

Algorithmen II

Peter Sanders, Timo Bingmann

Übungen:

Sebastian Lamm, Demian Hesse

Institut für Theoretische Informatik

Web:

http://algo2.itl.kit.edu/AlgorithmenII_WS18.php

Fortgeschrittene Graphenalgorithmen

2 Kürzeste Wege

Folien teilweise von Rob van Stee

Eingabe: Graph $G = (V, E)$

Kostenfunktion/Kantengewicht $c : E \rightarrow \mathbb{R}$

Anfangsknoten s .

Ausgabe: für alle $v \in V$

Länge $\mu(v)$ des kürzesten Pfades von s nach v ,

$$\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$$

$$\text{mit } c(\langle e_1, \dots, e_k \rangle) := \sum_{i=1}^k c(e_i).$$

Oft wollen wir auch „geeignete“ **Repräsentation der kürzesten Pfade**.



Allgemeine Definitionen

Wie bei BFS benutzen wir zwei Knotenarrays:

- $d[v]$ = aktuelle (vorläufige) Distanz von s nach v

Invariante: $d[v] \geq \mu(v)$

- $\text{parent}[v]$ = Vorgänger von v

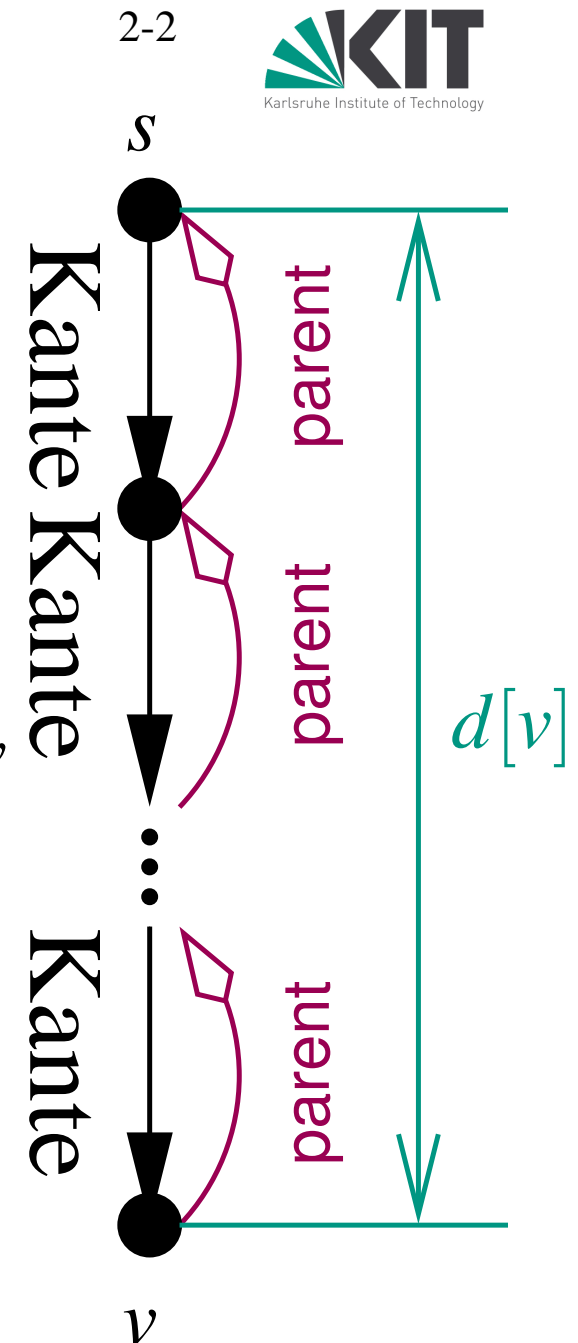
auf dem (vorläufigen) kürzesten Pfad von s nach v

Invariante: dieser Pfad bezeugt $d[v]$

Initialisierung:

$$d[s] = 0, \text{parent}[s] = s$$

$$d[v] = \infty, \text{parent}[v] = \perp$$



Kante (u, v) relaxieren

falls $d[u] + c(u, v) < d[v]$

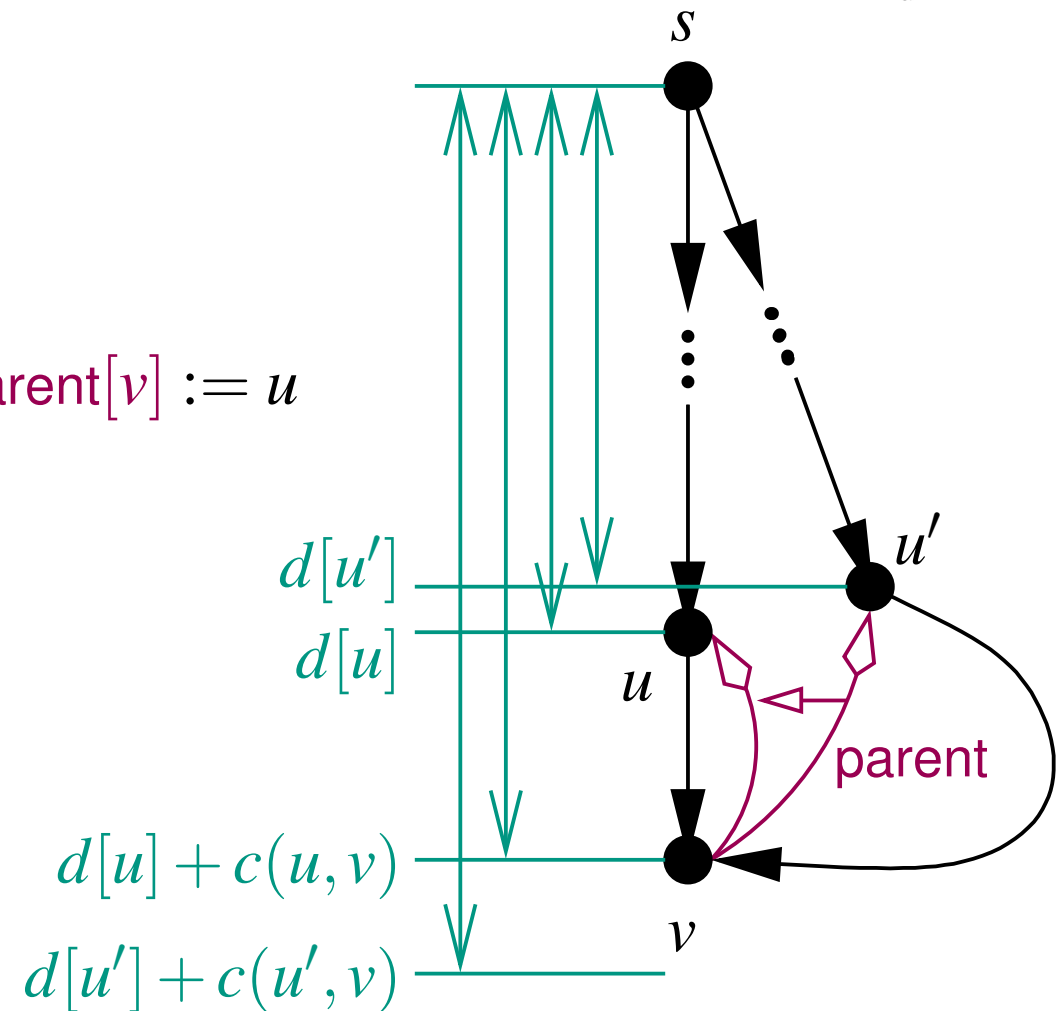
vielleicht $d[v] = \infty$

setze $d[v] := d[u] + c(u, v)$ und $\text{parent}[v] := u$

Invarianten bleiben erhalten!

Beobachtung:

$d[v]$ Kann sich mehrmals ändern!



Dijkstra's Algorithmus: Pseudocode

initialize d , parent

all nodes are non-scanned

while \exists non-scanned node u with $d[u] < \infty$

$u :=$ non-scanned node v with minimal $d[v]$

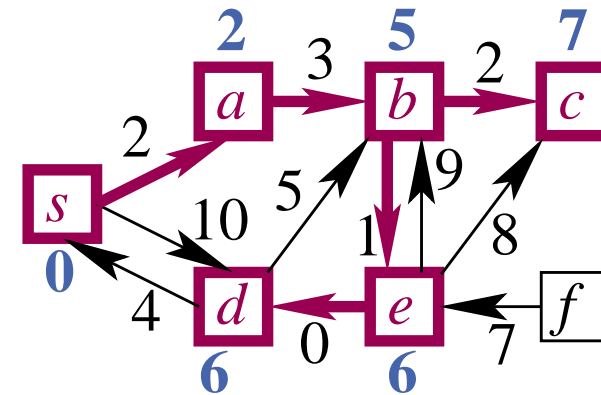
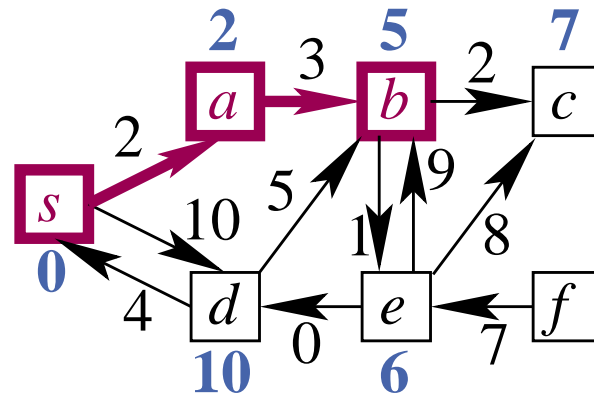
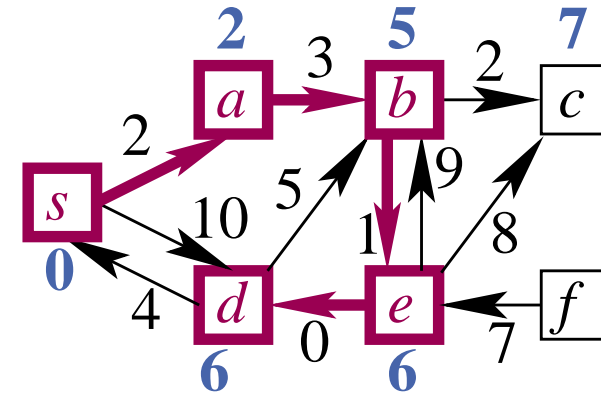
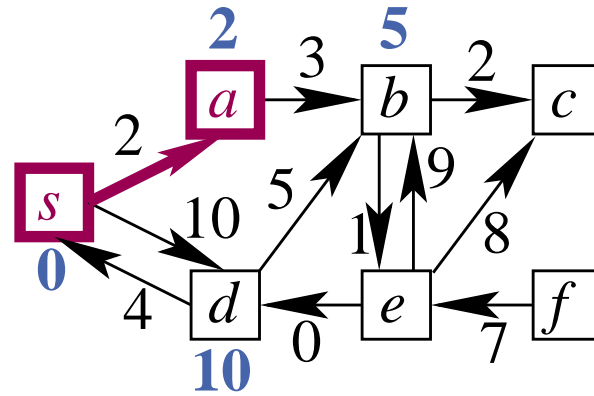
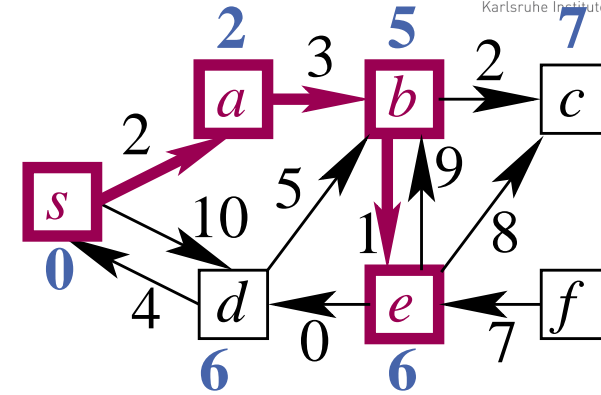
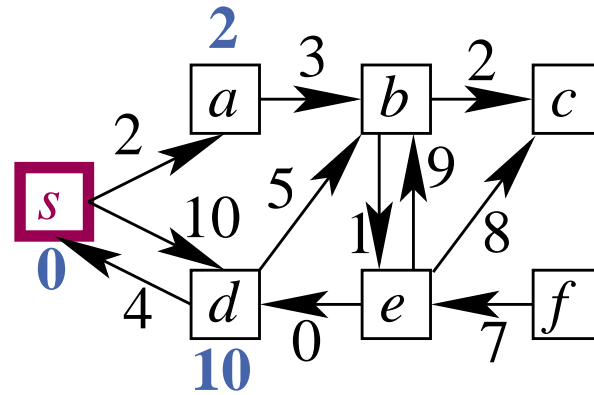
relax all edges (u, v) out of u

u is scanned now

Behauptung: Am Ende definiert d die optimalen Entfernungen und parent die zugehörigen Wege (siehe Algo I:)

- v erreichbar $\implies v$ wird irgendwann gescannt
- v gescannt $\implies \mu(v) = d[v]$

Beispiel



Laufzeit

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

Mit **Fibonacci-Heapprioritätslisten**:

- insert $O(1)$
- decreaseKey $O(1)$
- deleteMin $O(\log n)$ (amortisiert)

$$\begin{aligned} T_{\text{DijkstraFib}} &= O(m \cdot 1 + n \cdot (\log n + 1)) \\ &= O(m + n \log n) \end{aligned}$$

Aber: konstante Faktoren in $O(\cdot)$ sind hier größer als bei binären Heaps!

Laufzeit im Durchschnitt

Bis jetzt: $\leq m$ decreaseKeys ($\leq 1 \times$ pro Kante)

Wieviel decreaseKeys **im Durchschnitt**?

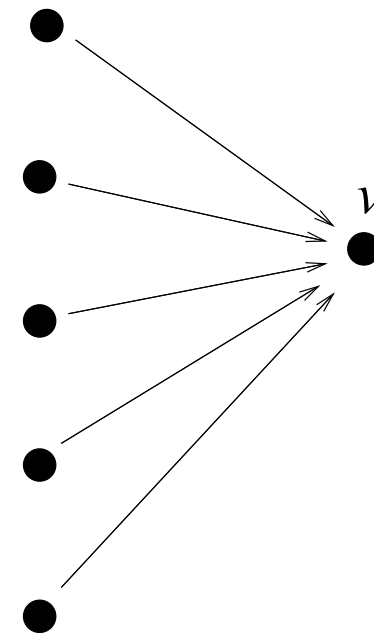
Modell:

- Beliebiger Graph G
- Beliebiger Anfangsknoten s
- Beliebige Mengen $C(v)$
von Kantengewichten für
eingehende Kanten von Knoten v

Gemittelt wird über alle Zuteilungen

$C(v) \rightarrow$ eingehende Kanten von v

Beispiel: alle Kosten unabhängig identisch verteilt

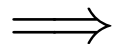


$\text{indegree}(v)=5$

$C(v)=\{c_1, \dots, c_5\}$

Laufzeit im Durchschnitt

Probabilistische Sichtweise: Zufällige gleichverteilte Auswahl der zu mittelnden Eingaben.



wir suchen den **Erwartungswert** der Laufzeit

Frage: Unterschied zu erwarteter Laufzeit bei randomisierten Algorithmen ?

Laufzeit im Durchschnitt

Satz 1. $\mathbb{E}[\#decreaseKey\text{-Operationen}] = O\left(n \log \frac{m}{n}\right)$

Dann

$$\begin{aligned}\mathbb{E}(T_{\text{DijkstraBHeap}}) &= O\left(m + n \log \frac{m}{n} \cdot T_{\text{decreaseKey}}(n) \right. \\ &\quad \left. + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\right) \\ &= O\left(m + n \log \frac{m}{n} \log n + n \log n\right) \\ &= O\left(m + n \log \frac{m}{n} \log n\right)\end{aligned}$$

(wir hatten vorher $T_{\text{DijkstraBHeap}} = O((m + n) \log n)$)

($T_{\text{DijkstraFib}} = O(m + n \log n)$ schlechtester Fall)

Lineare Laufzeit für dichte Graphen

$m = \Omega(n \log n \log \log n) \Rightarrow$ lineare Laufzeit.

(nachrechnen)

Also hier u. U. besser als Fibonacci heaps

Satz 1. $\mathbb{E}[\#decreaseKey\text{-}Operationen] = O\left(n \log \frac{m}{n}\right)$

Satz 1. $\mathbb{E}[\#\text{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

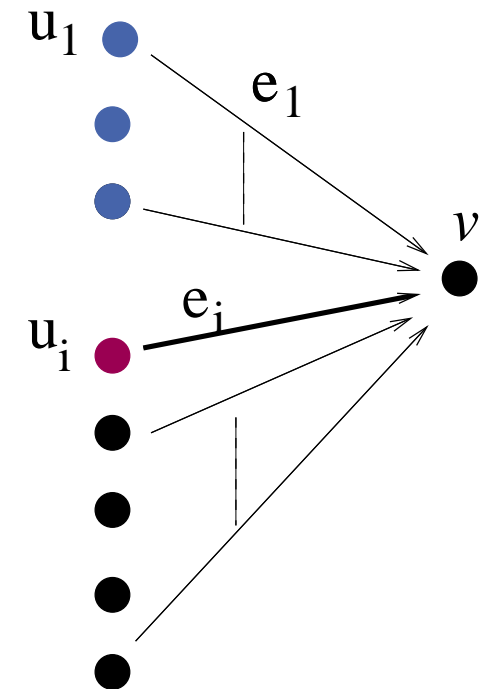
decreaseKey bei Bearbeitung von e_i nur wenn

$$\mu(u_i) + c(e_i) < \min_{j < i} (\mu(u_j) + c(e_j)).$$

Aber $\mu(u_i) \geq \mu(u_j)$ für $j < i$, also muss gelten:

$$c(e_i) < \min_{j < i} c(e_j).$$

Präfixminimum



Satz 1. $\mathbb{E}[\#decreaseKey\text{-Operationen}] = O\left(n \log \frac{m}{n}\right)$

Kosten in $C(v)$ erscheinen in **zufälliger Reihenfolge**

Wie oft findet man ein neues Minimum bei zufälliger Reihenfolge?

Harmonische Zahl H_k (Sect. 2.8, s.u.)

Erstes Minimum: führt zu $\text{insert}(v)$.

Also $\leq H_k - 1 \leq (\ln k + 1) - 1 = \ln k$ erwartete decreaseKeys

Satz 1. $\mathbb{E}[\#decreaseKey\text{-Operationen}] = O\left(n \log \frac{m}{n}\right)$

Für Knoten $v \leq H_k - 1 \leq \ln k$ decreaseKeys (erwartet) mit $k = \text{indegree}(v)$.

Insgesamt

$$\sum_{v \in V} \ln \text{indegree}(v) \leq n \ln \frac{m}{n}$$

(wegen Konkavität von $\ln x$)

Präfixminima einer Zufallsfolge

Definiere Zufallsvariable M_n als Anzahl Präfixminima einer Folge von n verschiedenen Zahlen (in Abhängigkeit von einer Zufallspermutation)

Definiere Indikatorzufallsvariable $I_i := 1$ gdw. die i -te Zahl ein Präfixminimum ist.

$$\begin{aligned}
 \mathbb{E}[M_n] &= \mathbb{E}\left[\sum_{i=1}^n I_i\right] \stackrel{\text{Lin. E}[\cdot]}{=} \sum_{i=1}^n \mathbb{E}[I_i] \\
 &= \sum_{i=1}^n \frac{1}{i} = H_n \text{ wegen } \mathbb{P}[I_i = 1] = \frac{1}{i}
 \end{aligned}$$

$$\underbrace{x_1, \dots, x_{i-1}}_{<x_i?}, x_i, x_{i+1}, \dots, x_n$$

Monotone ganzzahlige Prioritätslisten

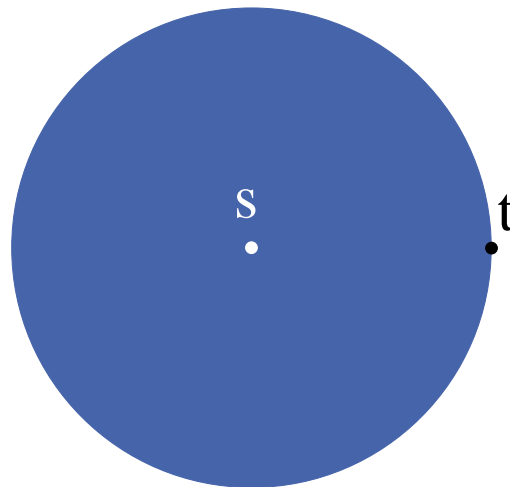
Grundidee: Datenstruktur auf Anwendung **zuschneiden**

Dijkstra's Algorithmus benutzt die **Prioritätsliste monoton**:

Operationen insert und decreaseKey benutzen Distanzen der Form

$$d[u] + c(e)$$

Dieser Wert **nimmt ständig zu**



Monotone ganzzahlige Prioritätslisten

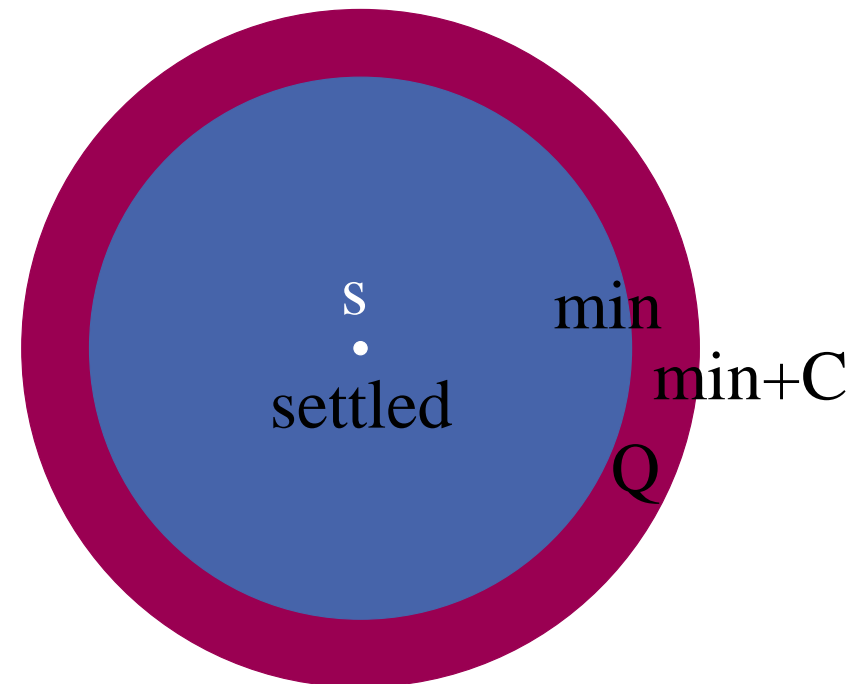
Annahme: Alle Kantengewichte sind ganzzahlig und im Intervall $[0, C]$

$$\implies \forall v \in V : d[v] \leq (n - 1)C$$

Es gilt sogar:

Sei min der letzte Wert,
der aus Q entfernt wurde.

In Q sind **immer**
nur Knoten mit Distanzen im
Intervall $[min, min + C]$.

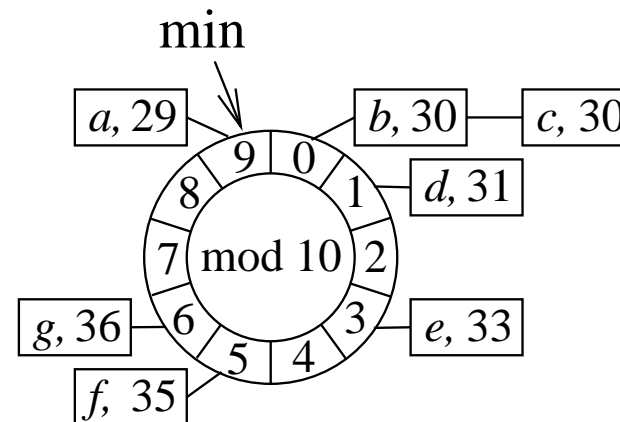


Bucket-Queue

Zyklisches Array B von $C + 1$ doppelt verketteten Listen

Knoten mit Distanz $d[v]$ wird in $B[d[v] \bmod (C + 1)]$ gespeichert.

Bucket queue with $C = 9$



Content=

<(a,29), (b,30), (c,30), (d,31)
(e,33), (f,35), (g,36)>

Operationen

Initialisierung: $C + 1$ leere Listen, $min = 0$

insert(v): fügt v in $B[d[v] \bmod (C + 1)]$ ein

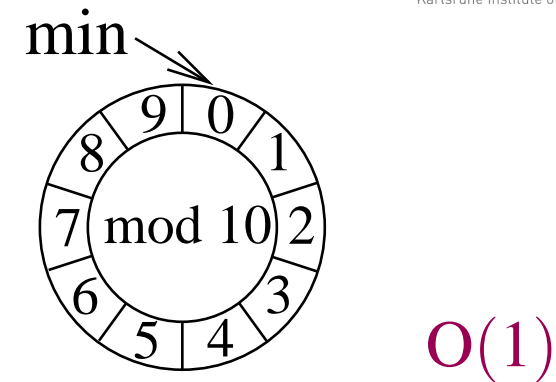
decreaseKey(v): entfernt v aus seiner Liste und fügt es ein in $B[d[v] \bmod (C + 1)]$

deleteMin: fängt an bei Bucket $B[min \bmod (C + 1)]$. Falls der leer ist, $min := min + 1$, wiederhole. **erfordert Monotonizität!**

min nimmt höchstens nC mal zu, höchstens n Elemente werden insgesamt aus Q entfernt \Rightarrow

Gesamtkosten deleteMin-Operationen = $O(n + nC) = O(nC)$.

Genauer: $O(n + \text{maxPathLength})$



Laufzeit Dijkstra mit Bucket-Queues

$$\begin{aligned} T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) \\ &\quad + \text{Kosten deleteMin-Operationen} \\ &\quad + n \cdot T_{\text{insert}}(n)) \end{aligned}$$

$$\begin{aligned} T_{\text{DijkstraBQ}} &= O(m \cdot 1 + nC + n \cdot 1) \\ &= O(m + nC) \text{ oder auch} \\ &= O(m + \text{maxPathLength}) \end{aligned}$$

Mit Radix-Heaps finden wir sogar $T_{\text{DijkstraRadix}} = O(m + n \cdot \log C)$

Idee: nicht alle Buckets gleich groß machen

Radix-Heaps

Wir verwenden die Buckets -1 bis K , für $K = 1 + \lfloor \log C \rfloor$

min = die zuletzt aus Q entfernte Distanz

Für jeden Knoten $v \in Q$ gilt $d[v] \in [min, \dots, min + C]$.

Betrachte **binäre Repräsentation** der möglichen Distanzen in Q .

Nehme zum Beispiel $C = 9$, binär 1001. Dann $K = 4$.

Beispiel 1: $min = 10000$, dann $\forall v \in Q : d[v] \in [10000, 11001]$

Beispiel 2: $min = 11101$, dann $\forall v \in Q : d[v] \in [11101, 100110]$

Speichere v in Bucket $B[i]$ falls $d[v]$ und min sich **zuerst an der i ten Stelle unterscheiden**, (in $B[K]$ falls $i > K$, in $B[-1]$ falls sie sich nicht unterscheiden)

Definition $msd(a, b)$

Die **Position** der **höchstwertigen Binärziffer** wo a und b sich unterscheiden

a	1100 1 010	10101 0 0	1110110
b	1100 0 101	10101 1 0	1110110
$msd(a, b)$	3	1	-1

$msd(a, b)$ können wir mit Maschinenbefehlen sehr schnell berechnen

Radix-Heap-Invariante

v ist gespeichert in Bucket $B[i]$ wo $i = \min(\text{msd}(\min, d[v]), K)$.

Beispiel 1: $\min = 10000$, $C = 9$, $K = 4$

Bucket	$d[v]$ binär	$d[v]$
-1	10000	16
0	10001	17
1	1001*	18,19
2	101**	20–23
3	11***	24–25
4	-	-

(In Bucket 4 wird nichts gespeichert)

Radix-Heap-Invariante

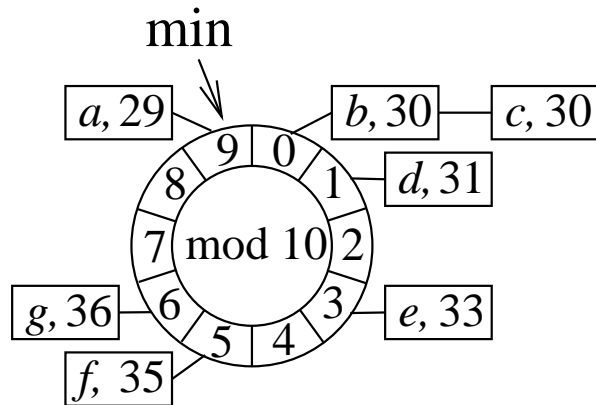
v ist gespeichert in Bucket $B[i]$ wo $i = \min(\text{msd}(\text{min}, d[v]), K)$.

Beispiel 2: $\text{min} = 11101$, $C = 9$, dann $K = 4$

Bucket	$d[v]$ binär	$d[v]$
-1	11101	29
0	-	-
1	1111*	30,31
2	-	-
3	-	-
4	100000 und höher	32 und höher

Falls $d[v] \geq 32$, dann $\text{msd}(\text{min}, d[v]) > 4!$

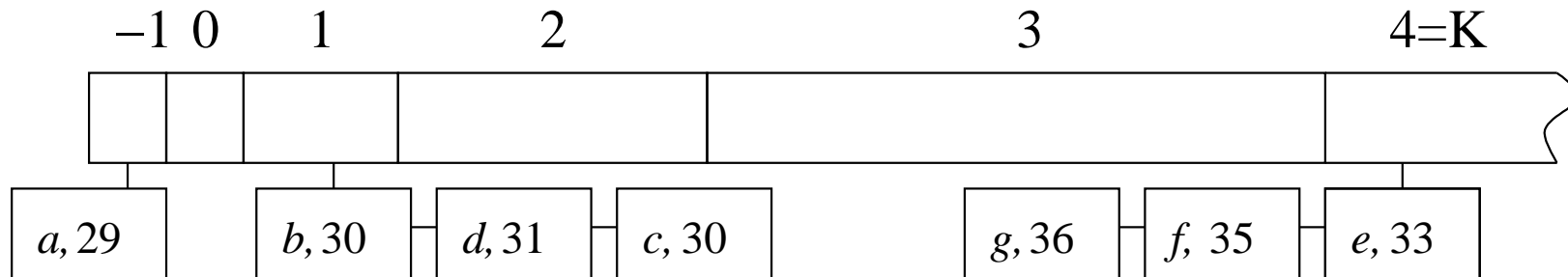
Bucket-Queues und Radix-Heaps



Bucket queue with $C = 9$

Content=

$\langle (a,29), (b,30), (c,30), (d,31), (e,33), (f,35), (g,36) \rangle$



Binary Radix Heap

Radix Heap: deleteMin

Function deleteMin: Element

if $B[-1] = \emptyset$

$i := \min \{ j \in 0..K : B[j] \neq \emptyset \}$

move $\min B[i]$ to $B[-1]$ and to \min

foreach $e \in B[i]$ **do** // exactly here invariant is violated !

 move e to $B[\min(\text{msd}(\min, d[v]), K)]$

result := $B[-1].\text{popFront}$

return result

$B[0], \dots, B[i-1]$: leer, also nichts zu tun.

$B[i+1], \dots, B[K]$: msd bleibt erhalten, weil altes und neues *min*

gleich für alle Bits $j > i$

Buckets $j > i$ bei Änderung von \min

Beispiel: $\min = 10000$, $C = 9$, $K = 4$.

Neues $\min = 10010$, war in Bucket 1

Bucket	$\min = 10000$		$\min = 10010$	
	$d[v]$ binär	$d[v]$	$d[v]$ binär	$d[v]$
-1	10000	16	10010	18
0	-	-	10011	19
1	1001*	18,19	-	-
2	101**	20–23	101**	20-23
3	11***	24–25	11***	24-27
4	-	-	-	-

Bucket $B[i]$ bei Änderung von min

Lemma: Elemente x aus $B[i]$ gehen zu Buckets mit **kleineren** Indices

Wir zeigen nur den Fall $i < K$.

Sei min_o der alte Wert von min .

Case $i < K$

	i	0
min_o	α	0
min	α	1
x	α	1

Kosten der deleteMin-Operationen

Bucket $B[i]$ finden: $O(i)$

Elemente aus $B[i]$ verschieben: $O(|B[i]|)$

Insgesamt $O(K + |B[i]|)$ falls $i \geq 0$, $O(1)$ falls $i = -1$

Verschiebung erfolgt immer nach **kleineren** Indices

Wir zahlen dafür **schon beim insert** (amortisierte Analyse):

es gibt höchstens K Verschiebungen eines Elements

Laufzeit Dijkstra mit Radix-Heaps

Insgesamt finden wir amortisiert

$$\square T_{\text{insert}}(n) = O(K)$$

$$\square T_{\text{deleteMin}}(n) = O(K)$$

$$\square T_{\text{decreaseKey}}(n) = O(1)$$

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

$$T_{\text{DijkstraRadix}} = O(m + n \cdot (K + K)) = O(m + n \cdot \log C)$$

Lineare Laufzeit für zufällige Kantengewichte

Vorher gesehen: Dijkstra mit binary heaps hat lineare Laufzeit für dichte Graphen ($m > n \log n \log \log n$)

Letzte Folie: $T_{\text{DijkstraRadix}} = O(m + n \cdot \log C)$

Jetzt: Dijkstra mit Radix-Heaps hat lineare Laufzeit ($O(m + n)$) falls Kantenkosten identisch uniform verteilt in $0..C$

– wir brauchen nur eine kleine Änderung im Algorithmus

Änderung im Algorithmus für zufällige Kantengewichte

Vorberechnung von $c_{\min}^{\text{in}}(v) := \min \{c((u, v)) : (u, v) \in E\}$ leichtestes eingehendes Kantengewicht.

Beobachtung: $d[v] \leq \text{min} + c_{\min}^{\text{in}}(v)$

$\implies d[v] = \mu(v)$.

\implies schiebe v in Menge F ungescannter Knoten mit korrekter Distanz

Knoten in F werden bei nächster Gelegenheit gescannt.

($\approx F$ als Erweiterung von $B[-1]$.)

Analyse

Ein Knoten v kommt **nie** in einem Bucket i mit $i < \log c_{\min}^{\text{in}}(v)$

Also wird v höchstens $K + 1 - \log c_{\min}^{\text{in}}(v)$ mal verschoben

Kosten von Verschiebungen sind dann insgesamt höchstens

$$\sum_v (K - \log c_{\min}^{\text{in}}(v) + 1) = n + \sum_v (K - \log c_{\min}^{\text{in}}(v)) \leq n + \sum_e (K - \log c(e)).$$

$K - \log c(e)$ = Anzahl Nullen am Anfang der binären Repräsentation von $c(e)$ als K -Bit-Zahl.

$$\mathbb{P}(K - \log c(e) = i) = 2^{-i} \quad \Rightarrow \quad \mathbb{E}(K - \log c(e)) = \sum_{i \geq 0} i 2^{-i} \leq 2$$

$$\text{Laufzeit} = O(m + n)$$

All-Pairs Shortest Paths

Bis jetzt gab es immer einen bestimmten Anfangsknoten s

Wie können wir kürzeste Pfade für **alle** Paare (u, v) in G bestimmen?

Annahme: negative Kosten erlaubt, aber keine negativen **Kreise**

Lösung 1: n mal Bellman-Ford ausführen

... Laufzeit $O(n^2m)$

Lösung 2: **Knotenpotentiale**

... Laufzeit $O(nm + n^2 \log n)$, deutlich schneller

Knotenpotentiale

Jeder Knoten bekommt ein Potential $\text{pot}(v)$

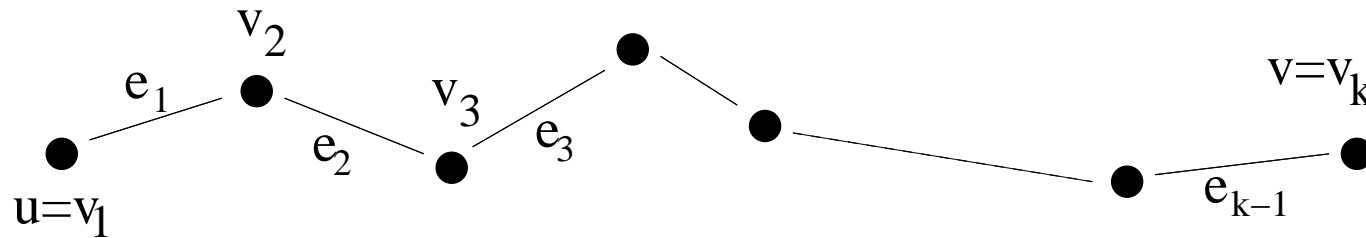
Mit Hilfe der Potentiale definieren wir **reduzierte Kosten** $\bar{c}(e)$ für Kante $e = (u, v)$ als

$$\bar{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v).$$

Mit diesen Kosten finden wir die **gleichen** kürzesten Pfade wie vorher!

Gilt für **alle** möglichen Potentiale – wir können sie also frei definieren

Knotenpotentiale



Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

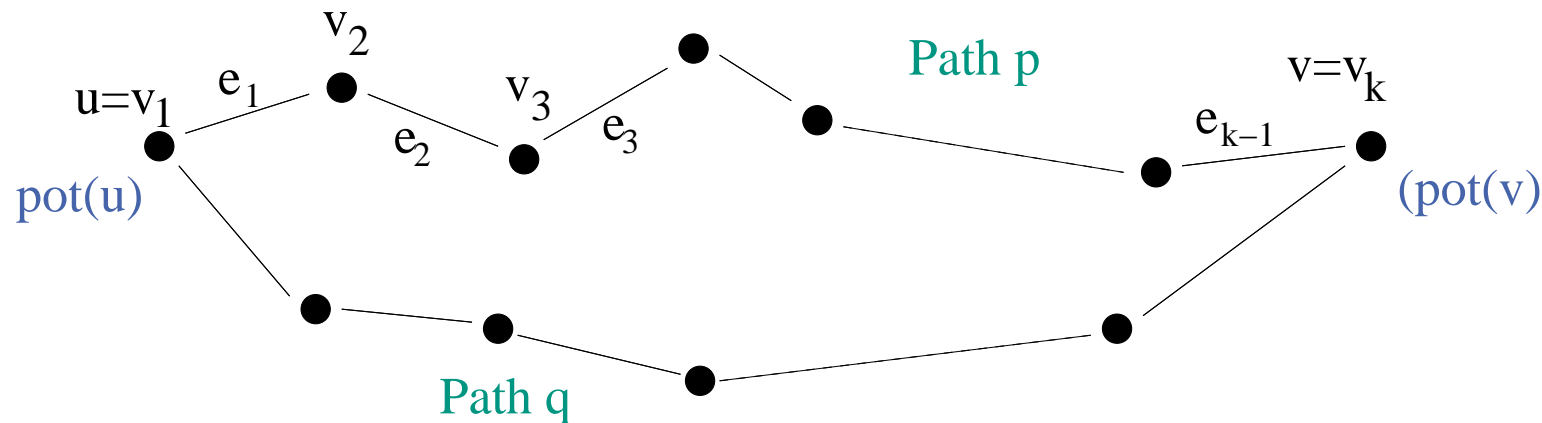
$$\begin{aligned}
 \bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) = \sum_{i=1}^{k-1} (\text{pot}(v_i) + c(e_i) - \text{pot}(v_{i+1})) \\
 &= \text{pot}(v_1) + \sum_{i=1}^{k-1} c(e_i) - \text{pot}(v_k) \\
 &= \text{pot}(v_1) + c(p) - \text{pot}(v_k).
 \end{aligned}$$

Knotenpotentiale

Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

$$\bar{c}(p) = \text{pot}(v_1) + c(p) - \text{pot}(v_k).$$

Sei q ein anderer u - v -Pfad, dann $c(p) \leq c(q) \Leftrightarrow \bar{c}(p) \leq \bar{c}(q)$.



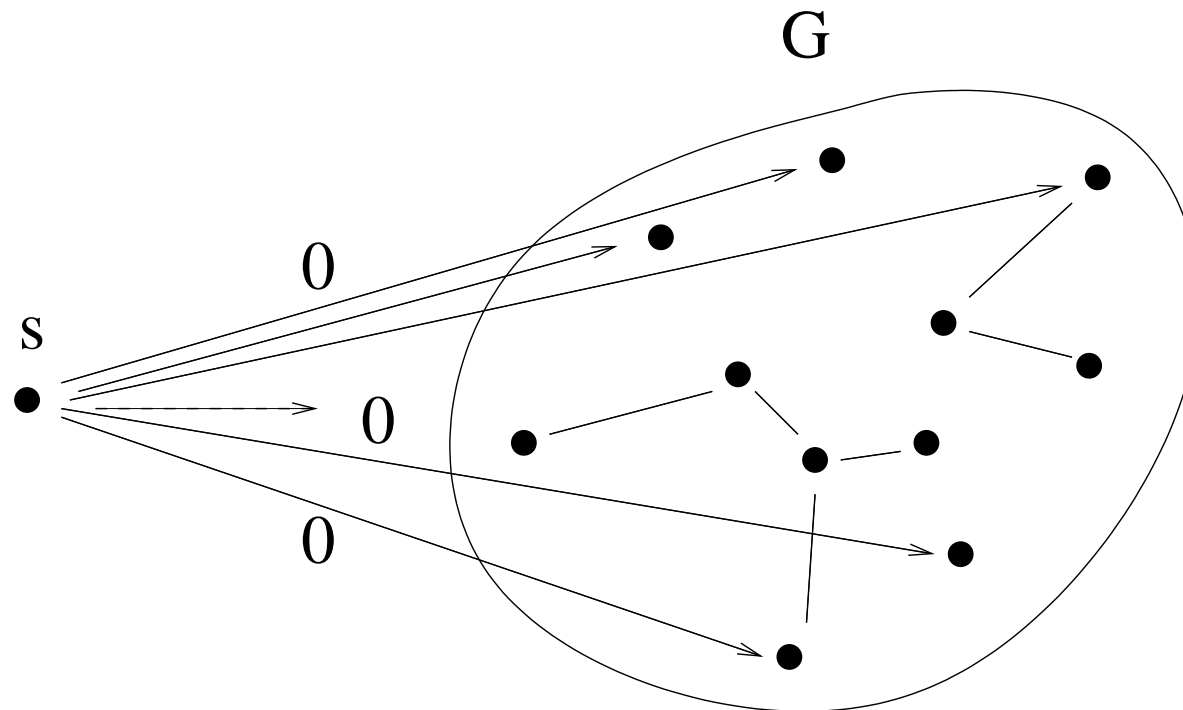
Definition: $\mu(u, v)$ = kürzeste Distanz von u nach v

Hilfsknoten

Wir fügen einen **Hilfsknoten s** an G hinzu

Für alle $v \in V$ fügen wir eine Kante (s, v) hinzu mit Kosten 0

Berechne kürzeste Pfade **von s aus** mit Bellman-Ford



Definition der Potentiale

Definiere $\text{pot}(v) := \mu(v)$ für alle $v \in V$

Jetzt sind die reduzierten Kosten alle **nicht negativ**: also können wir Dijkstra benutzen! (Evtl. s wieder entfernen...)

- Keine negativen Kreise, also $\text{pot}(v)$ wohldefiniert
- Für beliebige Kante (u, v) gilt

$$\mu(u) + c(e) \geq \mu(v)$$

deshalb

$$\bar{c}(e) = \mu(u) + c(e) - \mu(v) \geq 0$$

Algorithmus

All-Pairs Shortest Paths in the Absence of Negative Cycles

neuer Knoten s

foreach $v \in V$ **do** füge Kante (s, v) ein (Kosten 0) // $O(n)$

pot:= μ := BellmanFordSSSP(s, c) // $O(nm)$

foreach Knoten $x \in V$ **do** // $O(n(m + n \log n))$

$\bar{\mu}(x, \cdot)$:= DijkstraSSSP(x, \bar{c})

// zurück zur ursprünglichen Kostenfunktion

foreach $e = (v, w) \in V \times V$ **do** // $O(n^2)$

$\mu(v, w)$:= $\bar{\mu}(v, w) + \text{pot}(w) - \text{pot}(v)$

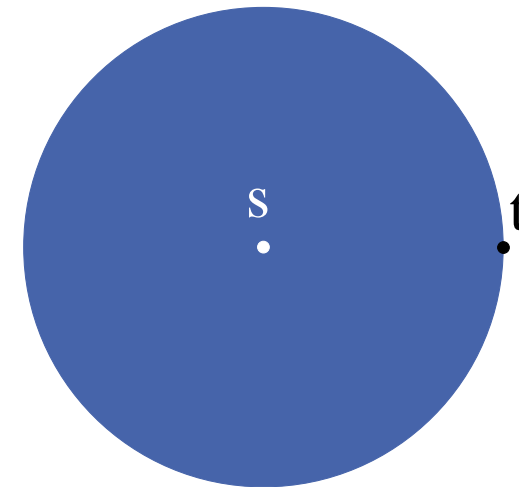
Laufzeit

- s hinzufügen: $O(n)$
- Postprocessing: $O(n^2)$ (zurück zu den ursprünglichen Kosten)
- n mal Dijkstra dominiert

$$\text{Laufzeit } O(n(m + n \log n)) = O(nm + n^2 \log n)$$

Distanz zu einem Zielknoten t

Was machen wir, wenn wir nur die Distanz von s zu einem bestimmten Knoten t wissen wollen?



Trick 0:

Dijkstra hört auf, wenn t aus Q entfernt wird

Spart "im Durchschnitt" Hälfte der Scans

Frage: Wieviel spart es (meist) beim Europa-Navi?



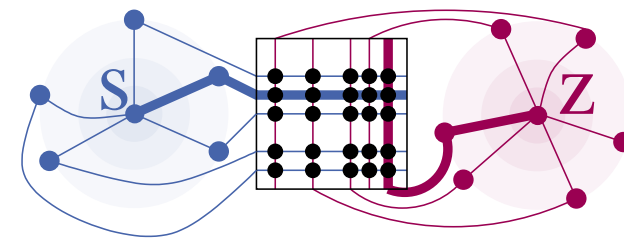
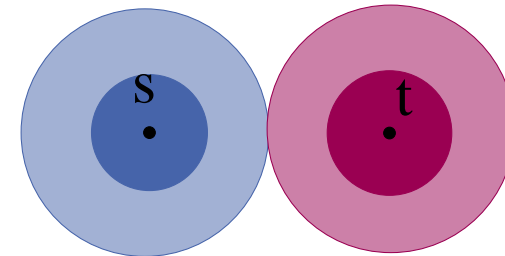
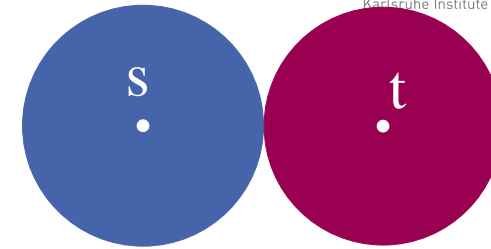
Ideen für Routenplanung

Vorwärts + Rückwärtsuche

Zielgerichtete Suche

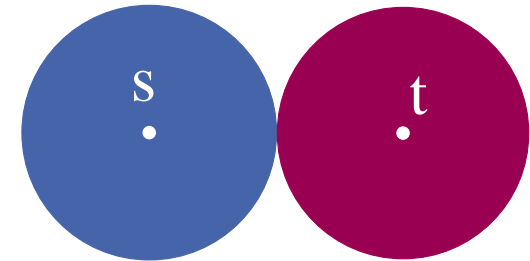
Hierarchien ausnutzen 

Teilabschnitte tabellieren 



Bidirektionale Suche

Idee: Suche abwechselnd s und t



Vorwärtssuche auf normalem Graph $G = (V, E)$

Rückwärtssuche auf Rückwärtsgraph $G^r = (V, E^r)$

(Suchrichtungen wechseln in jedem Schritt)

Vorläufige kürzeste Distanz wird in jedem Schritt gespeichert:

$$d[s, t] = \min(d[s, t], d_{\text{forward}}[u] + d_{\text{backward}}[u])$$

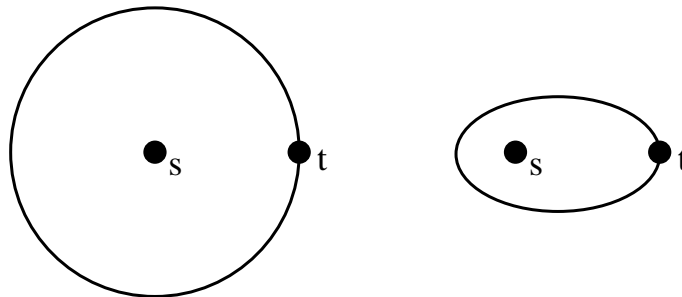
Abbruchkriterium:

Suche scannt Knoten, der in anderer Richtung bereits gescannt wurde.

$$d[s, t] \Rightarrow \mu(s, t)$$

A^* -Suche

Idee: suche “in die Richtung von t ”

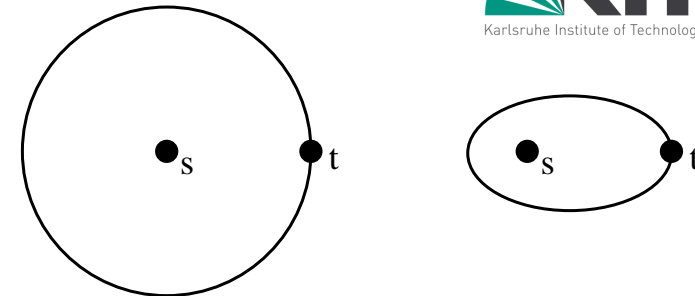


Annahme: Wir kennen eine Funktion $f(v)$ die $\mu(v, t)$ schätzt $\forall v$

Definiere $\text{pot}(v) = f(v)$ und $\bar{c}(u, v) = c(u, v) + f(v) - f(u)$

[Oder: in Dijkstra's Algorithmus, entferne nicht v mit minimalem $d[v]$ aus Q , sondern v mit minimalem $d[v] + f[v]$]

A*-Suche



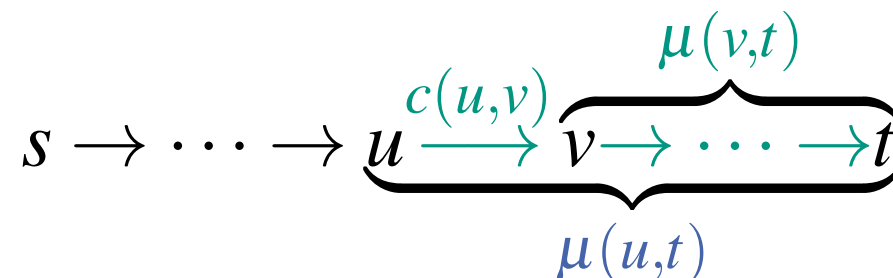
Idee: suche "in die Richtung von t "

Annahme: wir kennen eine Funktion $f(v)$ die $\mu(v, t)$ schätzt $\forall v$

Definiere $\text{pot}(v) = f(v)$ und $\bar{c}(u, v) = c(u, v) + f(v) - f(u)$

Beispiel: $f(v) = \mu(v, t)$.

Dann gilt: $\bar{c}(u, v) = c(u, v) + \mu(v, t) - \mu(u, t) = 0$ falls (u, v) auf dem kürzesten Pfad von s nach t liegt.



Also scannt Dijkstra nur die Knoten auf diesem Pfad!

Benötigte Eigenschaften von $f(v)$

- Konsistenz (reduzierte Kosten nicht negativ):

$$c(e) + f(v) \geq f(u) \quad \forall e = (u, v)$$

- $f(v) \leq \mu(v, t) \quad \forall v \in V$

- $f(t) = 0$ dann können wir aufhören, wenn t aus Q entfernt wird

Sei p irgendein Pfad von s nach t .

Alle Kanten auf p sind relaxiert? $\Rightarrow d[t] \leq c(p)$.

Sonst: $\exists v \in p \cap Q$, und $d[t] + f(t) \leq d[v] + f(v)$ weil t schon entfernt wurde. Deshalb

$$d[t] = d[t] + f(t) \leq d[v] + f(v) \leq d[v] + \mu(v, t) \leq c(p)$$

Wie finden wir $f(v)$?

Wir brauchen Heuristiken für $f(v)$.

Strecke im Straßennetzwerk: $f(v) = \text{euklidischer Abstand } ||v - t||_2$

bringt deutliche aber nicht überragende Beschleunigung

Fahrzeit: $\frac{||v - t||_2}{\text{Höchstgeschwindigkeit}}$

praktisch nutzlos

Noch besser aber mit Vorberechnung: **Landmarks**

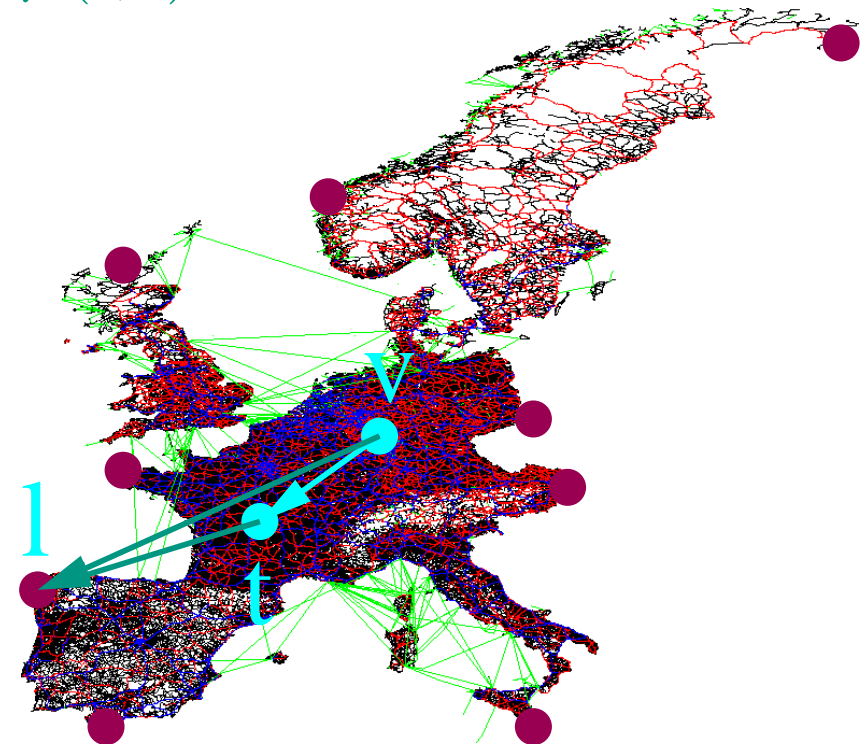
Landmarks [Goldberg Harrelson 2003]

Vorbereitung: Wähle Landmarkmenge L . $\forall \ell \in L, v \in V$
berechne/speichere $\mu(v, \ell)$.

Query: Suche Landmark $\ell \in L$ "hinter" dem Ziel.

Benutze untere Schranke $f_\ell(v) = \mu(v, \ell) - \mu(t, \ell)$

- + Konzeptuell einfach
- + Erhebliche Beschleunigungen
(\approx Faktor 20 im Mittel)
- + Kombinierbar mit anderen Techniken
- Landmarkauswahl kompliziert
- hoher Platzverbrauch



Zusammenfassung Kürzeste Wege

- Nichttriviale Beispiele für Analyse im Mittel. Ähnlich für MST
- Monotone, ganzzahlige Prioritätslisten als Beispiel wie Datenstrukturen auf Algorithmus angepasst werden
- Knotenpotentiale allgemein nützlich in Graphenalgorithmien
- Aktuelle Forschung trifft klassische Algorithmik