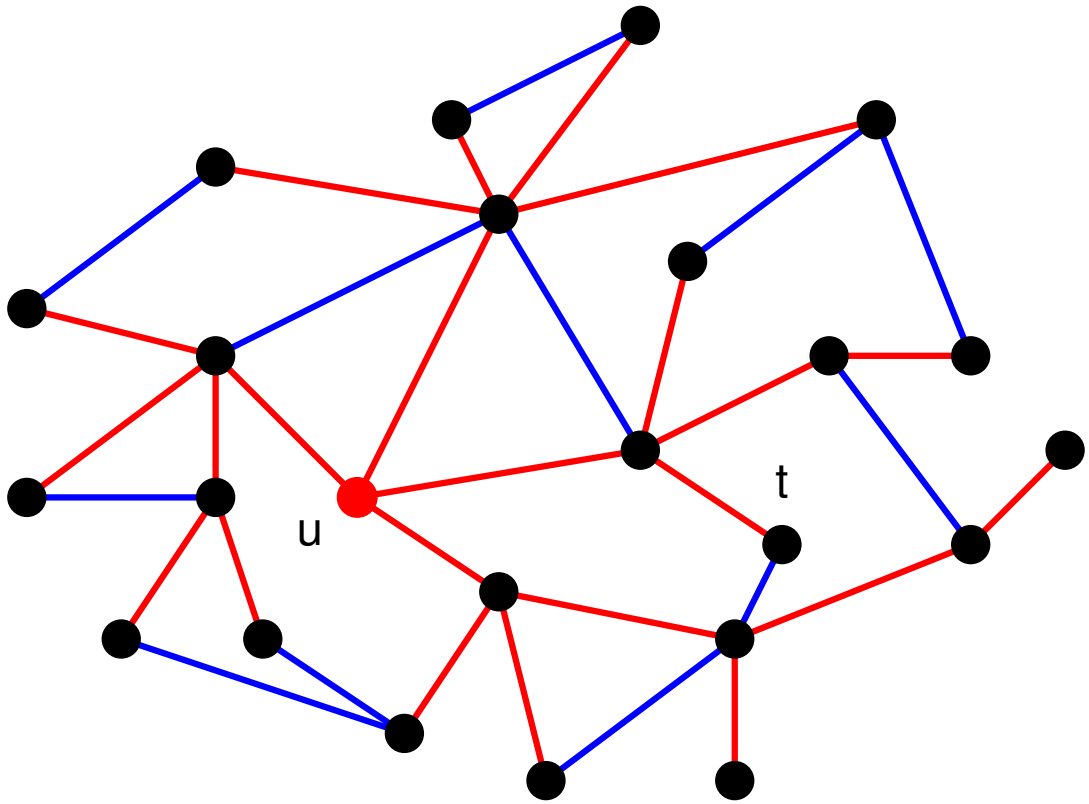
Trade-Off:

- 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



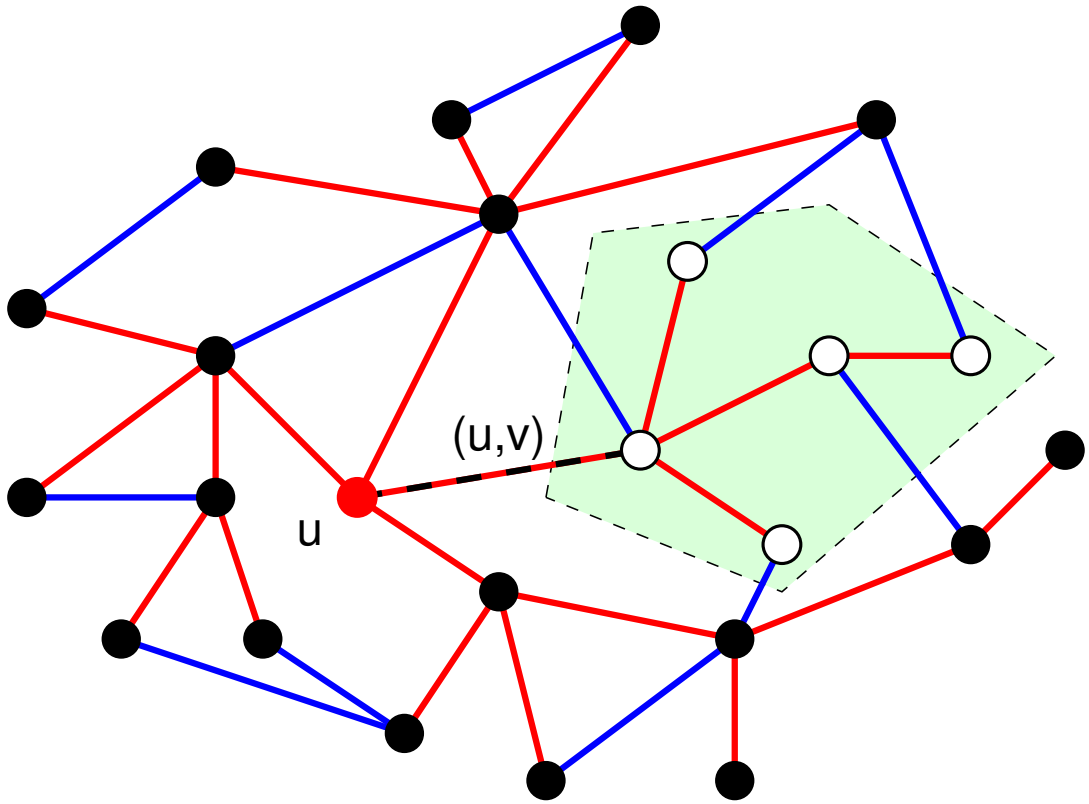
Zeit- und Platz-Bedarf:

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

Vorberechnung

Vorgehen:

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



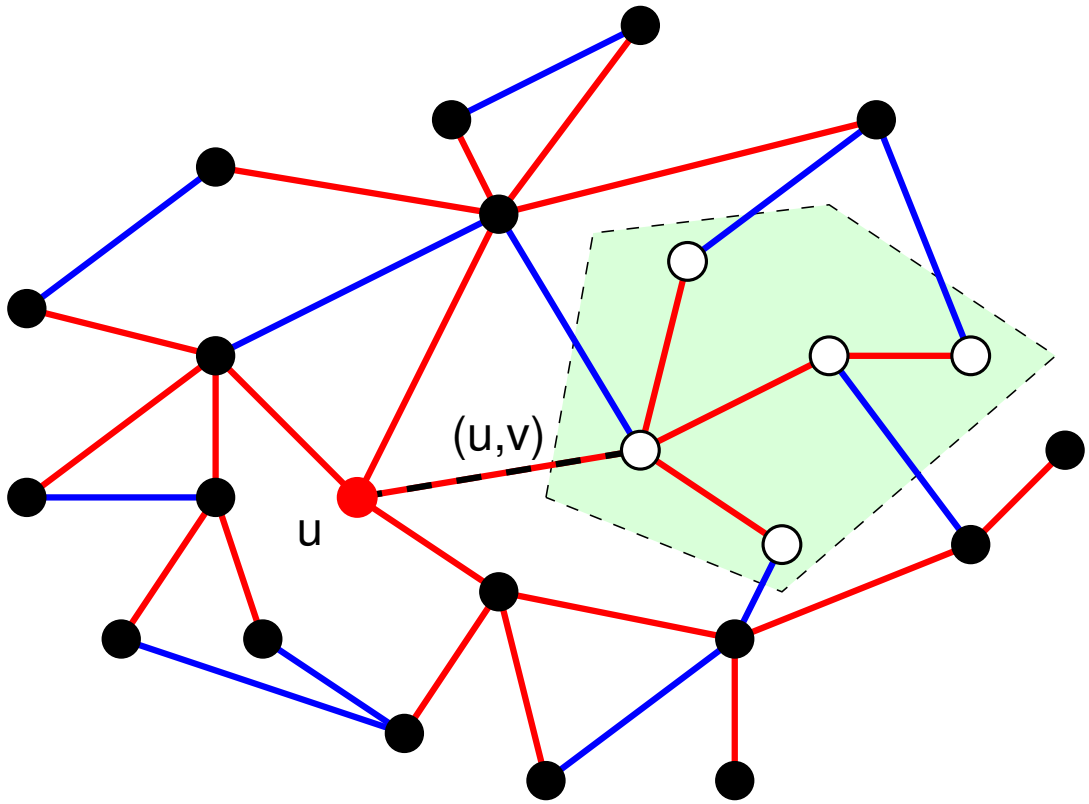
Zeit- und Platz-Bedarf:

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

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 $\mathcal{O}(\underbrace{m \cdot |C|}_{\text{Container}} + \underbrace{n(m + n \log n)}_{\text{all pair shortest path}})$
- 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

Diskussion

Vorteile:

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

Nachteile:

- Laufzeit der Vorberechnung $\mathcal{O}(\underbrace{m \cdot |C|}_{\text{Container}} + \underbrace{n(m + n \log n)}_{\text{all pair shortest path}})$
- 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

Diskussion

Vorteile:

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

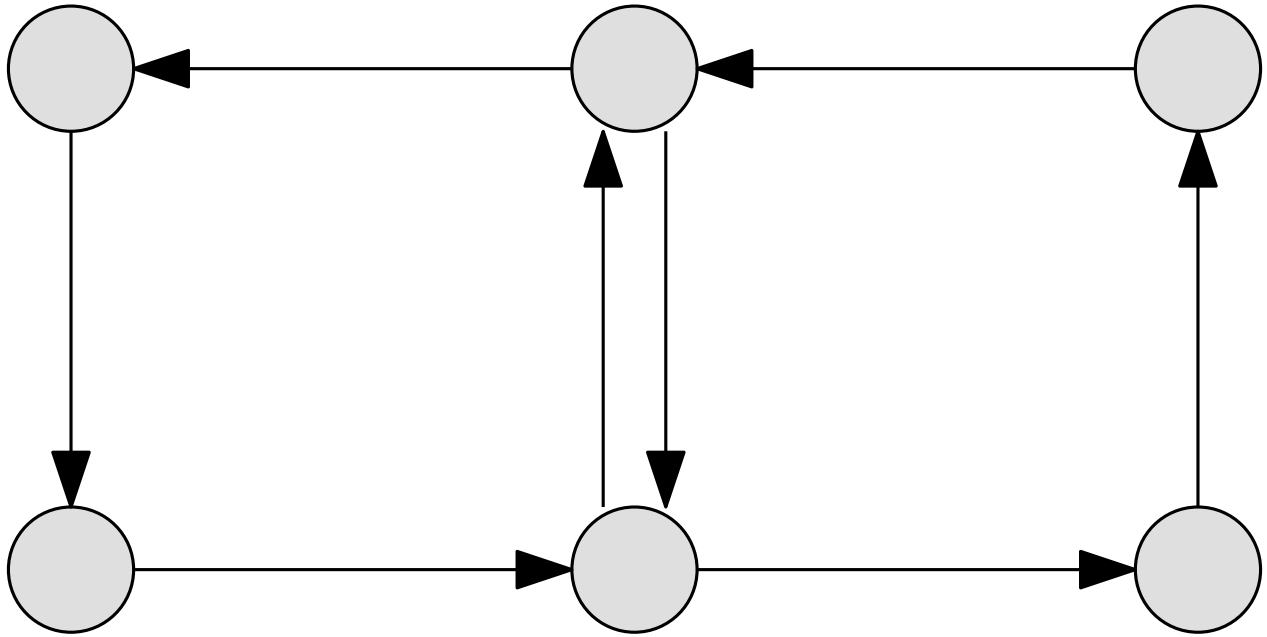
Nachteile:

- Laufzeit der Vorberechnung $\mathcal{O}(\underbrace{m \cdot |C|}_{\text{Container}} + \underbrace{n(m + n \log n)}_{\text{all pair shortest path}})$
- 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:

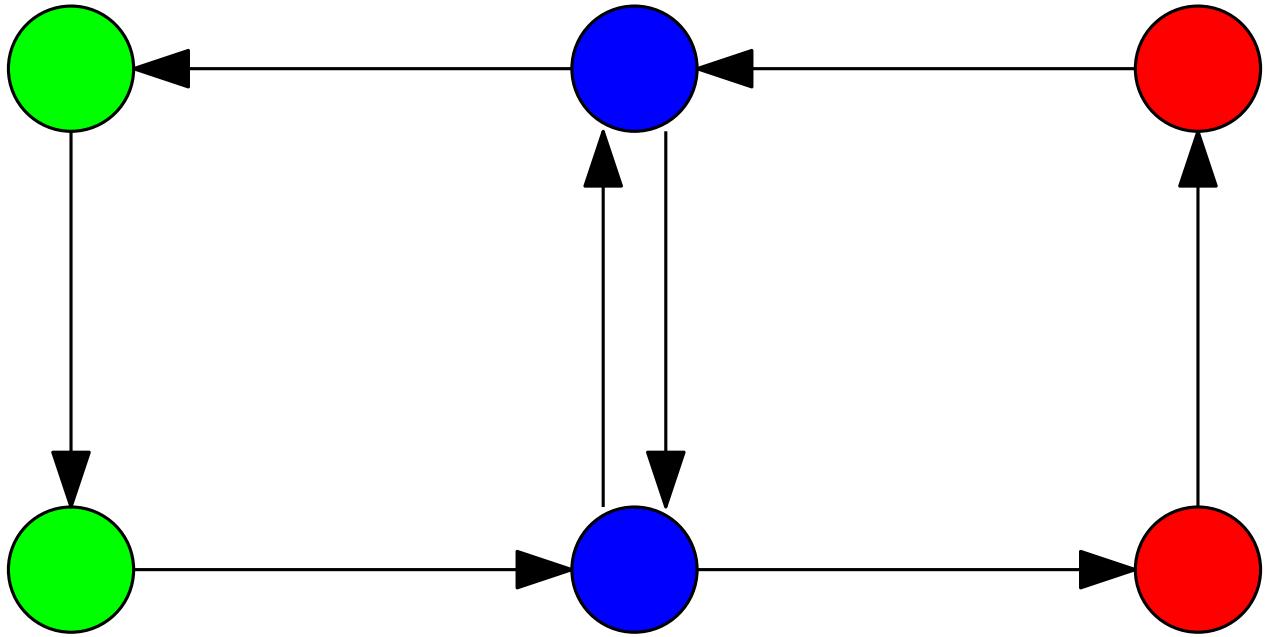
- invertiere die Idee der Geometrischen Container
- partitioniere Graphen in k Zellen
- hänge Label mit k Bits an jede Kante
- gibt an, ob e für Ziele in Zielzelle T benötigt wird



Arc-Flags

Idee:

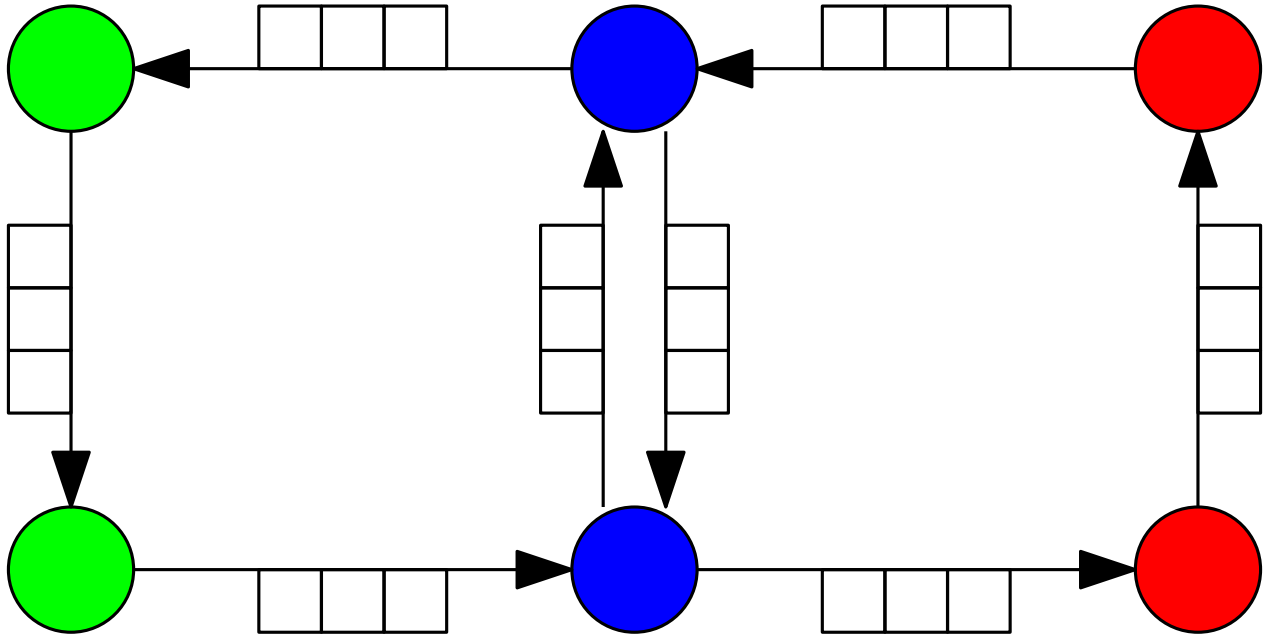
- invertiere die Idee der Geometrischen Container
- partitioniere Graphen in k Zellen
- hänge Label mit k Bits an jede Kante
- gibt an, ob e für Ziele in Zielzelle T benötigt wird



Arc-Flags

Idee:

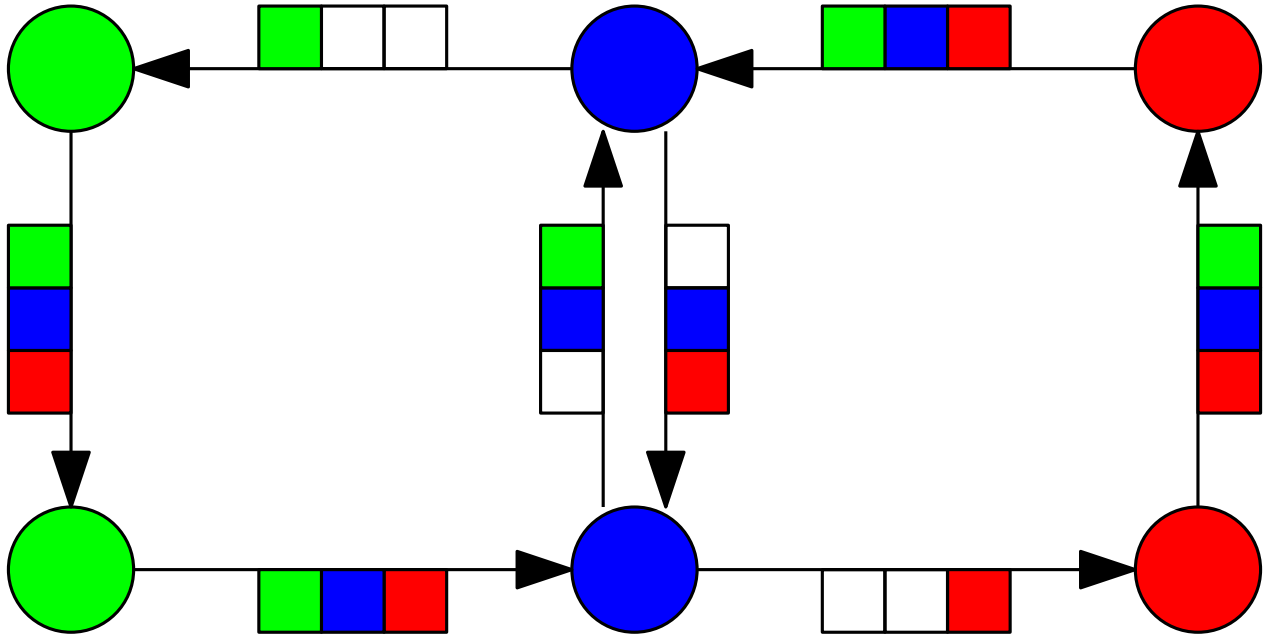
- invertiere die Idee der Geometrischen Container
- partitioniere Graphen in k Zellen
- hänge Label mit k Bits an jede Kante
- gibt an, ob e für Ziele in Zielzelle T benötigt wird



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?

Partitionierung

Anforderungen:

- balanciert

mögliche Partitionen:

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

weitere Anforderungen?

Partitionierung

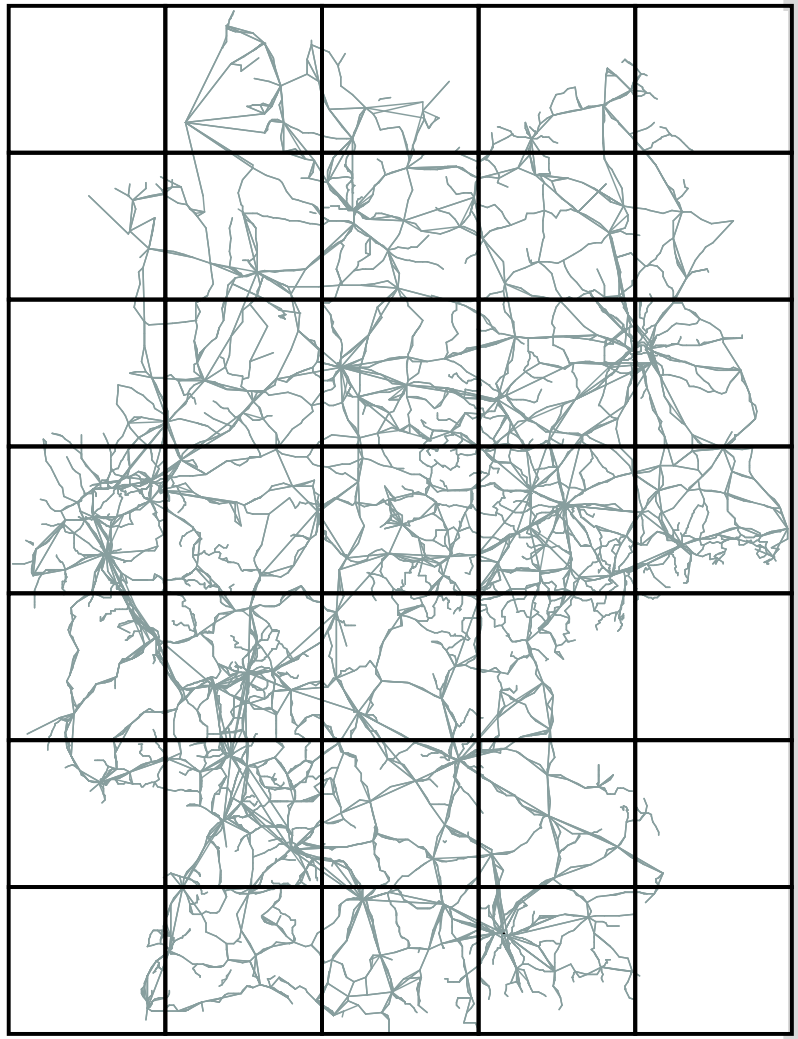
Anforderungen:

- balanciert

mögliche Partitionen:

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

weitere Anforderungen?



Partitionierung

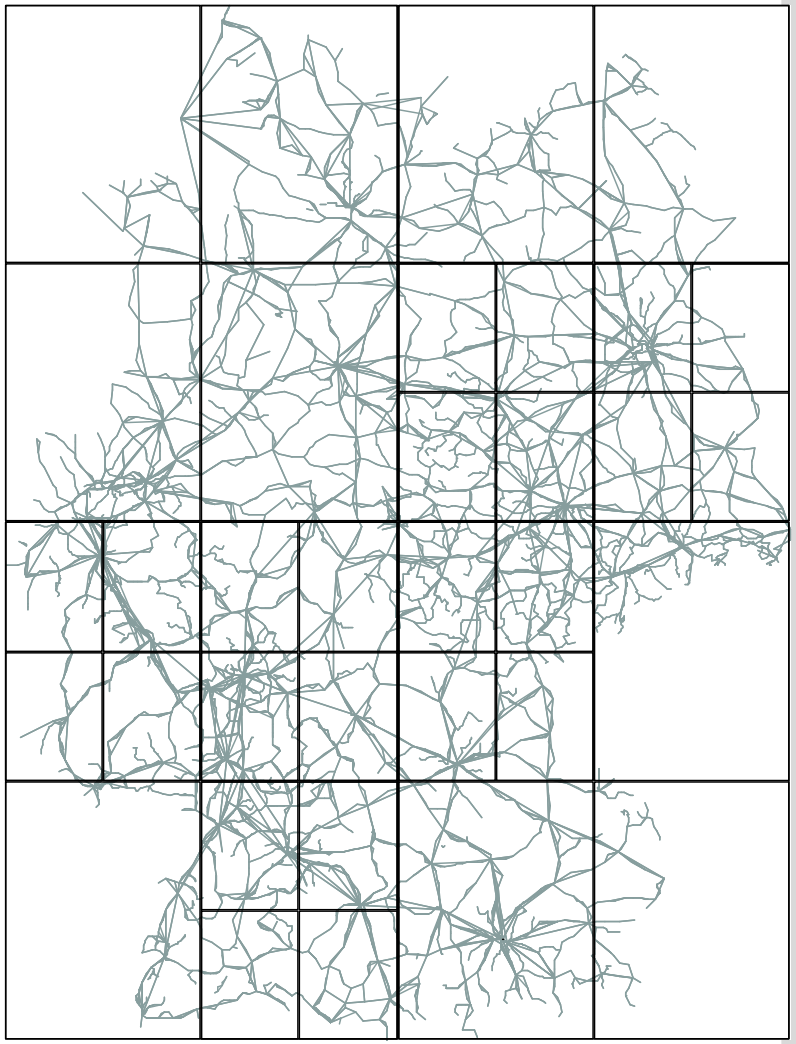
Anforderungen:

- balanciert

mögliche Partitionen:

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

weitere Anforderungen?



Partitionierung

Anforderungen:

- balanciert

mögliche Partitionen:

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

weitere Anforderungen?

Partitionierung

Anforderungen:

- balanciert

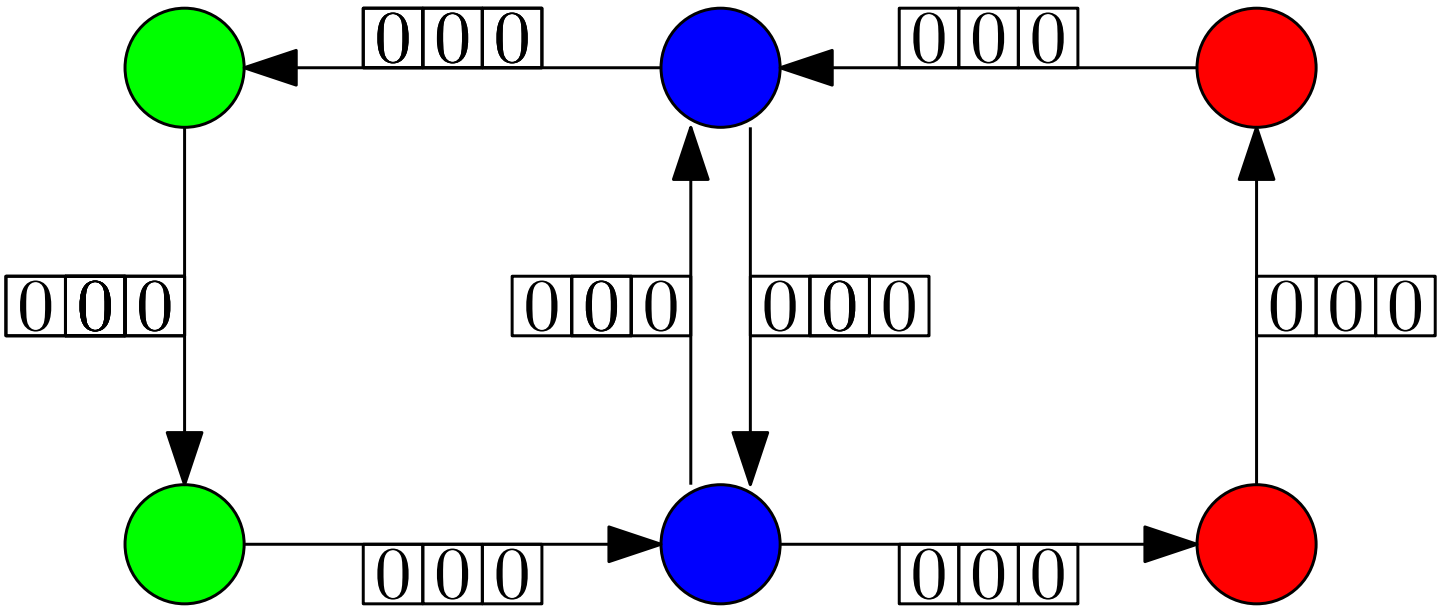
mögliche Partitionen:

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

weitere Anforderungen?

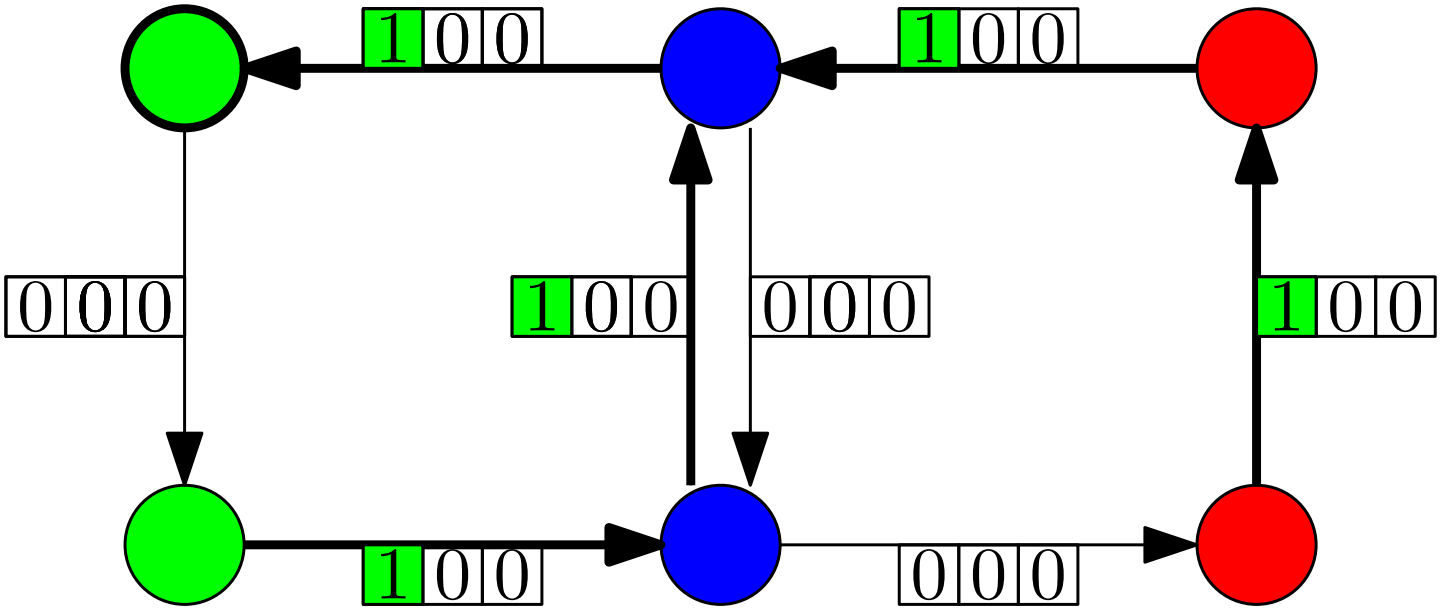
Flaggenberechnung: Erster Versuch

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



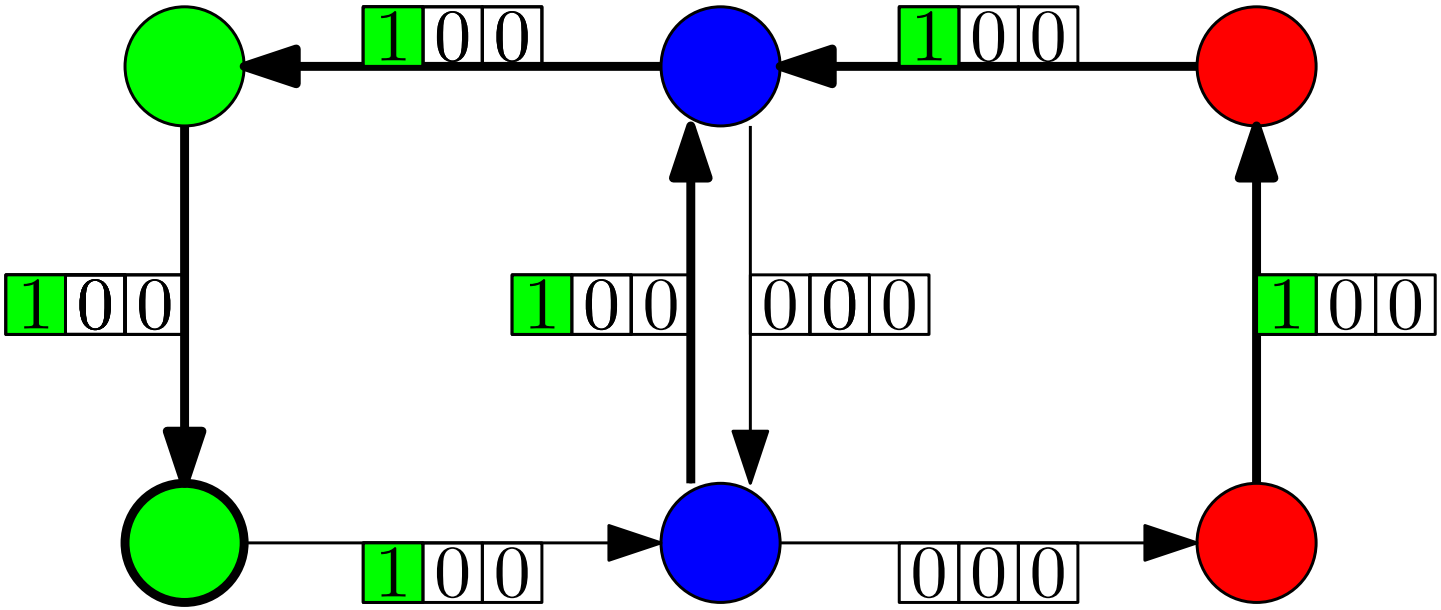
Flaggenberechnung: Erster Versuch

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



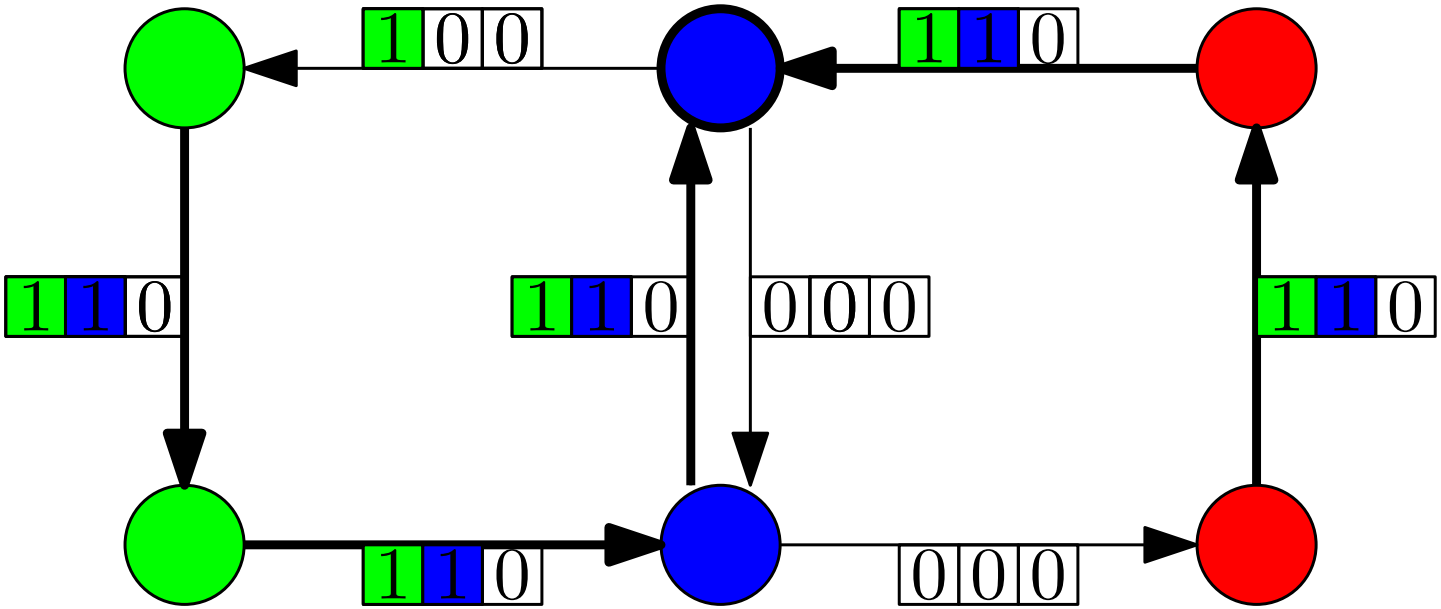
Flaggenberechnung: Erster Versuch

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



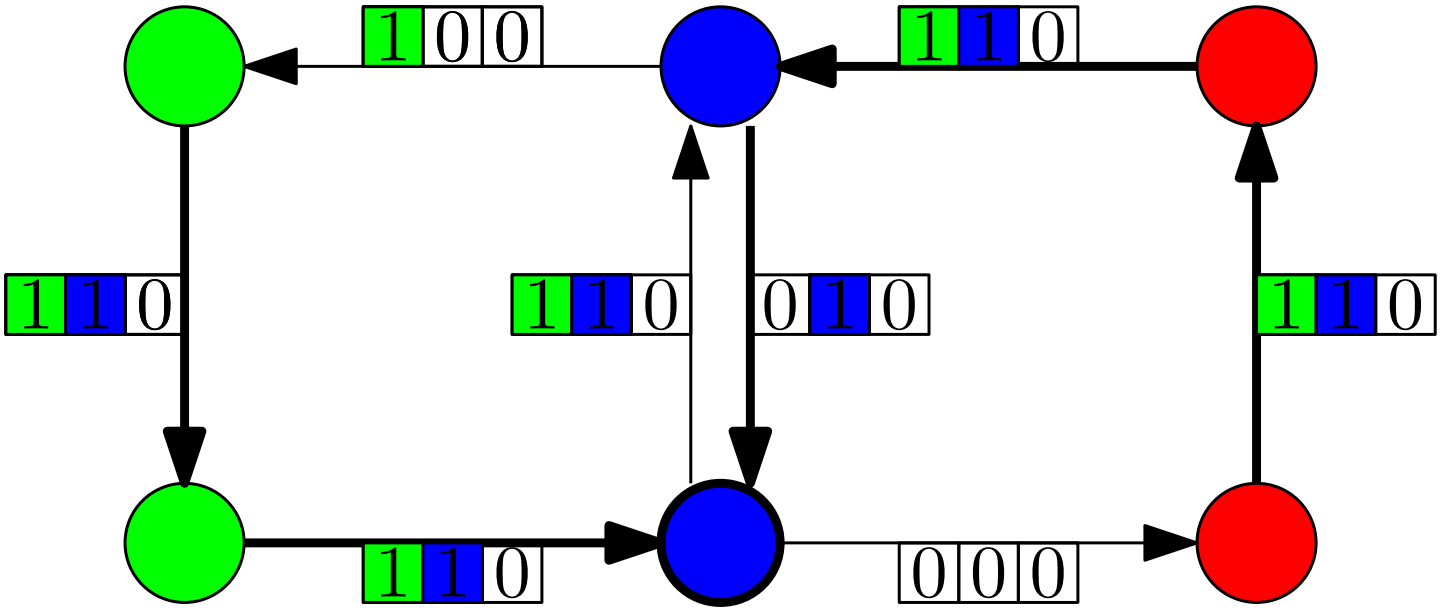
Flaggenberechnung: Erster Versuch

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



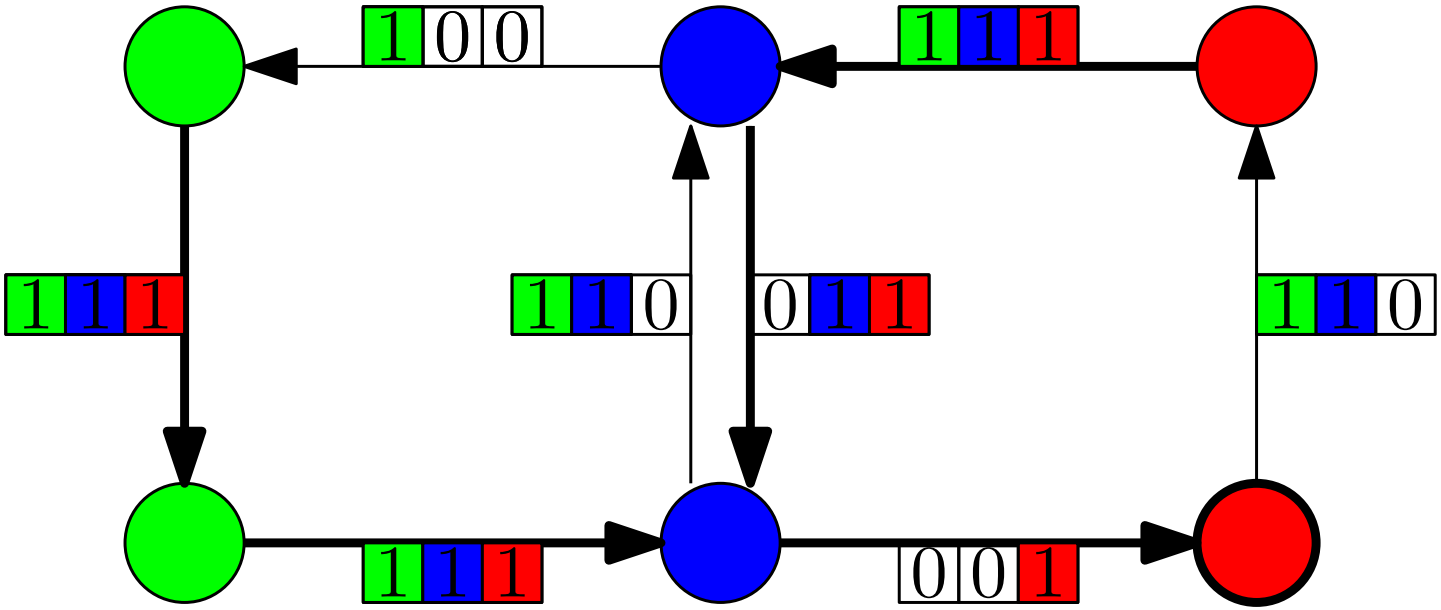
Flaggenberechnung: Erster Versuch

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



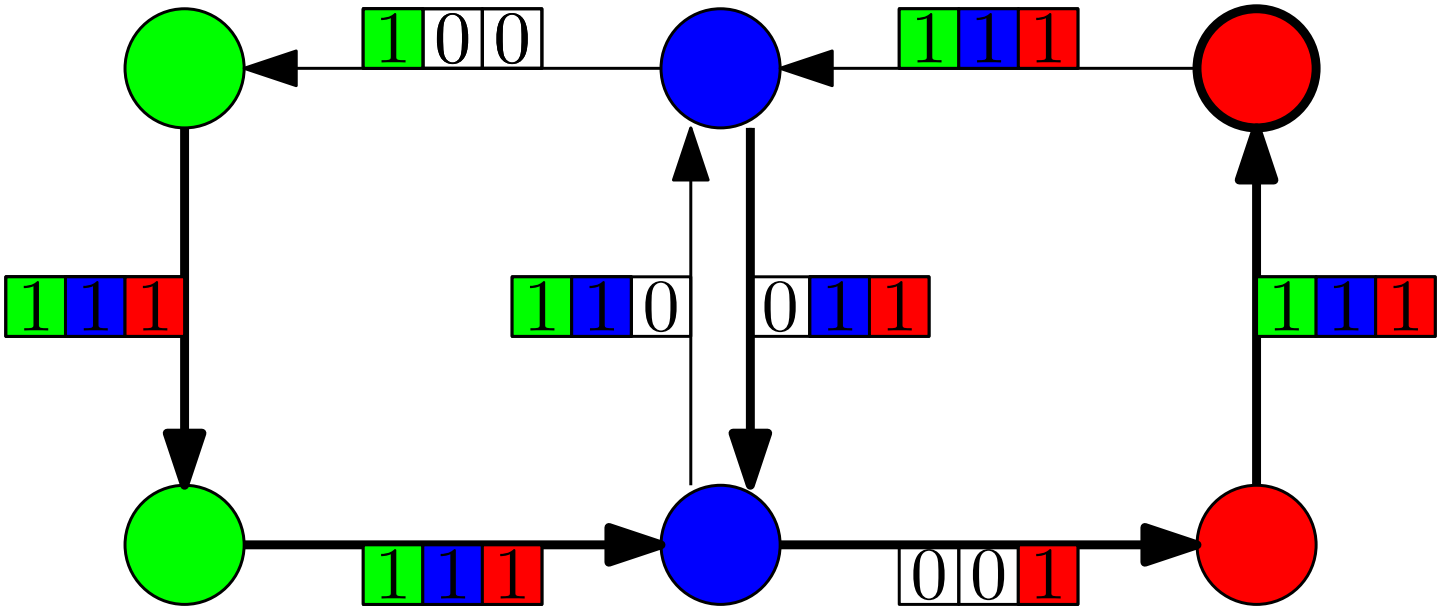
Flaggenberechnung: Erster Versuch

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



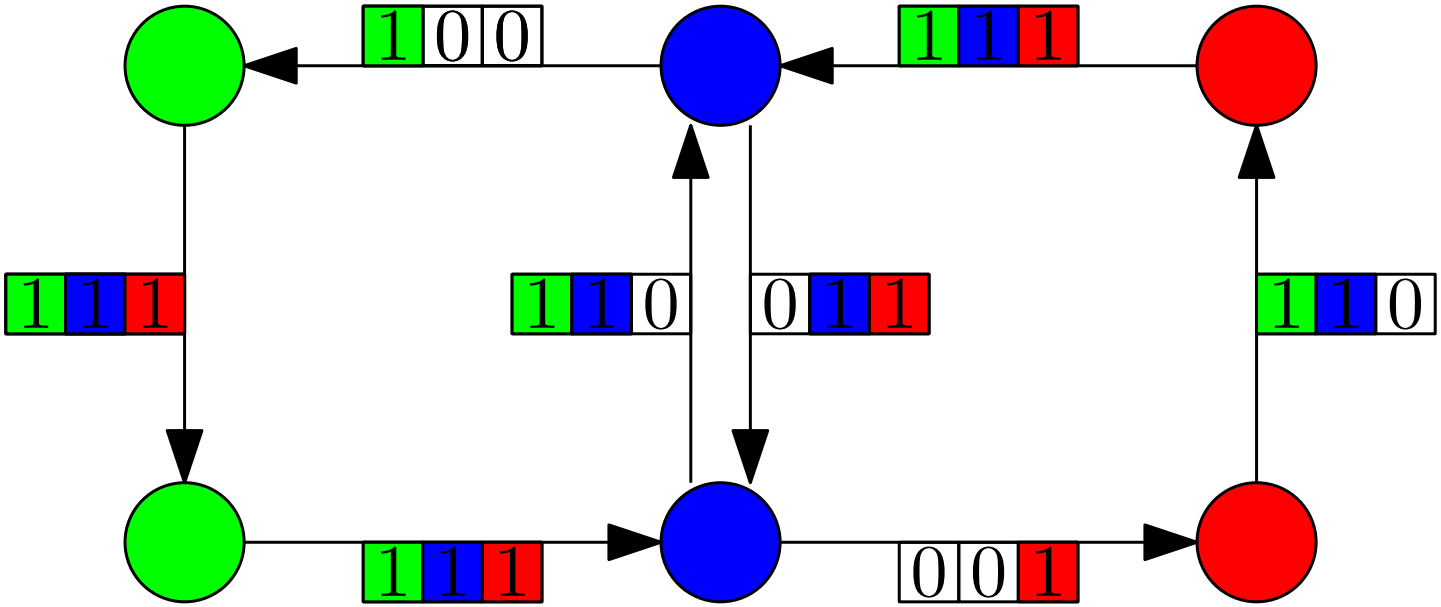
Flaggenberechnung: Erster Versuch

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



Flaggenberechnung: Erster Versuch

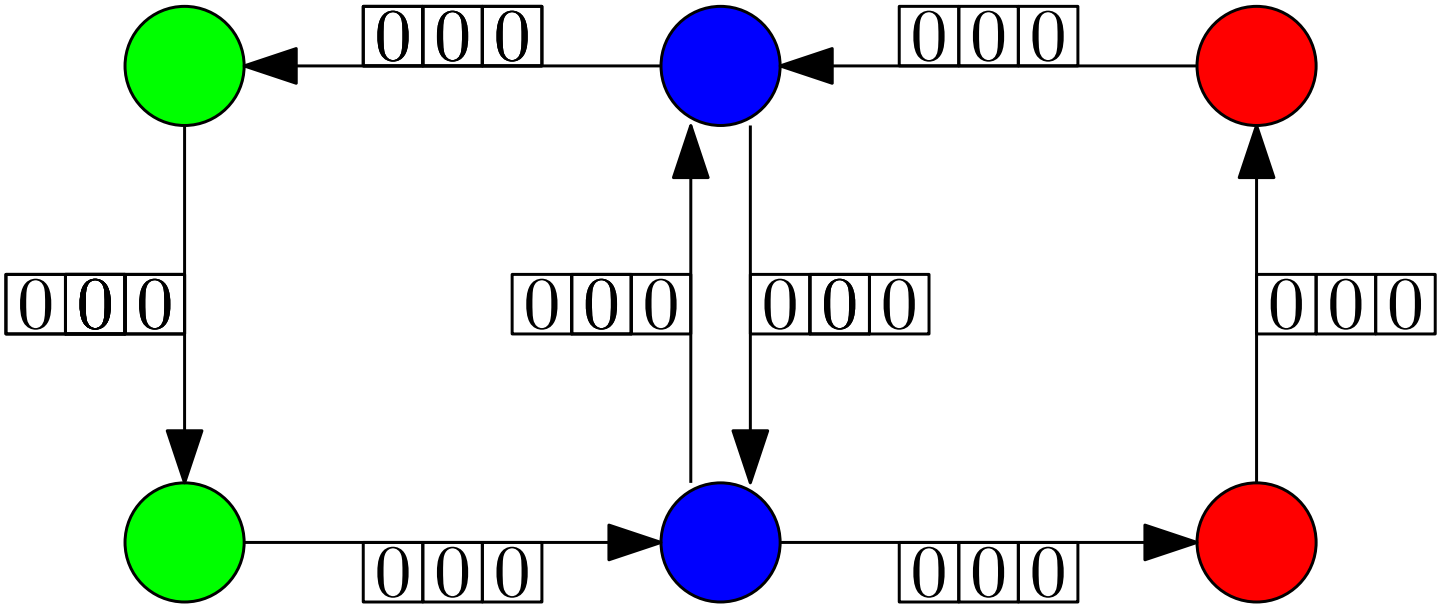
- von jedem Knoten
- konstruiere Rückwärts KW-Baum
- setze Flagge der Region der Wurzel für jede Baumkante
- wieder APSP und somit $>$ 400 Jahre



Flaggenberechnung: Randknoten

Beobachtung: Man muss durch Randknoten in die Zelle

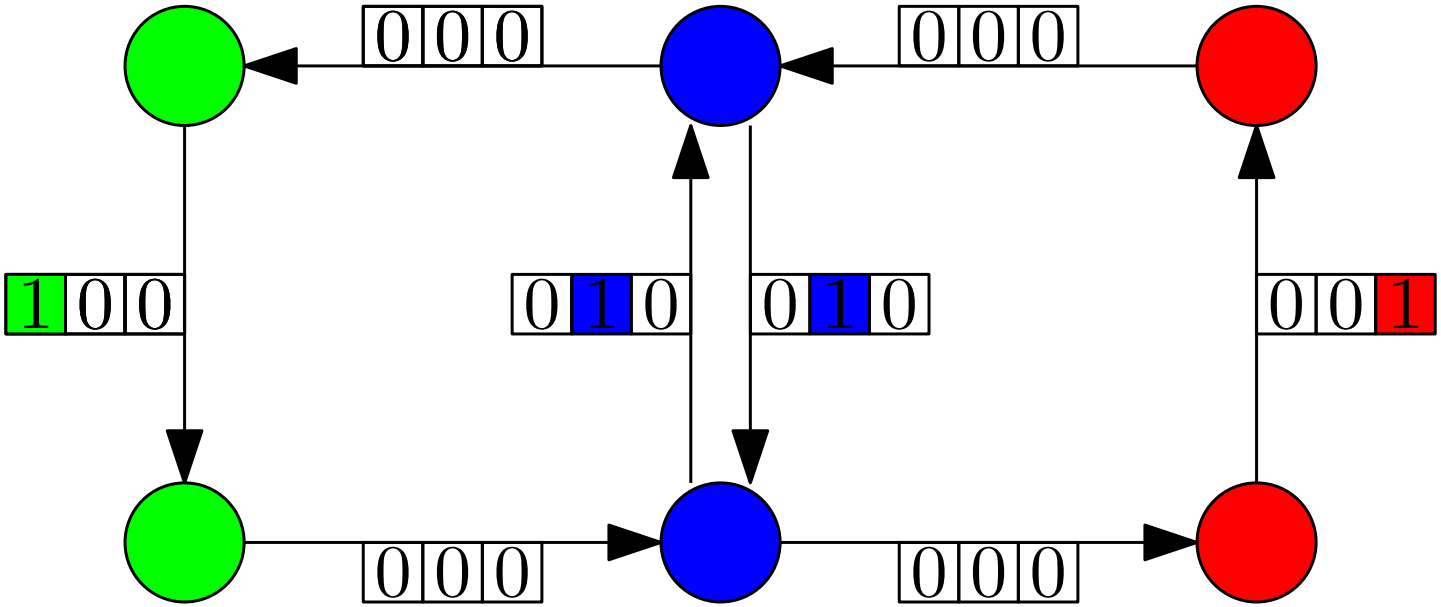
- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn e Baumkante des Baums von b ist



Flaggenberechnung: Randknoten

Beobachtung: Man muss durch Randknoten in die Zelle

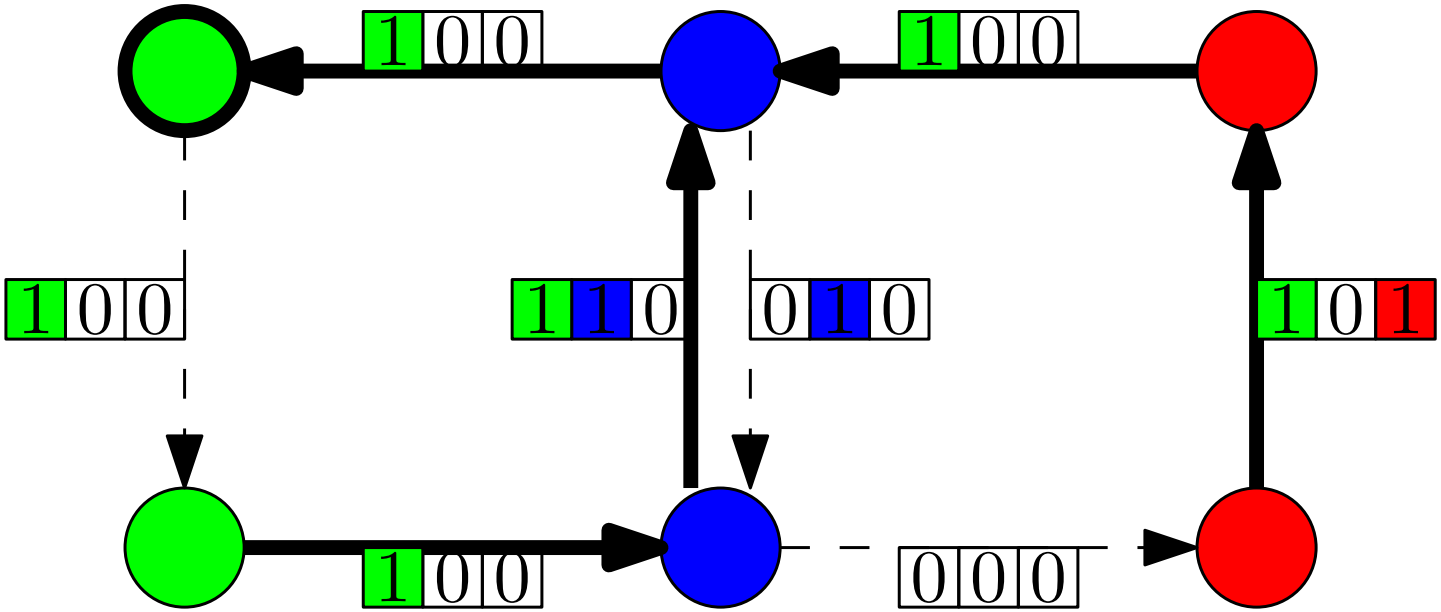
- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn e Baumkante des Baums von b ist



Flaggenberechnung: Randknoten

Beobachtung: Man muss durch Randknoten in die Zelle

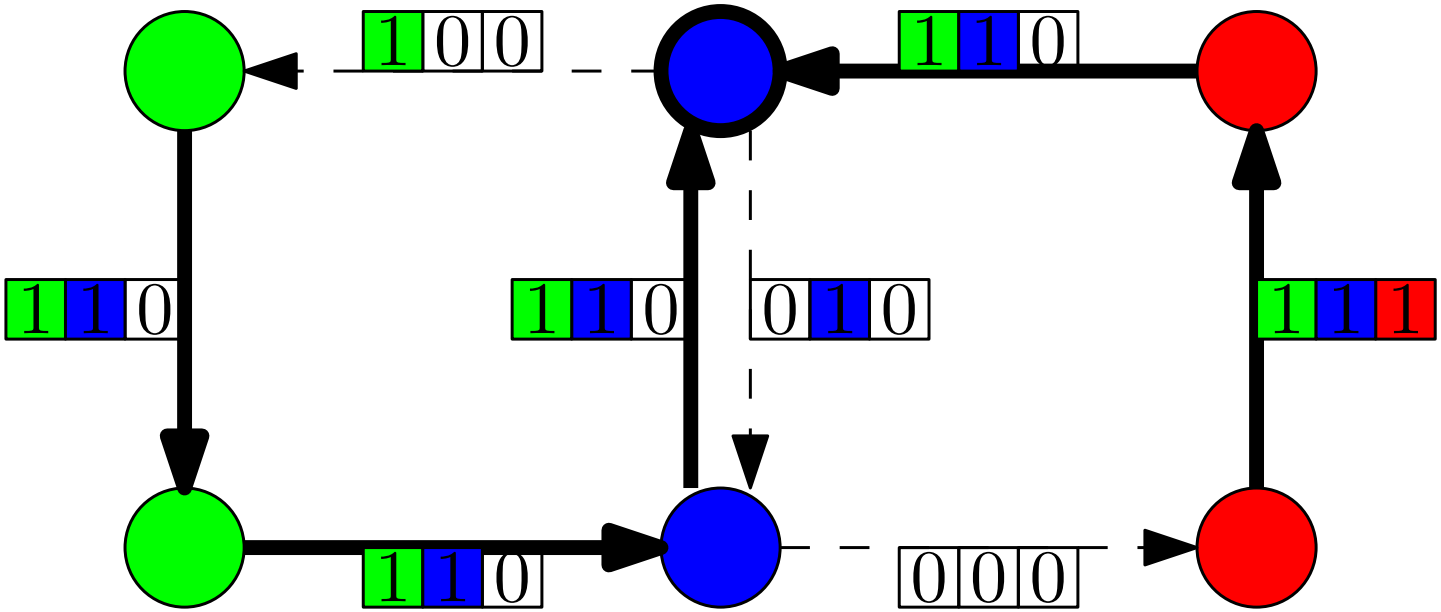
- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn e Baumkante des Baums von b ist



Flaggenberechnung: Randknoten

Beobachtung: Man muss durch Randknoten in die Zelle

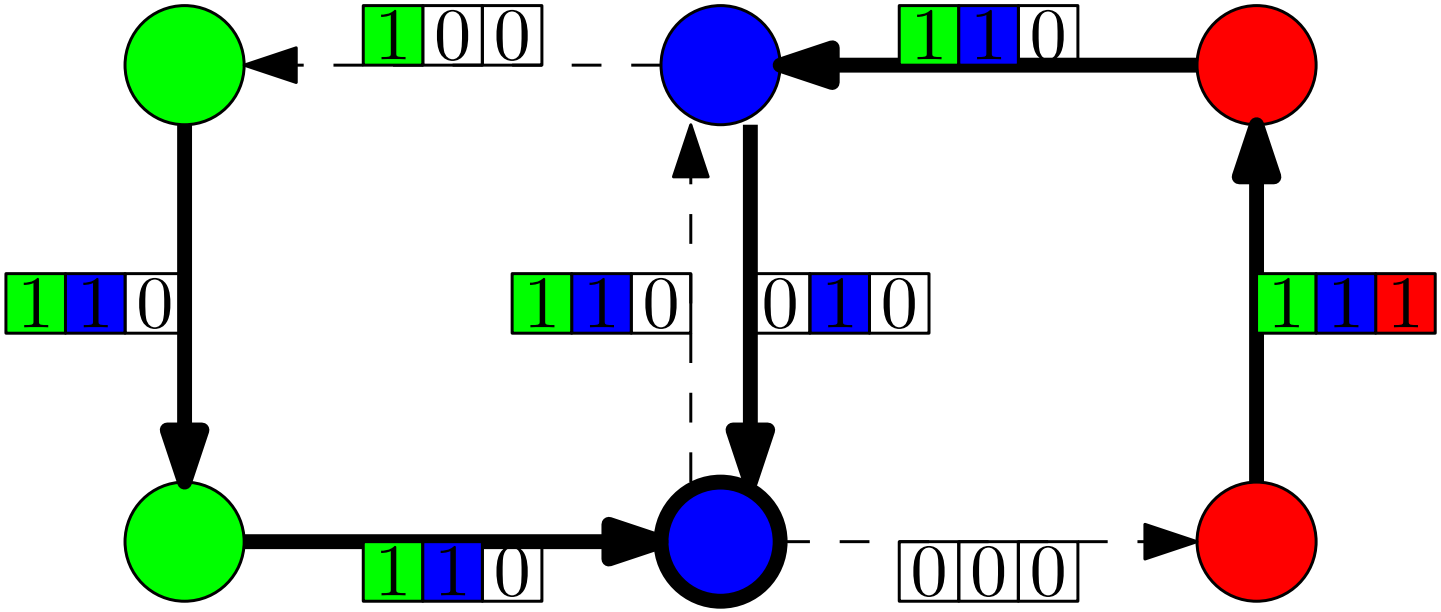
- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn e Baumkante des Baums von b ist



Flaggenberechnung: Randknoten

Beobachtung: Man muss durch Randknoten in die Zelle

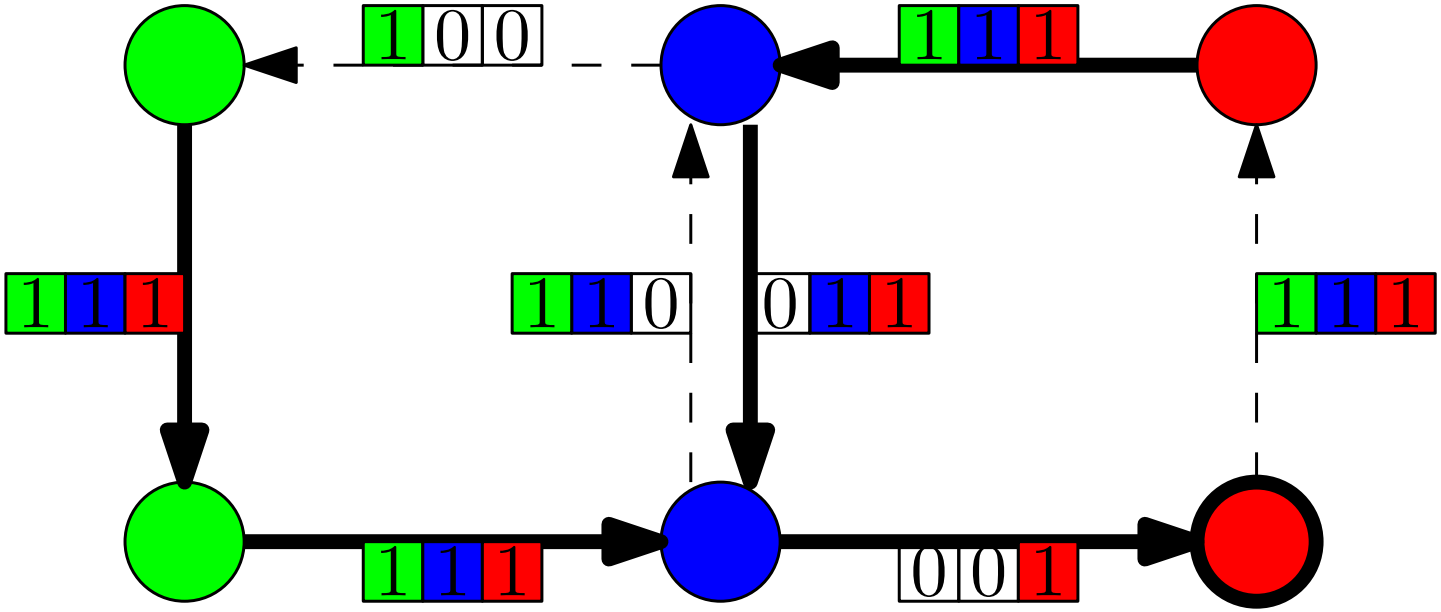
- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten *b*
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn *e* Baumkante des Baums von *b* ist



Flaggenberechnung: Randknoten

Beobachtung: Man muss durch Randknoten in die Zelle

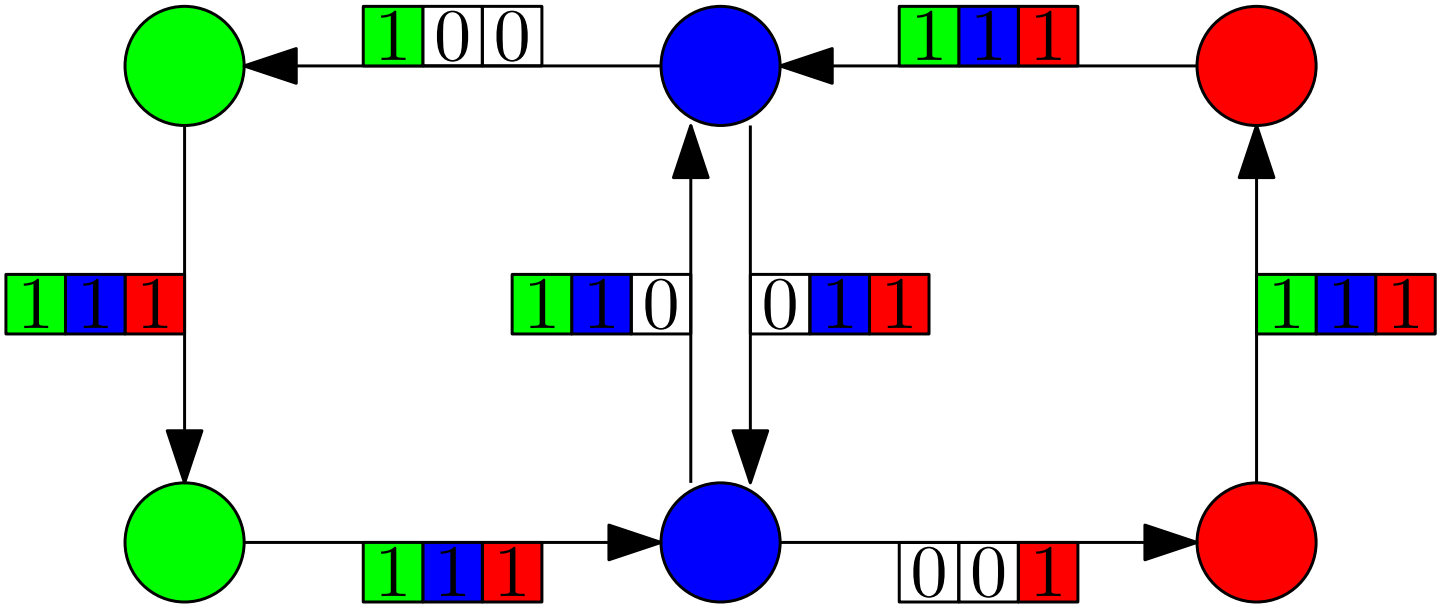
- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \texttt{true}$ wenn e Baumkante des Baums von b ist



Flaggenberechnung: Randknoten

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



Ende

Nächste Vorlesung:
Montag, 3. Mai, 14:00 Uhr

Ende

Literatur (Bi-Suche, ALT):

- Andrew V. Goldberg and Chris Harrelson:
  Computing the Shortest Path: A Search Meets Graph Theory.
  In: *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, 2005 pages 156–165.
- Andrew V. Goldberg and Renato F. Werneck:
  Computing Point-to-Point Shortest Paths from External Memory.
  In: *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, 2005 pages 26–40.

Username: `routePlanning` Passwort: `ss10`

Ende

Literatur (Geometrische Container, Arc-Flags):

- Dorothea Wagner and Thomas Willhalm and Christos Zaroliagis:
  Geometric Containers for Efficient Shortest-Path Computation
  In: *ACM Journal of Experimental Algorithmics, 2005 article 1.3.*
- Moritz Hilger and Ekkehard Köhler and Rolf H. Möhring and Heiko Schilling:
  Fast Point-to-Point Shortest Path Computations with Arc-Flags
  In: *Shortest Paths: Ninth DIMACS Implementation Challenge, 2009*

Username: `routePlanning` Passwort: `ss10`