Shortest Paths

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

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

Anfangsknoten s

Frage: was ist die Distanz von s zu v (für alle anderen Knoten $v \in V$)

Fundamentales Problem

Wie komme ich am schnellsten zur Uni?

Gibt es immer einen kürzesten Pfad?

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

Kostenfunktion $c : E \to \mathbb{R}$

Anfangsknoten s

Frage: was ist die Distanz von s zu v (für alle anderen Knoten $v \in V$)

Gibt es immer einen kürzesten Pfad?

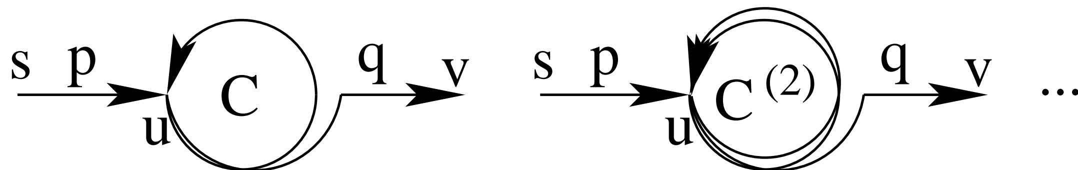
Eingabe: Graph $G = (V, E)$

Kostenfunktion $c : E \to \mathbb{R}$

Anfangsknoten s

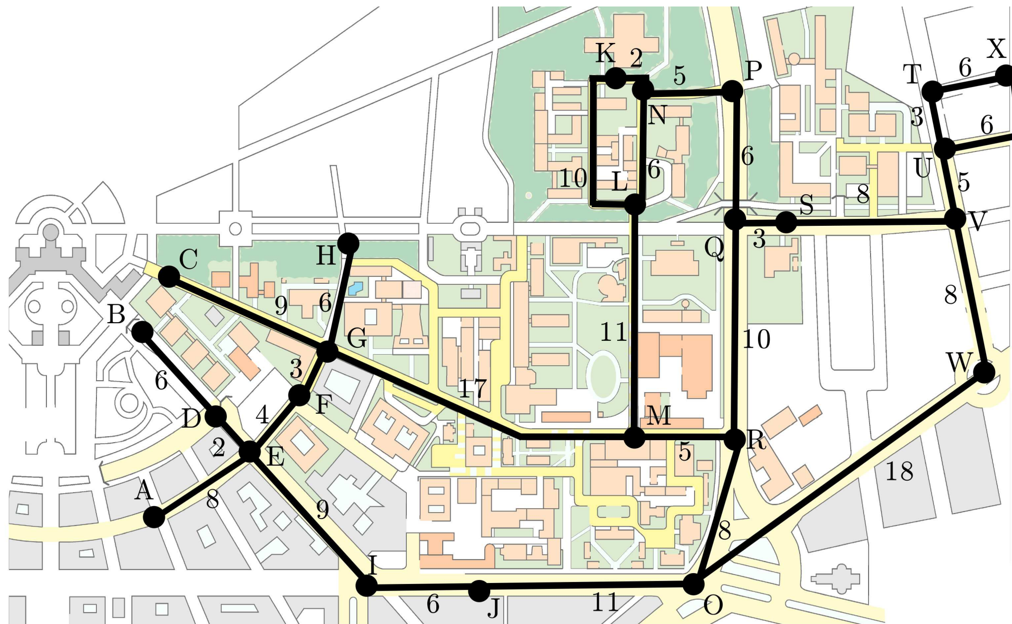
Frage: was ist die Distanz von s zu v (für alle anderen Knoten $v \in V$)

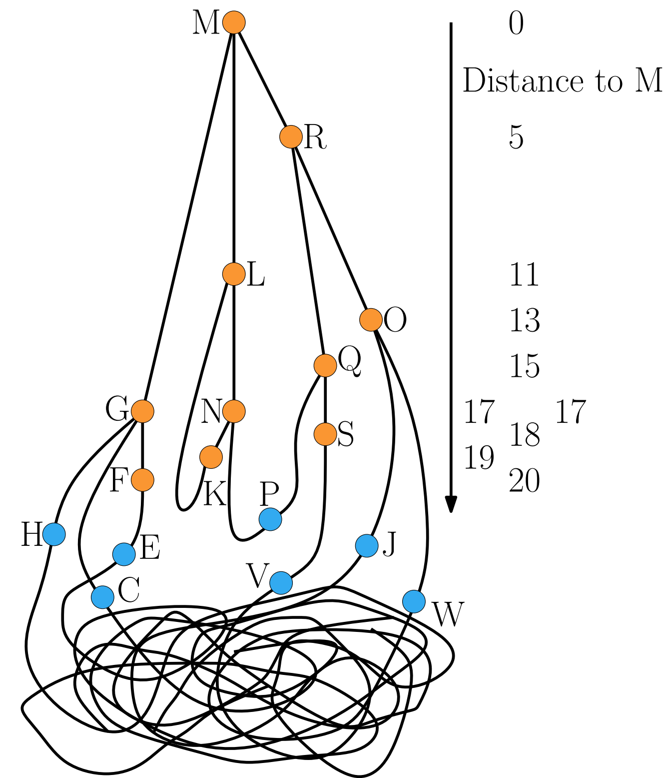
Es kann negative Kreise geben!



Lösung ohne Rechner

Vorläufige Annahme: alle Kosten sind mindestens 0





Ein einfacher Fall

Alle Kosten gleich, dann Breitensuche (BFS)

Findet erst die Knoten mit Distanz 1, dann 2, ...

BFS gibt die Lösung

Allgemeine Kostenfunktion: Dijkstra's Algorithmus

Edsger Wybe Dijkstra

1972 ACM Turingpreis

THE: das erste Mulitasking-OS

Semaphor

Selbst-stabilisierende Systeme

GOTO Statement Considered Harmful

Allgemeine Definitionen

Wir benutzen zwei Knotenarrays:

- $d[v]$ = aktuelle (vorläufige) Distanz von s nach v
- $parent[v]$ = Vorgänger von v auf dem (vorläufigen) kürzesten Pfad von s nach v

Initialisierung: $d[s] = 0$, $parent[s] = s$

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

Relaxierung einer Kante

Wir werden den Graphen traversieren und dabei ständig Distanzen aktualisieren.

Falls es eine Kante (u, v) gibt

und $d[u] + c(e) < d[v]$ (vielleicht $d[v] = \infty$)

können wir die Distanz von s nach v jetzt besser abschätzen.

Wir setzen $d[v] := d[u] + c(e)$ und $parent[v] := u$

Kann sich mehrmals ändern!

Dijkstra’s Algorithmus: Pseudocode

Alle Knoten sind ungescannt, d und *parent* initialisiert

So lange es noch ungescannte Knoten gibt mit endlicher vorläufiger Distanz:

> $u :=$ ungescannter Knoten mit minimaler vorläufiger Distanz
>
> Relaxiere alle von u ausgehenden Kanten (u, v)
>
> u ist jetzt gescannt

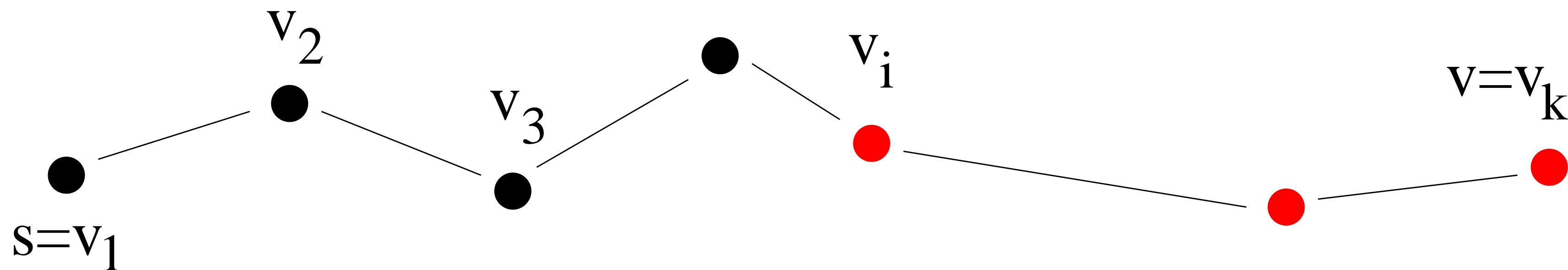
Korrektheit

Annahme: alle Kosten nicht negativ!

Def. $\mu(v) =$ (tatsächliche) Distanz von s nach v

- Falls v erreichbar ist, wird er irgendwann gescannt
- Wenn ein Knoten v gescannt wird, dann $\mu(v) = d[v]$

Beweise: betrachte jeweils den kürzesten Pfad von s nach v (das gibt es, falls v erreichbar ist!)



Implementierung?

Alle Knoten sind ungescannt, d und *parent* initialisiert

So lange es noch ungescannte Knoten gibt mit endliche vorläufige Distanz:

$u :=$ ungescannter Knoten mit minimale vorläufige Distanz

Relaxiere alle Kanten (u,v) aus u

u ist jetzt gescannt

Die wichtigste Operation im Algorithmus ist: finde u

Adressierbare Prioritätsliste

Wir speichern ungescannte erreichte Knoten in Prioritätsliste Q

Schlüssel von v in Q ist $d[v]$

Wir haben auch ein Knotenarray A mit den Handles zu den Elementen in Q

$A[v] = \text{nil}$ falls $v \notin Q$

Prioritätsliste Q plus Knotenarray A nennen wir NodePQ (Knotenprioritätsliste)

NodePQ

Procedure build($\{v_1,\ldots,v_n\}$) $M := \{v_1,\ldots,v_n\}$

Function size **return** $|M|$

Procedure insert(v) fügt v ein

Function min **return** min M

Function deleteMin findet v mit minimalen $d[v]$, entfernt v,
$A[v] :=$ nil, **return** v

Function remove(h : Handle) $v := h$; $M := M\setminus\{v\}$; **return** v

Procedure decreaseKey(v) **assert** key$(v) \geq d[v]$;

Finde v mit Hilfe von $A[v]$, key$(v) := d[v]$

Procedure merge(M') $M := M \cup M'$

Implementierung

Function Dijkstra(s : NodeId) : NodeArray×NodeArray

// returns (d, parent)

Initialisierung:

$d = \langle\infty, \ldots, \infty\rangle$: NodeArray **of** $\mathbb{R} \cup \{\infty\}$

// tentative distance from root

parent$= \langle\bot, \ldots, \bot\rangle$: NodeArray **of** NodeId

parent$[s]$:= s // self-loop signals root

Q : NodePQ // unscanned reached nodes

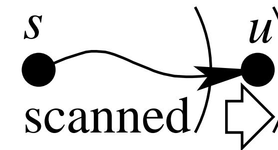
$d[s] := 0$; Q.insert(s)

Initialisierung:

$d = \{\infty, \ldots, \infty\}$; parent$[s]$:= s; $d[s]$:= 0; Q.insert(s)

while $Q \neq \emptyset$ **do**

  u := Q.deleteMin   // we have $d[u] = \mu(u)$



  foreach edge $e = (u, v) \in E$ **do**

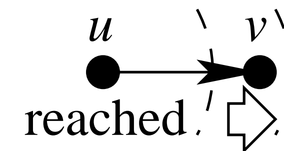
    if $d[u] + c(e) < d[v]$ **then**   // relax

      $d[v]$:= $d[u] + c(e)$

      parent$[v]$:= u // update tree

      if $v \in Q$ **then** Q.decreaseKey(v)

      else Q.insert(v)



return $(d$, parent$)$

Laufzeit

Function Dijkstra(s : NodeId) : NodeArray×NodeArray

// returns (d, parent)

Initialisierung:

$d = \langle \infty, \ldots, \infty \rangle$: NodeArray **of** $\mathbb{R} \cup \{\infty\}$ // $O(n)$

// tentative distance from root

$\text{parent} = \langle \bot, \ldots, \bot \rangle$: NodeArray **of** NodeId // $O(n)$

$\text{parent}[s] := s$ // self-loop signals root

Q : NodePQ // unscanned reached nodes, $O(n)$

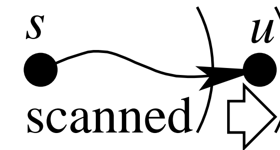
$d[s] := 0; \quad Q.\text{insert}(s)$

Initialisierung:

$d = \{\infty, \ldots, \infty\}$; parent$[s]$:= s; $d[s]$:= 0; Q.insert(s)

while $Q \neq \emptyset$ **do**

u := Q.deleteMin // $\leq$einmal pro Knoten



foreach edge $e = (u, v) \in E$ **do**

if $d[u] + c(e) < d[v]$ **then** // relax

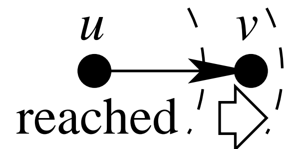
$d[v]$:= $d[u] + c(e)$

parent$[v]$:= u // update tree

if $v \in Q$ **then** Q.decreaseKey(v)

// $\leq$ einmal pro Kante

else Q.insert(v) $\leq$ einmal pro Knoten



return (d, parent)

Laufzeit

Insgesamt finden wir

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

Laufzeit

Implementierung aus dem Jahr 1959:

- Array d für Distanzen
- Array das für jeden Knoten angibt, ob er gescannt oder erreicht worden ist

insert $\mathrm{O}(1)$, decreaseKey $\mathrm{O}(1)$, but deleteMin $O(n)$

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}(m \cdot T_{\mathsf{decreaseKey}}(n) + n \cdot (T_{\mathsf{deleteMin}}(n) + T_{\mathsf{insert}}(n))) \\
T_{\mathsf{Dijkstra59}} &= \mathrm{O}(m \cdot 1 + n \cdot (n+1)) \\
&= \mathrm{O}\left(m + n^2\right)
\end{aligned}
$$

Laufzeit

Bessere Implementierung mit Binary-Heapprioritätslisten:

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

$$
\begin{aligned}
T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{DijkstraBHeap}} &= O(m \cdot \log n + n \cdot (\log n + 1)) \\
&= O((m+n)\log n)
\end{aligned}
$$

Laufzeit

(Noch) besser mit Fibonacci-Heapprioritätslisten:

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

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

Aber: konstante Faktoren in $O(\cdot)$ sind hier größer!

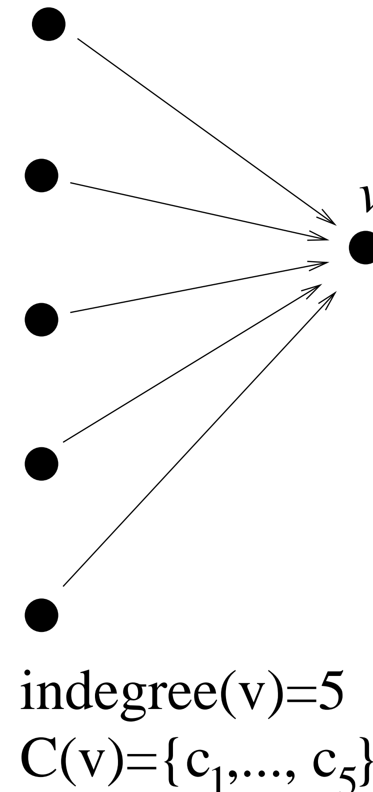
Laufzeit im Durchschnitt

Bis jetzt: decreaseKey wird 'höchtens m mal ausgeführt' (höchstens einmal pro Kante)

Wie oft wird diese Operation im Durchschnitt ausgeführt?

Modell:

- Beliebiger Graph G
- Beliebiger Anfangsknoten $s \in V_G$
- Beliebige Mengen $C(v)$ von nonnegativen reellen Zahlen für jeden Knoten v; $|C(v)| = indegree(v)$



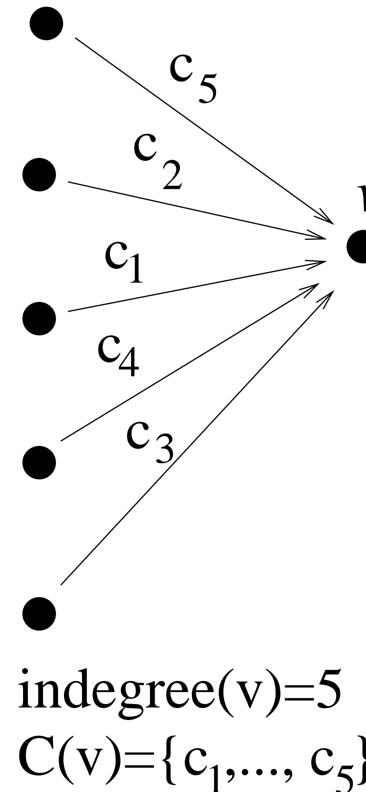
Laufzeit im Durchschnitt

Wir machen eine probabilistische Analyse

Annahme: für jedes v werden die Kosten in $C(v)$ willkürlich an eingehende Kanten zugewiesen

Sehr allgemeines Modell

Beispiel: alle Kosten unabhängig identisch verteilt



Laufzeit im Durchschnitt

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Dann

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

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

($T_{\mathsf{DijkstraFib}} = \mathrm{O}\left(m + n \log \frac{m}{n} + n \log n\right) = \mathrm{O}(m + n \log m)$ in Durchschnitt)

Lineare Laufzeit für dichte Graphen

Wie groß sollte m sein, damit Dijkstra lineare Laufzeit hat?

Größe der Eingabe = $\mathrm{O}(m+n)$.

Wir brauchen $n(\log\frac{m}{n}\log n) = \mathrm{O}(m+n)$.

Wir können zeigen: $m = \Omega(n\log n\log\log n)$ reicht, dann Laufzeit $= \mathrm{O}(m)$.

Wit benutzen dann in der Praxis lieber Binary Heaps statt Fibonacci-Heapprioritätslisten wegen kleineren Konstanten in $\mathrm{O}(\cdot)$.

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Betrachten wir einen Knoten v.
Bei Bearbeitung welcher Kanten wird decreaseKey(v) ausgeführt?

$$e_i := (u_i, v) \qquad \mu(u_i) \leq \mu(v)$$

Nummerieren wir diese Kanten $e_1, \ldots, e_k$.
Reihenfolge = Folge der Scans der u_i. Dann

$$\mu(u_1) \leq \mu(u_2) \leq \cdots \leq \mu(u_k) \leq \mu(v)$$

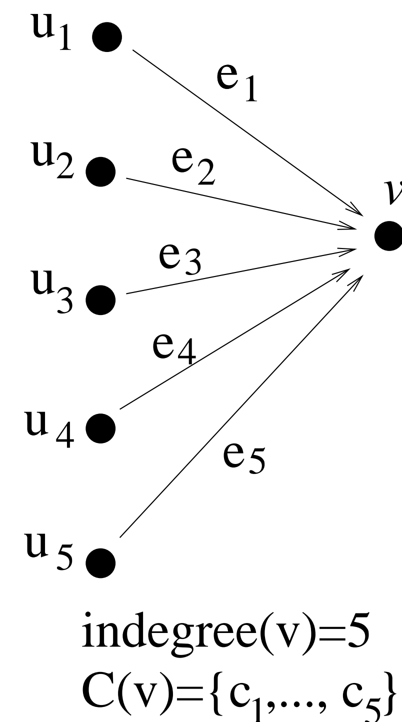
Relaxierung der Kanten wird in diese Reihenfolge vorgenommen – unabhängig von $C(v)$!

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Betrachten wir einen Knoten v.

Bei Bearbeitung welcher Kanten wird decreaseKey(v) ausgeführt?

$$e_i := (u_i, v) \qquad \mu(u_i) \leq \mu(v)$$



Nummerieren wir diese Kanten $e_1, \ldots, e_k$. Reihenfolge = Folge der Scans der u_i. Dann

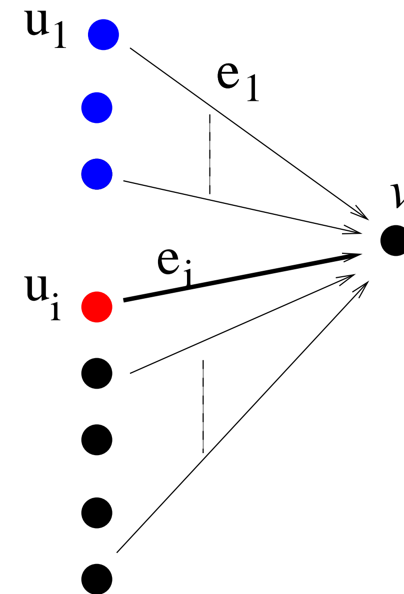
$$\mu(u_1) \leq \mu(u_2) \leq \cdots \leq \mu(u_k) \leq \mu(v)$$

Relaxierung der Kanten wird in dieser Reihenfolge vorgenommen – unabhängig von $C(v)$!

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Während der Bearbeitung von e_i wird decreaseKey nur ausgeführt falls

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



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

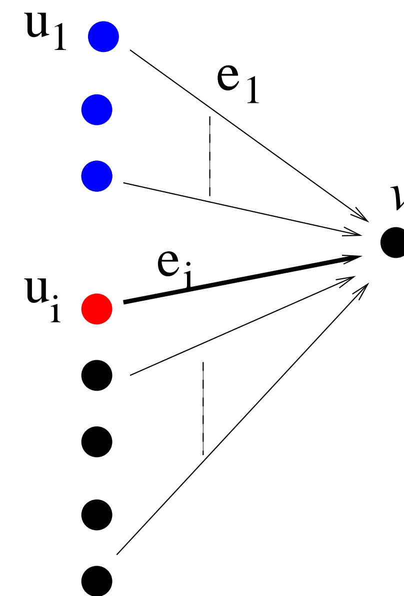
$$c(e_i) < \min_{j<i} c(e_j).$$

decreaseKey wird nur ausgeführt wenn $c(e_i)$ kleiner ist als alle vorherigen Kosten $c(e_1), \ldots, c(e_{i-1})$.

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Während der Bearbeitung von e_i wird decreaseKey nur ausgeführt falls

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



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

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

decreaseKey wird nur ausgeführt wenn $c(e_i)$ kleiner ist als alle vorherigen Kosten $c(e_1), \ldots, c(e_{i-1})$.

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Unsere Annahme war: für jedes v werden die Kosten in $C(v)$ willkürlich an eingehende Kanten zugewiesen

Wie oft findet man ein neues Minimum wenn man diese Sequenz durchläuft?

Erwartete Anzahl ist höchstens H_k (Sect. 2.8).

Das erste gefundene Minimum zählt aber nicht mit! Es verursacht *insert*(v) aber kein *decreaseKey*(v).

Also wird *decreaseKey* höchstens $H_k - 1 \leq \ln k$ mal ausgeführt (im Erwartungswert)

Satz 1. $\mathbb{E}$*(Anzahl decreaseKey-Operationen)*$= \mathrm{O}\left(n \log \frac{m}{n}\right)$

Für Knoten v wird decreaseKey höchstens $H_k - 1 \leq \ln k$ mal ausgeführt (im Erwartungswert).

Hier ist k die Anzahl der eingehenden Kanten für die decreaseKey ausgeführt werden kann.

Also $k \leq indegree(v)$.

Insgesamt wird decreaseKey dann

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

mal ausgeführt.

Monotone Integer Prioritätslisten

Wir haben gesehen: mit der richtigen Implementierung hat Dijkstra lineare Laufzeit für Graphen die dicht genug sind

Kann es noch besser?

Wir brauchen eigentlich keine allgemeine Prioritätslisten

Dijkstra's Algorithmus benutzt die Liste monoton:
Operationen *insert* und *decreaseKey* benutzen Distanzen der Form $d[u]+c(e)$

Dieser Wert nimmt ständig zu

Monotone Integer Prioritätslisten

Annahme: Alle Kosten sind ganzzahlig und im Interval $[0, C]$

Die größt mögliche Distanz zwischen zwei Knoten im Graphen G ist dann $(n-1)C$

Sei min der letzte Wert, der aus Q entfernt wurde

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

Bucket-Queues

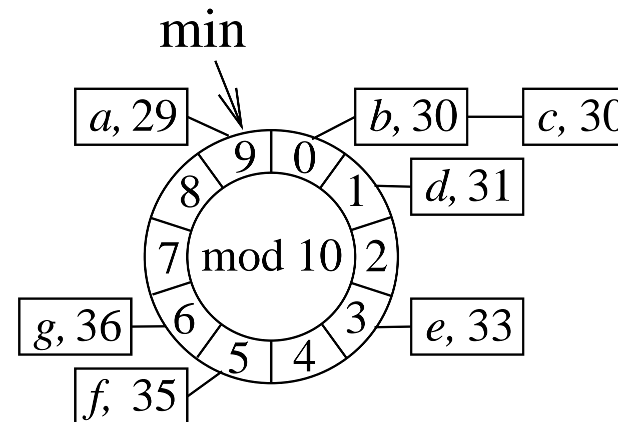
Eine Bucket-Queue ist ein kreisförmiges Array B von $C+1$ doppelt gelinkten Listen

Ein Knoten mit aktuelle Distanz $d[v]$ wird gespeichert bei Index

$$d[v] \quad \bmod (C+1)$$

Alle Knoten im gleichen Bucket haben die gleiche Distanz $d[v]$!

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 $B[d[v] \mod (C+1)]$ ein $O(1)$

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

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

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

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

Laufzeit Dijkstra mit Bucket-Queues

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathcal{O}(m \cdot T_{\mathsf{decreaseKey}}(n) \\
&\quad + \text{Kosten deleteMin-Operationen} \\
&\quad + n \cdot T_{\mathsf{insert}}(n))) \\
T_{\mathsf{DijkstraBQ}} &= \mathrm{O}(m \cdot 1 + nC + n \cdot 1)) \\
&= \mathrm{O}(m + nC)
\end{aligned}
$$

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

Idee: nicht alle Buckets gleich groß machen

Radix-Heaps

Wir benutzen $K = 1 + \lfloor \log C \rfloor$ buckets

min = die zuletzt aus Q entfernte Distanz

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

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 $\forall v \in Q : d[v] \in [11101, 101000]$

Speichere v in Bucket $B[i]$ falls $d[v]$ und min sich zuerst unterscheiden an der $i$ten Stelle, **oder** in Bucket $B[K]$

Definition $msd(a,b)$

Der signifikanteste unterschiedliche Index von a und b
= größte Index für die a und b unterschiedlich sind in der binären Schreibweise

a	11001010	1010100	1110110
b	11000101	1010110	1110110
$\text{msd}(a,b)$	3	1	-1

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

Bucket-Queue-Invariante

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

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

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)

Bucket-Queue-Invariante

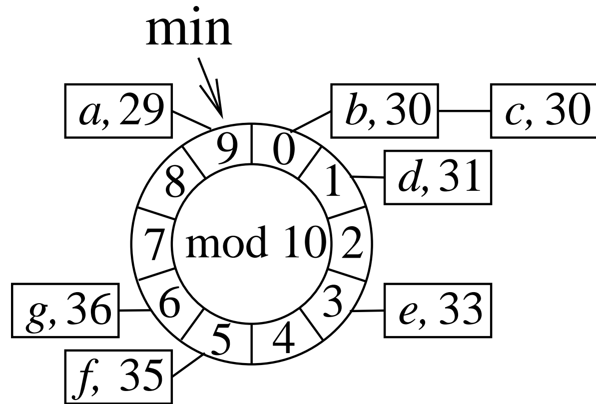
v ist gespeichert in Bucket $B[i]$ wo $i = \min(msd(min, d[v]), K)$.

Beispiel 2: $min = 11101$, $C = 9$, 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

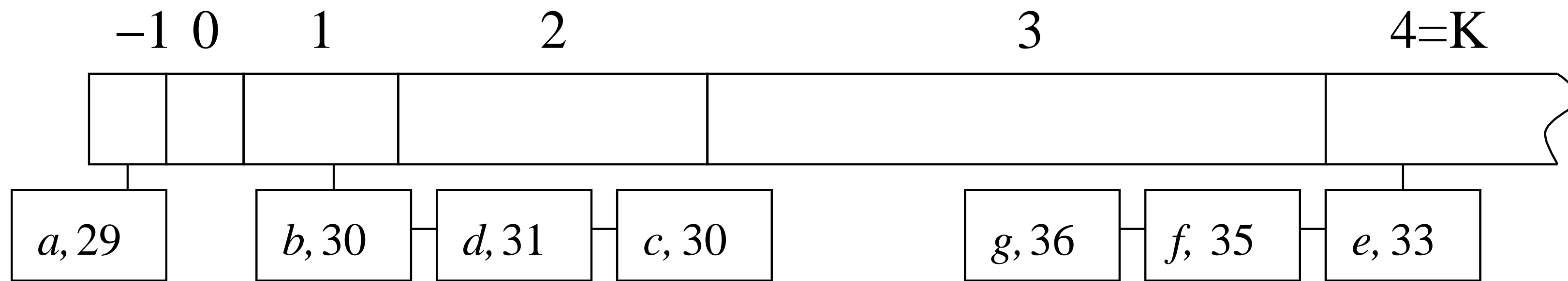
Falls $d[v] \geq 32$, dann $msd(min, d[v]) > 4$!

Bucket-Queues



Bucket queue with C = 9

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



Binary Radix Heap

Die deleteMin-Operation

Falls *min* geändert wird, müssen wir Knoten bewegen, damit die Invariante wieder gilt.

deleteMin sucht die kleinste i für die $B[i]$ nicht leer ist

In $B[0],\ldots,B[i-1]$ gibt es also keine Knoten und wir müssen nichts machen

Die Knoten aus $B[i+1],\ldots,B[K]$ bleiben wo sie sind!

Grund: neues *min* war in $B[i]$, also sind das alte und das neue *min* **gleich** für alle Bits $j>i$

Buckets $j > i$ bei Änderung von min

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

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	10001	17	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*

Die Knoten aus $B[i]$ gehen zu Buckets mit kleineren Indices

Nehme an: $i < K$ (es gilt auch für $i = K$)

Alle Knoten in $B[i]$ haben eine 1 als $i$tes Bit–sonst wären sie kleiner als *min*

Alle Knoten in $B[i]$ sind **gleich** für alle Bits $j > i$

Also ist das signifikanteste bit wo ein Knoten in $B[i]$ sich unterscheidet vom neuen *min* kleiner als i

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 Knotens

Laufzeit Dijkstra mit Radix-Heaps

Insgesamt finden wir amortisiert

- $T_{\mathsf{insert}}(n) = \mathrm{O}(K)$
- $T_{\mathsf{deleteMin}}(n) = \mathrm{O}(1)$
- $T_{\mathsf{decreaseKey}}(n) = \mathrm{O}(1)$

$$
\begin{aligned}
T_{\mathrm{Dijkstra}} &= \mathrm{O}(m \cdot T_{\mathsf{decreaseKey}}(n) + n \cdot (T_{\mathsf{deleteMin}}(n) + T_{\mathsf{insert}}(n))) \\
T_{\mathsf{DijkstraRadix}} &= \mathrm{O}(m + n \cdot (K + K)) = \mathrm{O}(m + n \cdot \log C)
\end{aligned}
$$

Lineare Laufzeit für random Graphen

Vorher gesehen: Dijkstra mit Bucket-Queues hat lineare Laufzeit für dichte Graphen ($m > n \log n \log\log n$)

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

Jetzt: Dijkstra mit Radix-Heaps hat lineare Laufzeit ($\mathrm{O}(m+n)$) falls Kantenkosten identisch uniform verteilt in $0..C$
–wir brauchen nur eine kleine Änderung im Algorithmus

Änderung im Algorithmus für random Graphen

Für jeden Knoten v speichern wir $c^{in}_{\min}(v) =$ Minimumkosten einer eingehenden Kante

Falls v von einem Bucket zum anderen geht, und $d[v] \leq min + c^{in}_{\min}(v)$, dann sind wir fertig mit diesem Knoten

Wir können diesen Knoten dann komplett aus den Buckets entfernen–spart später Zeit!

Wir brauchen nur eine neue Menge F von ungescannten Knoten mit korrekten Distanzen

Berechnung

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

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

Kosten von Verschiebungen sind dann insgesamt höchstens

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

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

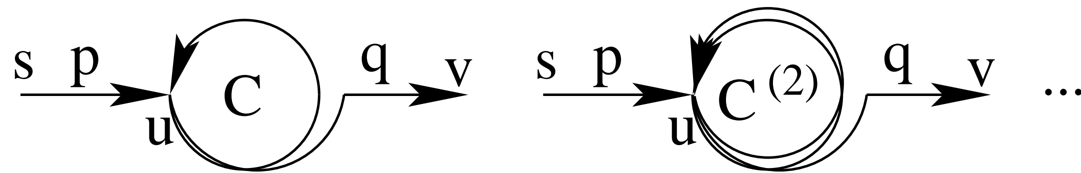
$\mathbb{P}(K - \log c(e) = i) = 2^{-i} \quad : \quad \mathbb{E}(K - \log c(e)) = \sum_{i \ge 0} i 2^{-i} = 1$

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

Negative Kosten

Was machen wir wenn es Kanten mit negativen Kosten gibt?

Es kann Knoten geben mit $d[v] = -\infty$



Wie finden wir heraus, welche das sind?

Endlosschleifen vermeiden!

Idee: alle Kanten $n-1$ mal relaxieren

Relaxierung einer Kante

Falls es eine Kante (u,v) gibt

und $d[u]+c(e)<d[v]$ (vielleicht $d[v]=\infty$)

können wir die Distanz von s nach v jetzt besser abschätzen.

Wir setzen $d[v]:=d[u]+c(e)$ und $\mathit{parent}[v]:=u$

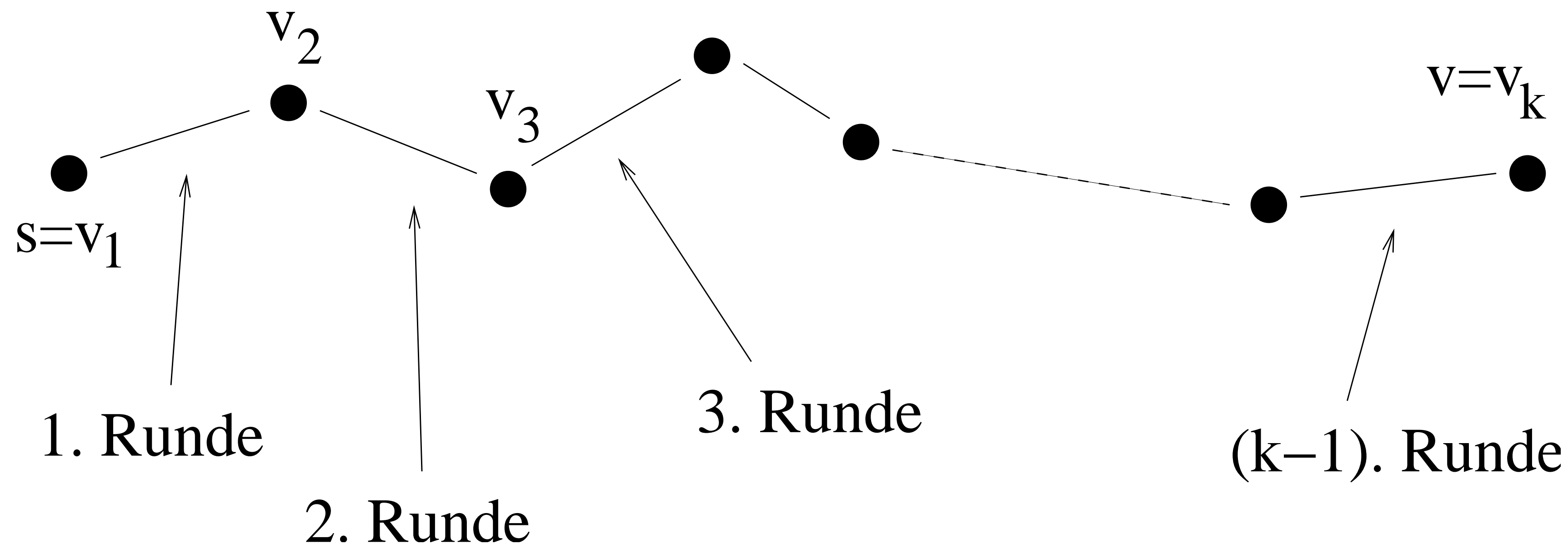
Kann sich mehrmals ändern!

Bellman-Ford-Algorithmus

Wir relaxieren alle Kanten (in irgendeiner Reihenfolge) $n-1$ mal

Alle kürzeste Pfade in G haben höchstens $n-1$ Kanten

Jeder kürzeste Pfad ist eine Teilfolge dieser Relaxierungen!



Bellman-Ford-Algorithmus

Nach $n-1$ Relaxierungen haben wir $d[v] = \mu(v)$ für alle Knoten v für die $-\infty < d[v] < \infty$; auch $parent[v]$ stimmt

Für bestimmte Knoten v gibt es keinen kürzesten Pfad von s nach v

Welche sind das?

Genau die Knoten v, für die es noch eine Kante (u,v) gibt die relaxiert werden könnte

Wenn wir nach $n-1$ Runden noch neue Relaxierungen finden für v, gilt $d[v] = -\infty$

Infizierung

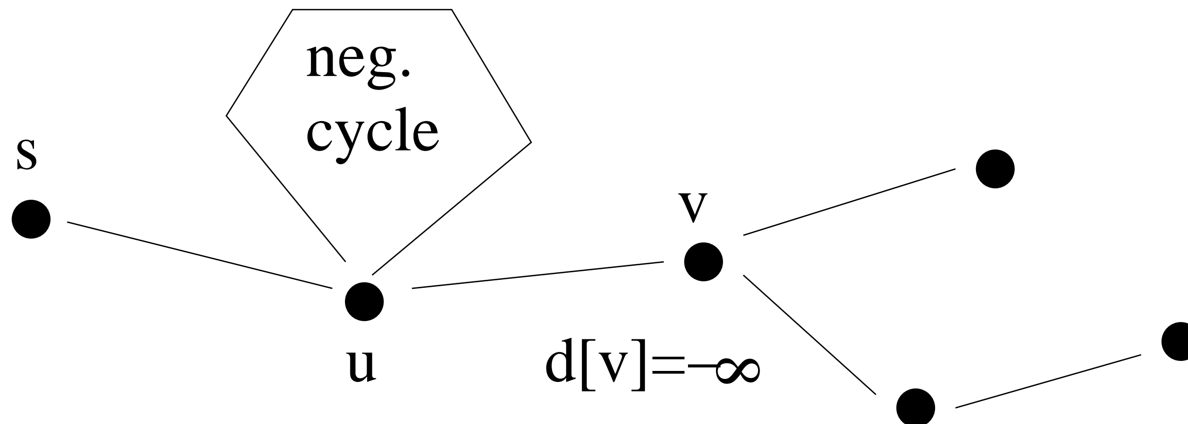
Wir besuchen noch einmal alle Kanten $e = (u, v) \in E$

Falls $d[u] + c(e) < d[v]$, dann wissen wir $d[v] = -\infty$

Das gilt dann auch für alle aus v erreichbare Knoten

Die können wir mit Hilfe von DFS finden (Rekursion)

Diesen Prozess nennen wir *Infizierung* der Knoten



Infizierung

Wir besuchen noch einmal alle Kanten $e = (u, v) \in E$

Falls $d[u] + c(e) < d[v]$, dann wissen wir $d[v] = -\infty$

Das gilt dann auch für alle aus v erreichbare Knoten

Die können wir mit Hilfe von DFS finden (Rekursion)

Diesen Prozess nennen wir *Infizierung* der Knoten

Procedure infect(v)

  if $d[v] > -\infty$ **then**

    $d[v] := -\infty$

    foreach $(v, w) \in E$ **do** infect(w)

Function BellmanFord(s : NodeId) : NodeArray×NodeArray

  $d=\langle\infty,\ldots,\infty\rangle$: NodeArray **of** $\mathbb{R}\cup\{-\infty,\infty\}$ // distance from root

  parent$=\langle\bot,\ldots,\bot\rangle$: NodeArray **of** NodeId

  $d[s]:=0$; parent$[s]:=s$ // self-loop signals root

  for $i:=1$ **to** $n-1$ **do**

    forall $e\in E$ **do** relax(e) // round i

  forall $e=(u,v)\in E$ **do** // postprocessing

    invariant $\forall v\in V : d[v]=-\infty$

      $\rightarrow\forall w$ reachable from $v : d[w]=-\infty$

    if $d[u]+c(e)<d[v]$ **then** infect(v)

  return (d,parent)

Bellman-Ford-Algorithmus

Am Ende haben wir drei Kategorien von Knoten:

- $d[v] = \infty$: v ist nicht von s aus erreichbar
- $d[v] = -\infty$: v ist erreichbar über einen negativen Kreis
- $d[v] = \mu(v)$: kürzester Pfad von s nach v wurde vollständig relaxiert

Laufzeit: $O(n \cdot m)$

- n mal alle Kanten relaxieren
- Postprocessing in Zeit $O(m)$

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 negative Kreise

Lösung 1: n mal Bellman-Ford ausführen

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

Lösung 2: Knotenpotentiale

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

Knotenpotentiale

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

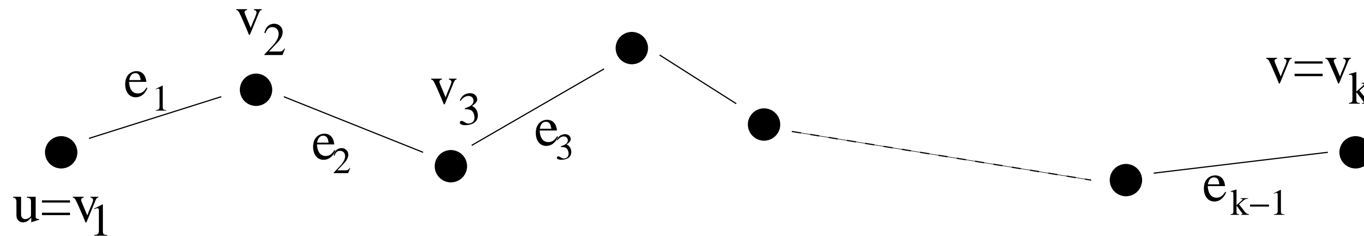
Mit Hilfe der Potentiale definieren wir reduzierte Kosten $\bar{c}(e)$ für Kante $e = (u, v)$ als

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

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

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

Knotenpotentiale

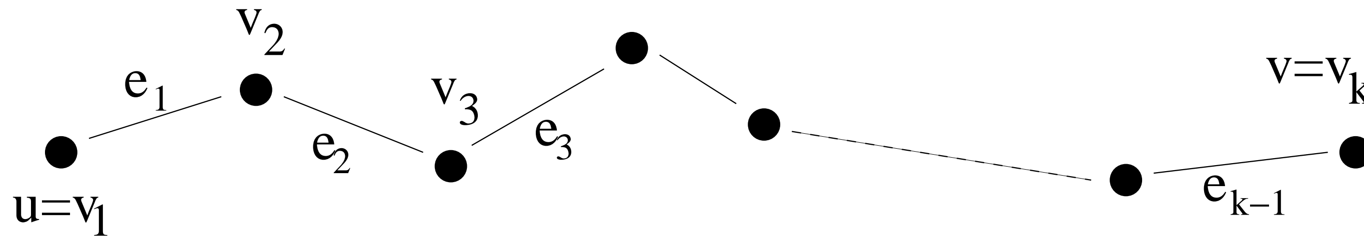


Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

$$\begin{aligned}
\bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(e_i) = \sum_{0 \le i < k} (\mathrm{pot}(v_i) + c(e_i) - \mathrm{pot}(v_{i+1})) \\
&= \mathrm{pot}(v_0) + \sum_{0 \le i < k} c(e_i) - \mathrm{pot}(v_k) \\
&= \mathrm{pot}(v_0) + c(p) - \mathrm{pot}(v_k).
\end{aligned}$$

Sei q ein anderer Pfad, dann $c(p) \le c(q) \Leftrightarrow \bar{c}(p) \le \bar{c}(q)$.

Knotenpotentiale

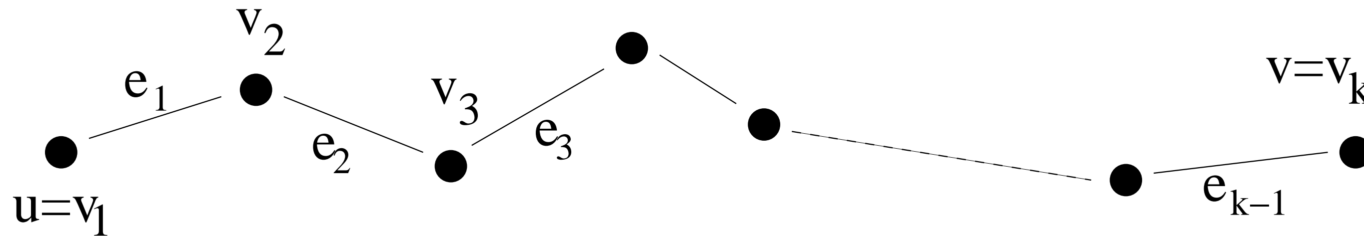


Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

$$\begin{aligned}
\bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(e_i) = \sum_{0 \le i < k} (\mathrm{pot}(v_i) + c(e_i) - \mathrm{pot}(v_{i+1})) \\
&= \mathrm{pot}(v_0) + \sum_{0 \le i < k} c(e_i) - \mathrm{pot}(v_k) \\
&= \mathrm{pot}(v_0) + c(p) - \mathrm{pot}(v_k).
\end{aligned}$$

Sei q ein anderer Pfad, dann $c(p) \le c(q) \Leftrightarrow \bar{c}(p) \le \bar{c}(q)$.

Knotenpotentiale

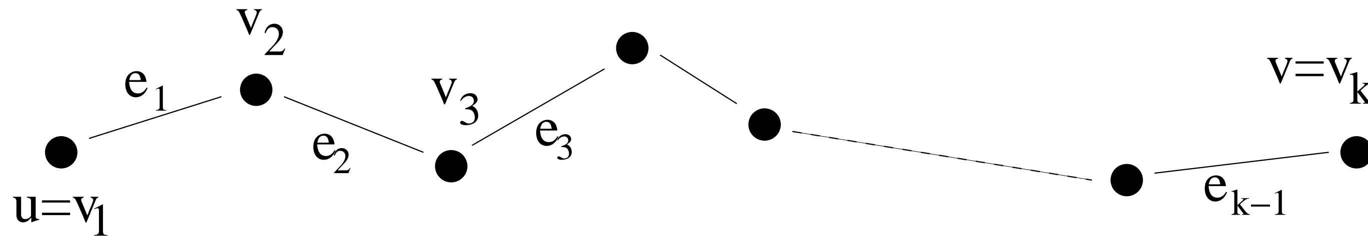


Sei p ein Pfad von u nach v mit Kosten $c(p)$. Dann

$$\begin{aligned}\bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(e_i) = \sum_{0\le i<k} (\mathsf{pot}(v_i) + c(e_i) - \mathsf{pot}(v_{i+1})) \\ &= \mathsf{pot}(v_0) + \sum_{0\le i<k} c(e_i) - \mathsf{pot}(v_k) \\ &= \mathsf{pot}(v_0) + c(p) - \mathsf{pot}(v_k).\end{aligned}$$

Sei q ein anderer Pfad, dann $c(p) \le c(q) \Leftrightarrow \bar{c}(p) \le \bar{c}(q)$.

Knotenpotentiale



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

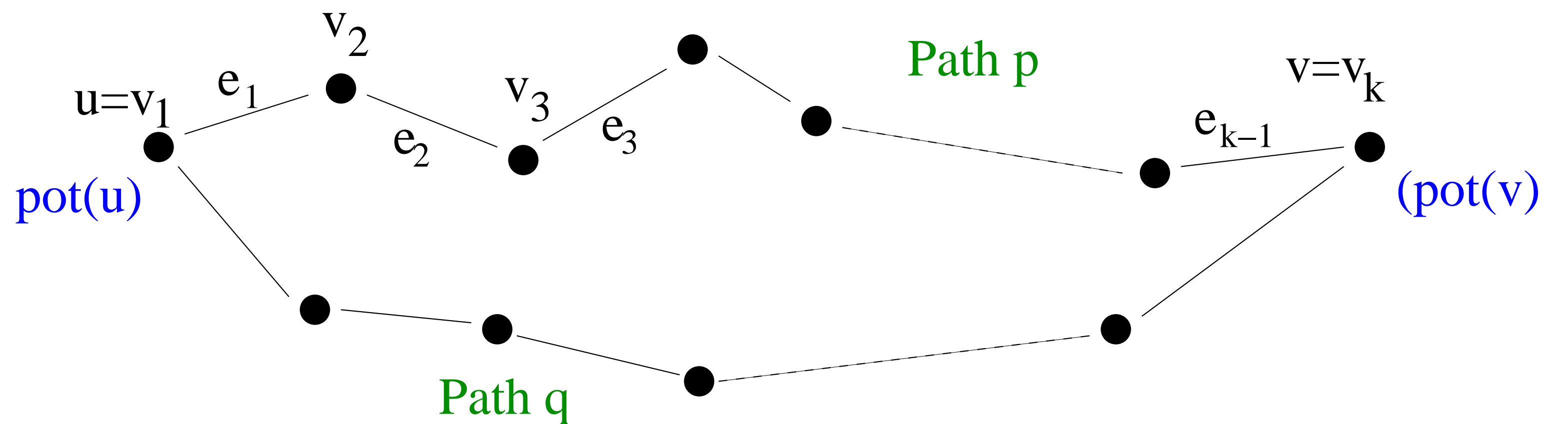
$$\begin{aligned}
\bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(e_i) = \sum_{0 \le i < k} (\mathrm{pot}(v_i) + c(e_i) - \mathrm{pot}(v_{i+1})) \\
&= \mathrm{pot}(v_0) + \sum_{0 \le i < k} c(e_i) - \mathrm{pot}(v_k) \\
&= \mathrm{pot}(v_0) + c(p) - \mathrm{pot}(v_k).
\end{aligned}$$

Knotenpotentiale

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

$$\bar{c}(p) = \mathsf{pot}(v_0) + c(p) - \mathsf{pot}(v_k).$$

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



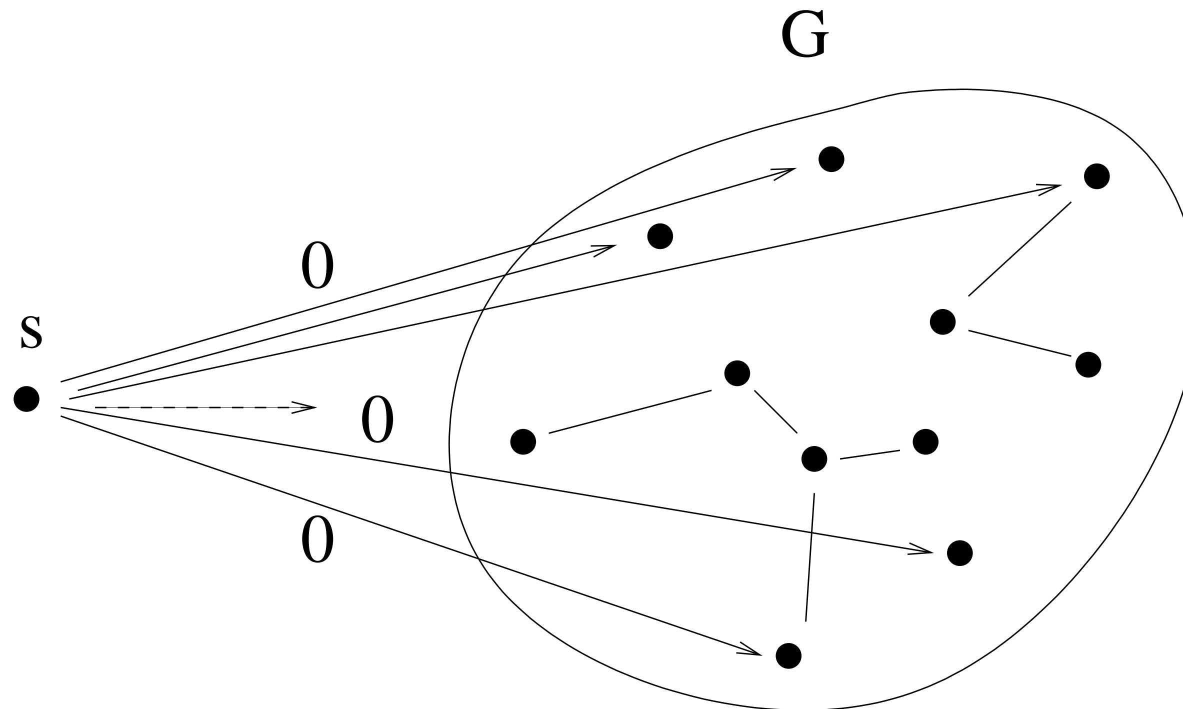
Definition: $\mu(u, v) =$ kürzeste Distanz von u nach v

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 $\mathrm{pot}(v) := \mu(v)$ 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 $\mathrm{pot}(v)$ 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

füge neuen Knoten s and Kanten (s,v) mit Kosten 0 hinzu

  für alle $v \in V$ // $\mathrm{O}(n)$

berechne $\mu(v)$ für alle v mit Bellman-Ford // $\mathrm{O}(nm)$

$\mathsf{pot}(v) := \mu(v)$ // $\mathrm{O}(m)$

forall Knoten x **do** // $\mathrm{O}(n(m + n\log n))$

  für die reduzierte Kosten $\bar{c}$, berechne Kosten $\bar{\mu}(x,v)$

  der kürzesten Pfaden von x aus mit Dijkstra.

// zurück zu den ursprünglichen Kostenfunktion // $\mathrm{O}(m)$

forall $e = (v,w) \in V \times V$ **do**

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

Laufzeit

- s hinzufügen: $\mathrm{O}(n)$
- Postprocessing: $\mathrm{O}(m)$ (zurück zu den ursprünglichen Kosten)
- n mal Dijkstra dominiert

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

Distanz zu Zielknoten t

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

1) Dijkstra, aufhören wenn t aus Q entfernt wird

Spart im Durchschnitt Hälfte der Scans

In der Praxis spart dies viel mehr (Navigationssysteme in Autos)

Distanz zu Zielknoten t

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

1) Dijkstra, aufhören wenn t aus Q entfernt wird

Spart im Durchschnitt Hälfte der Scans

In der Praxis spart dies viel mehr (Navigationssysteme in Autos)

2) Dijkstra von s **und** von t aus

Von t aus betrachten wir den umgekehrten Graphen

Parallel ausführen, Queues Q_s und Q_t gleich groß halten

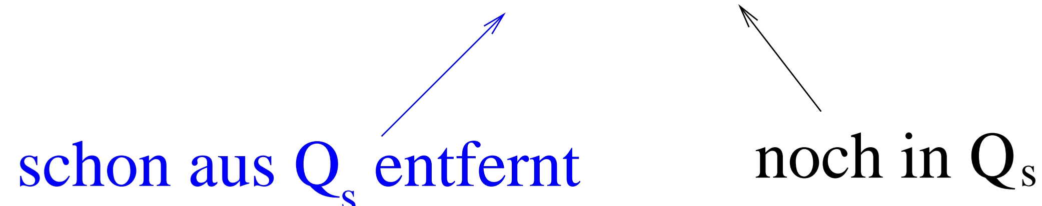
Distanz zu Zielknoten t

Wann können wir aufhören?

Idee 1 (falsch): wenn irgendein Knoten u sowohl aus Q_s als aus Q_t entfernt wurde, gilt $\mu(s,t) = \mu(s,u) + \mu(u,t)$

Stimmt leider nicht ganz! Betrachte den kürzesten Pfad p von s nach t zu diesem Zeitpunkt:

$$p = \{s = v_0, \ldots, v_i, v_{i+1} \ldots, v_k = t\}$$



Wir wissen $\mu(s,u) \leq \mu(s, v_{i+1})$ weil v_{i+1} noch in Q_s ist

Distanz zu Zielknoten t

$$p = \{s = v_0, \ldots, v_i, v_{i+1} \ldots, v_k = t\}$$

Knoten u wurde sowohl aus Q_s als aus Q_t entfernt, also $\mu(s,u) \leq \mu(s,v_{i+1})$. Dann sehen wir

$$\mu(s,v_{i+1}) + \mu(v_{i+1},t) = c(p) \leq \mu(s,u) + \mu(u,t)$$

und deshalb ($\mu(s,v_{i+1})$ abziehen)

$$\mu(v_{i+1},t) \leq \underbrace{\mu(s,u) - \mu(s,v_{i+1})}_{\leq 0} + \mu(u,t) \leq \mu(u,t)$$

Distanz zu Zielknoten t

$$p = \{s = v_0, \ldots, v_i, v_{i+1} \ldots, v_k = t\}$$

Knoten u wurde sowohl aus Q_s als aus Q_t entfernt, also $\mu(s,u) \leq \mu(s, v_{i+1})$. Dann sehen wir

$$\mu(s, v_{i+1}) + \mu(v_{i+1}, t) = c(p) \leq \mu(s,u) + \mu(u,t)$$

und deshalb ($\mu(s, v_{i+1})$ abziehen)

$$\mu(v_{i+1}, t) \leq \underbrace{\mu(s,u) - \mu(s, v_{i+1})}_{\leq 0} + \mu(u,t) \leq \mu(u,t) = d_t[u]$$

Dann folgt $d_t[v_{i+1}] = \mu(v_{i+1}, t)$. Suche in Q_s nach v_{i+1}!

Algorithmus

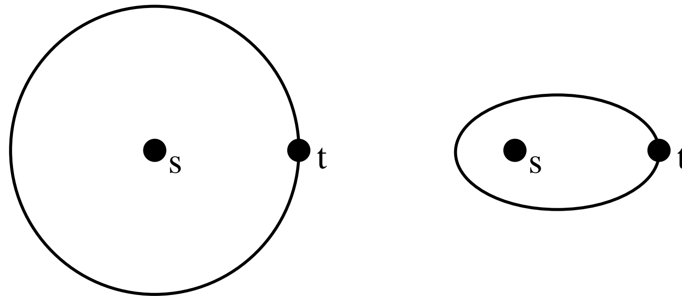
$$p = \{s = v_0, \ldots, v_i, v_{i+1} \ldots, v_k = t\}$$

- Führe Dijkstra von s und von t (auf dem umgekehrten Graphen) aus
- Höre auf wenn ein Knoten u aus Q_s und Q_t entfernt wurde
- Finde in Q_s den Knoten v für den $d_s[v] + d_t[v]$ minimal ist
- Für $v = v_{i+1}$ gilt

$$d_s[v_{i+1}] + d_t[v_{i+1}] = \mu(s, v_{i+1}) + \mu(v_{i+1}, t) = c(p)$$

A^*-Suche

Idee: suche “in die Richtung von t”



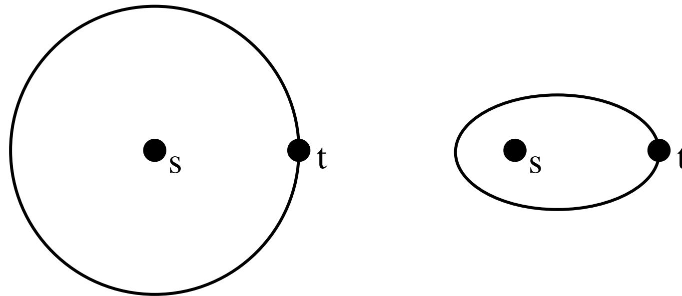
Annahme: Wir kennen eine Funktion $f(v)$ die $\mu(v,t)$ schätzt $\forall v$

Definiere $\mathsf{pot}(v) = f(v)$ 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 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 betrachtet Dijkstra nur die Kanten auf diesem Pfad!

Benötigte Eigenschaften von $f(v)$

- Konsistenz (reduzierte Kosten nicht negativ):
  $c(e) + f(v) \geq f(u)\ \forall e = (u,v)$
- $f(v) \leq \mu(v,t)\ \forall v \in V$. Dann gilt $f(t) = 0$ und wir können aufhören wenn t aus Q entfernt wird.

Sei p irgendein Pfad von s nach t.
Alle Kanten auf p sind relaxiert: $d[t] \leq c(p)$.
Sonst: $\exists v \in p \cup Q$, 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 $f(v)$.

Straßennetzwerke: $f(v) =$ Distanz über eine gerade Linie

Oder, wenn wir die schnellste Verbindung suchen: $f(v) =$
Distanz über gerade Linie durch bestmögliche Geschwindigkeit

Andere Ideen

- Hierarchien
- Transitknoten