Script generated by TTT

Title: Täubig: GAD (17.07.2012)

Date: Tue Jul 17 14:33:49 CEST 2012

Duration: 85:47 min

Pages: 47

Kürzeste Pfade: SSSP / Dijkstra

Dijkstra-Algorithmus

Input : $G = (V, E)$, $c : E \to \mathbb{R}$, $s \in V$
Output : Distanzen $d(s, v)$ zu allen $v \in V$

$P = \emptyset$; $T = V$;
$d(s, v) = \infty$ for all $v \in V \setminus s$;
$d(s, s) = 0$; $pred(s) = 0$;
while $P \neq V$ **do**
  $v = \text{argmin}_{v \in T}\{d(s, v)\}$;
  $P = P \cup v$; $T = T \setminus v$;
  forall the $(v, w) \in E$ **do**
    if $d(s, w) > d(s, v) + c(v, w)$ **then**
      $d(s, w) = d(s, v) + c(v, w)$;
      $pred(w) = v$;

Dijkstra-Algorithmus für SSSP

Input : $G = (V, E)$, $c : E \to \mathbb{R}_{\geq 0}$, $s \in V$
Output : Distanzen $d[v]$ von s zu allen $v \in V$

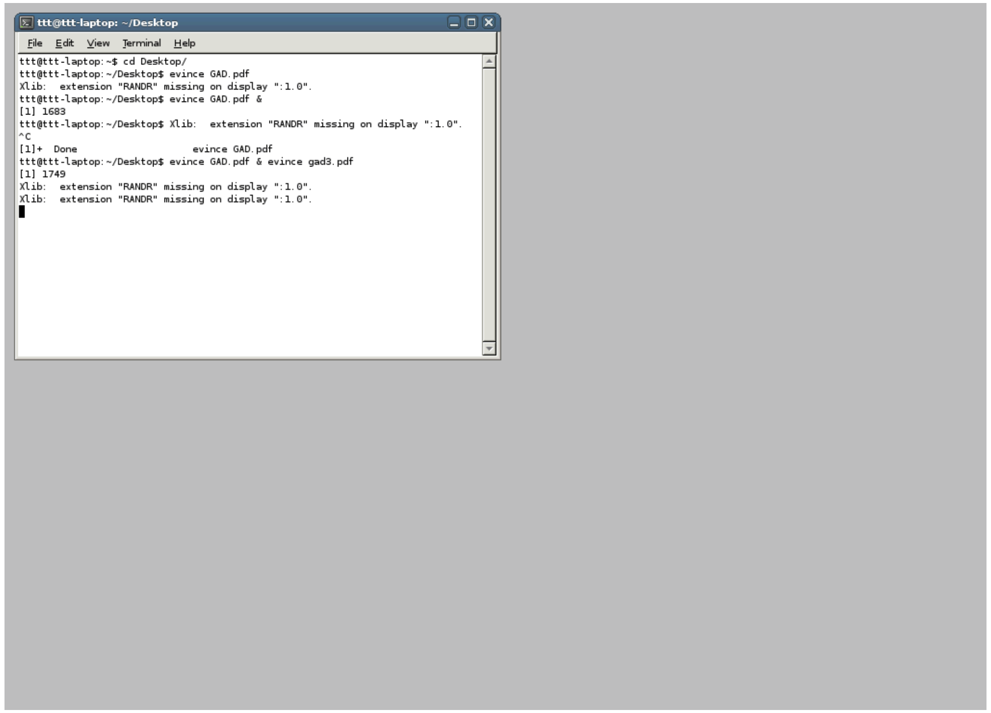
$d[v] = \infty$ for all $v \in V \setminus s$;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $\neg pq.empty()$ **do**
  $v = pq$.deleteMin();
  forall the $(v, w) \in E$ **do**
    newDist $= d[v] + c(v, w)$;
    if $newDist < d[w]$ **then**
      $pred[w] = v$;
      if $d[w] == \infty$ **then** pq.insert$(w, \text{newDist})$;
      else pq.decreaseKey$(w, \text{newDist})$;
      $d[w] = $ newDist;

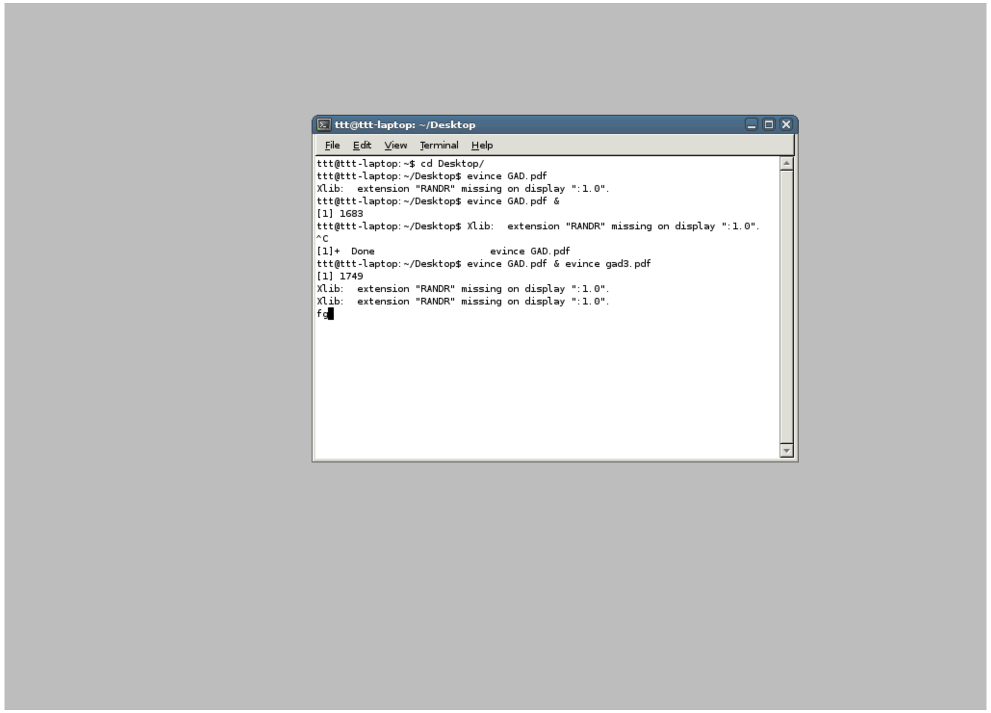
Grundlagen: Algorithmen und Datenstrukturen
File Edit View Go Help

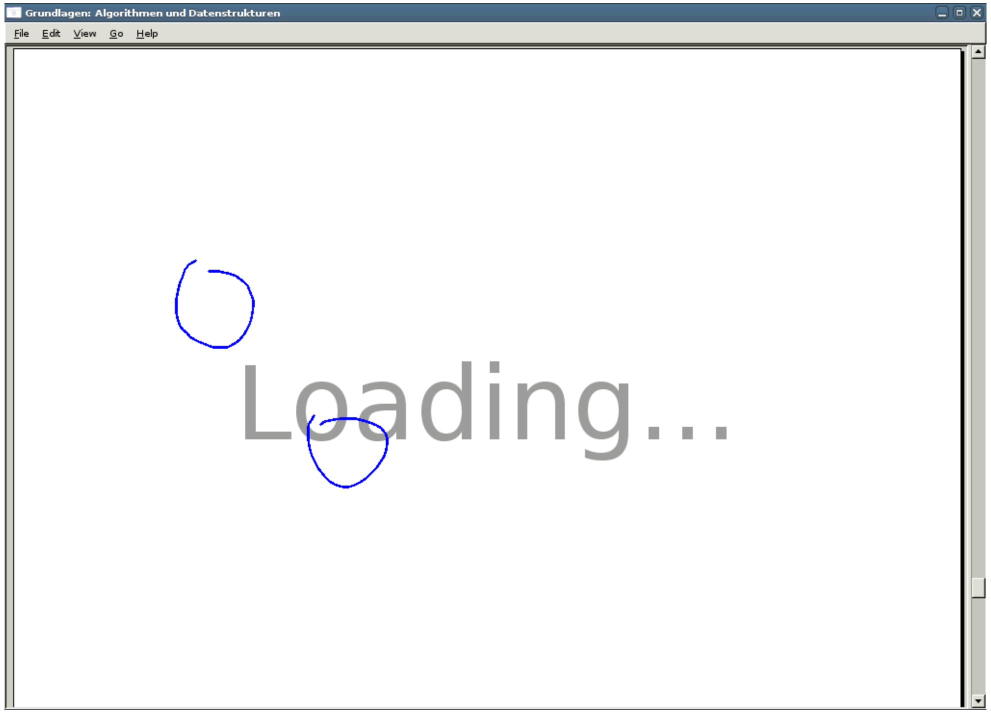
Dijkstra-Algorithmus für SSSP

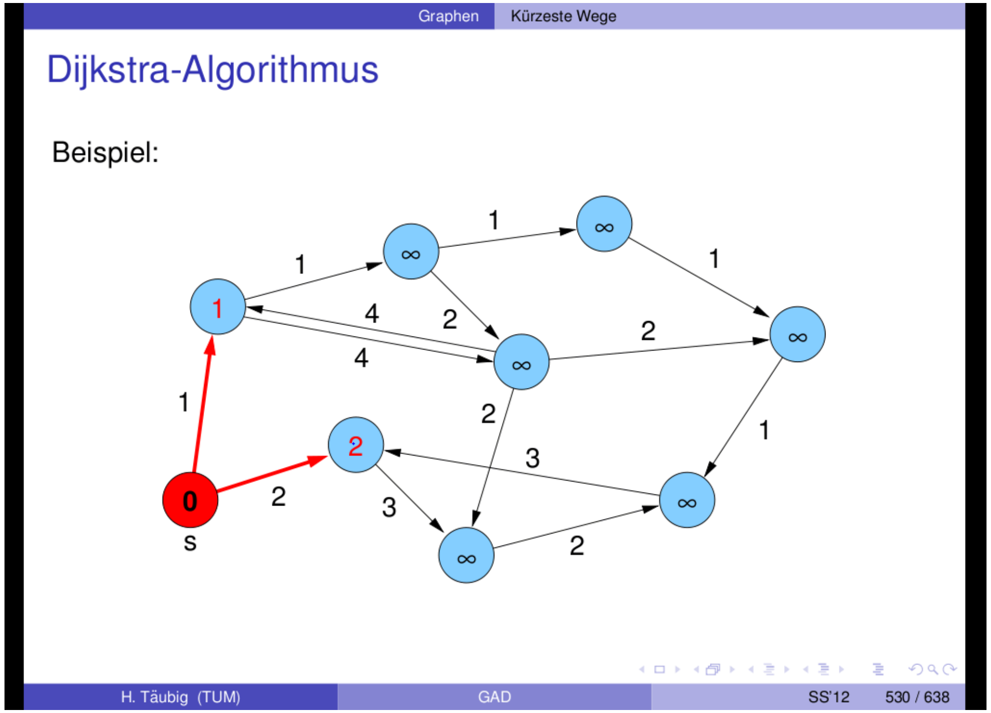
Input : $G = (V, E)$, $c : E \to \mathbb{R}_{\geq 0}$, $s \in V$
Output : Distanzen $d[v]$ von s zu allen $v \in V$

$d[v] = \infty$ for all $v \in V \setminus s$;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $\neg pq.empty()$ **do**
  $v = pq$.deleteMin();
  forall the $(v, w) \in E$ **do**
    newDist $= d[v] + c(v, w)$;
    if $newDist < d[w]$ **then**
      $pred[w] = v$;
      if $d[w] == \infty$ **then** pq.insert$(w, \text{newDist})$;
      else pq.decreaseKey$(w, \text{newDist})$;
      $d[w] = $ newDist;


ttt@ttt-laptop: ~/Desktop
File Edit View Terminal Help
ttt@ttt-laptop:~$ cd Desktop/
ttt@ttt-laptop:~/Desktop$ evince GAD.pdf
Xlib: extension "RANDR" missing on display ":1.0".
ttt@ttt-laptop:~/Desktop$ evince GAD.pdf &
[1] 1683
ttt@ttt-laptop:~/Desktop$ Xlib: extension "RANDR" missing on display ":1.0".
^C
[1]+ Done   evince GAD.pdf
ttt@ttt-laptop:~/Desktop$ evince GAD.pdf & evince gad3.pdf
[1] 1749
Xlib: extension "RANDR" missing on display ":1.0".
Xlib: extension "RANDR" missing on display ":1.0".

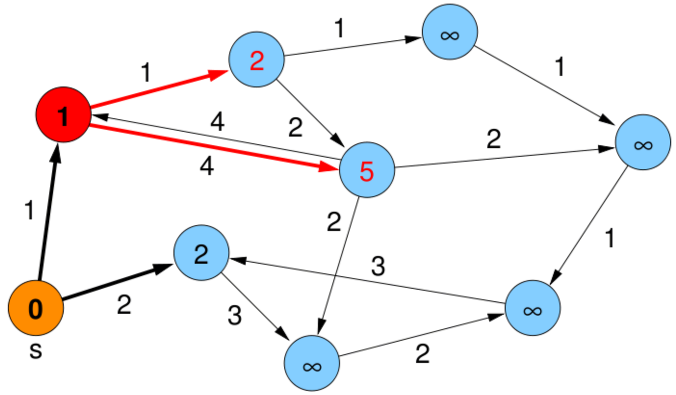

ttt@ttt-laptop: ~/Desktop
File Edit View Terminal Help
ttt@ttt-laptop:~$ cd Desktop/
ttt@ttt-laptop:~/Desktop$ evince GAD.pdf
Xlib: extension "RANDR" missing on display ":1.0".
ttt@ttt-laptop:~/Desktop$ evince GAD.pdf &
[1] 1683
ttt@ttt-laptop:~/Desktop$ Xlib: extension "RANDR" missing on display ":1.0".
^C
[1]+ Done   evince GAD.pdf
ttt@ttt-laptop:~/Desktop$ evince GAD.pdf & evince gad3.pdf
[1] 1749
Xlib: extension "RANDR" missing on display ":1.0".
Xlib: extension "RANDR" missing on display ":1.0".
fg


Grundlagen: Algorithmen und Datenstrukturen
File Edit View Go Help
Loading...


Graphen
Kürzeste Wege
Dijkstra-Algorithmus
Beispiel:
s
H. Täubig (TUM)
GAD
SS'12
530 / 638

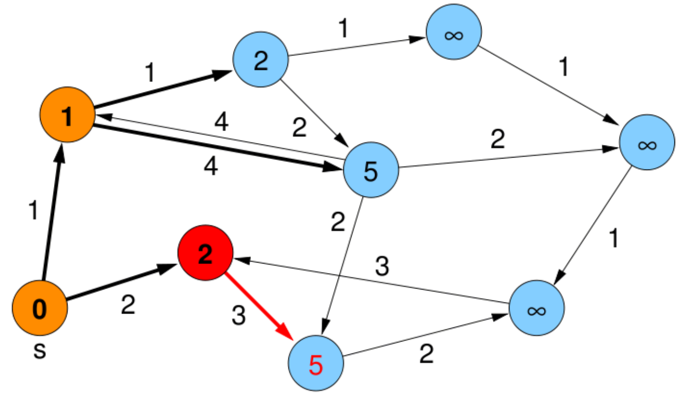
Dijkstra-Algorithmus

Beispiel:



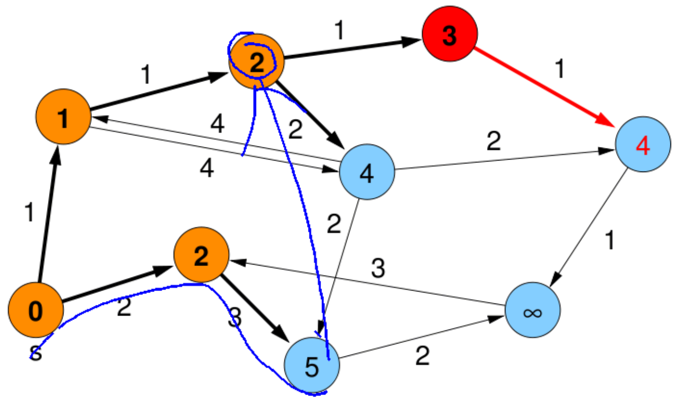
Dijkstra-Algorithmus

Beispiel:



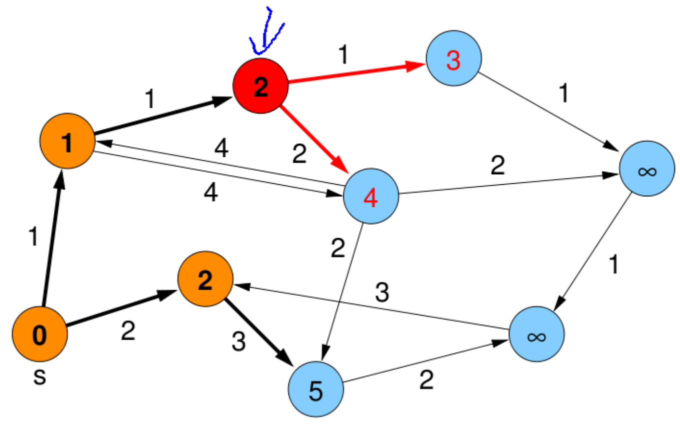
Dijkstra-Algorithmus

Beispiel:



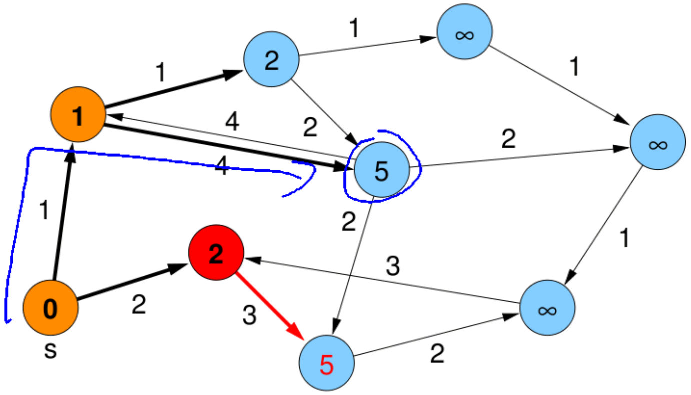
Dijkstra-Algorithmus

Beispiel:



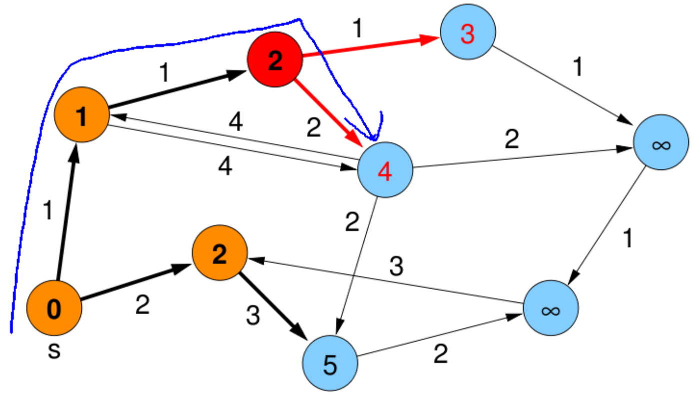
Dijkstra-Algorithmus

Beispiel:



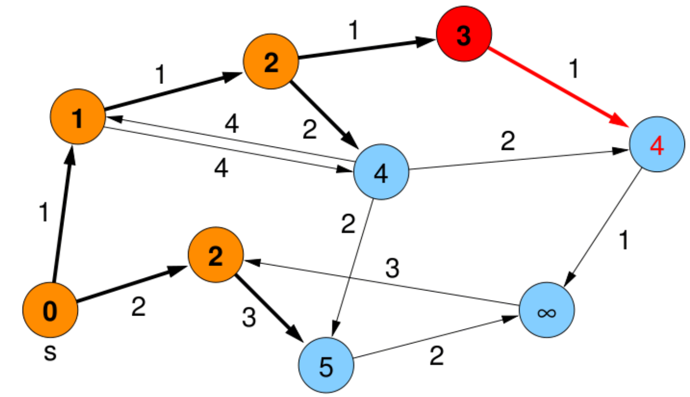
Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Korrektheit:

- Annahme: Algorithmus liefert für w einen zu kleinen Wert $d(s, w)$
- sei w der erste Knoten, für den die Distanz falsch festgelegt wird (kann nicht s sein, denn die Distanz $d(s, s)$ bleibt immer 0)
- kann nicht ~~sein, weil~~ $d(s, w)$ nur dann aktualisiert wird, wenn man über ~~einen von s~~ schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann
- d.h. $d(s, v)$ müsste schon falsch gewesen sein (Widerspruch zur Annahme, ~~dass w der erste Knoten mit~~ falscher Distanz war)

Dijkstra-Algorithmus

- setze Startwert $d(s,s) = 0$ und zunächst $d(s,v) = \infty$
- verwende Prioritätswarteschlange, um die Knoten zusammen mit ihren aktuellen Distanzen zuspeichern
- am Anfang nur Startknoten (mit Distanz 0) in Priority Queue
- dann immer nächsten Knoten v (mit kleinster Distanz) entnehmen, endgültige Distanz dieses Knotens v steht nun fest
- betrachte alle Nachbarn von v,
  füge sie ggf. in die PQ ein bzw.
  aktualisiere deren Priorität in der PQ

Monotone Priority Queues

$d_{min} + c(v,w)$

Beobachtung:

- aktuelles Distanz-Minimum der verbleibenden Knoten ist beim Dijkstra-Algorithmus monoton wachsend

Monotone Priority Queue

- Folge der entnommenen Elemente hat monoton steigende Werte
- effizientere Implementierung möglich, falls Kantengewichte ganzzahlig

d_{min}

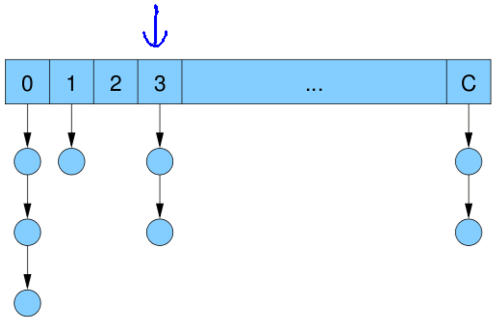
Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

$\Rightarrow$ enthaltene Distanzwerte immer im Bereich $[d, d + C]$

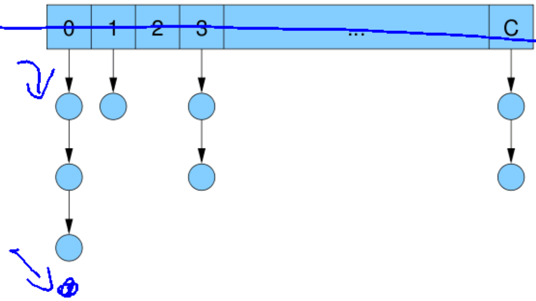
Bucket Queue

- Array B aus $C + 1$ Listen
- Variable d_{min} für aktuelles Distanzminimum $\bmod (C + 1)$



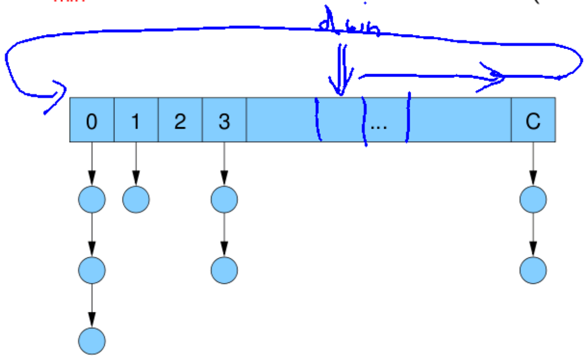
Bucket Queue

- jeder Knoten v mit aktueller Distanz $d[v]$ in Liste $B[d[v] \bmod (C + 1)]$
- alle Knoten in Liste $B[d]$ haben dieselbe Distanz, weil alle aktuellen Distanzen im Bereich $[d, d + C]$ liegen



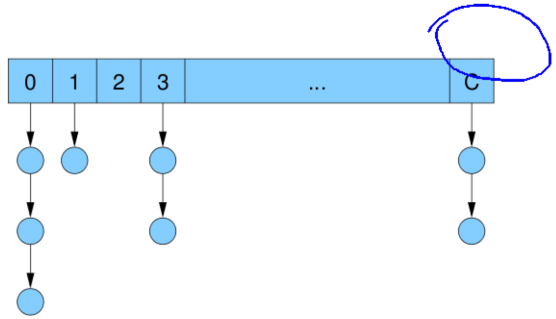
Bucket Queue

- Array B aus $C+1$ Listen
- Variable $d_{\min}$ für aktuelles Distanzminimum $\mathrm{mod}(C+1)$



Bucket Queue

- jeder Knoten v mit aktueller Distanz $d[v]$ in Liste $B[d[v] \bmod (C+1)]$
- alle Knoten in Liste $B[d]$ haben dieselbe Distanz, weil alle aktuellen Distanzen im Bereich $[d, d+C]$ liegen



Dijkstra mit Bucket Queue

- insert, decreaseKey: $O(1)$
- deleteMin: $O(C)$
- Dijkstra: $O(m + C \cdot n)$
- lässt sich mit Radix Heaps noch verbessern
- verwendet exponentiell wachsende Bucket-Größen
- Details in der Vorlesung Effiziente Algorithmen und Datenstrukturen
- Laufzeit ist dann $O(m + n \log C)$

Beliebige Graphen mit beliebigen Gewichten

Gegeben:

- beliebiger Graph mit beliebigen Kantengewichten

$\Rightarrow$ Anhängen einer Kante an einen Weg kann zur Verkürzung des Weges (Kantengewichtssumme) führen (wenn Kante negatives Gewicht hat)

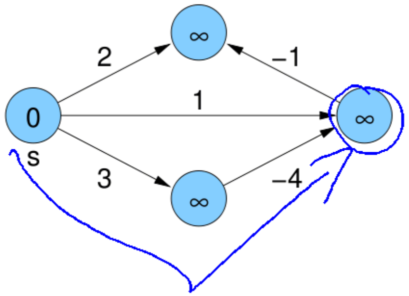
$\Rightarrow$ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- Dijkstra kann nicht mehr verwendet werden, weil Knoten nicht unbedingt in der Reihenfolge der kürzesten Distanz zum Startknoten s besucht werden

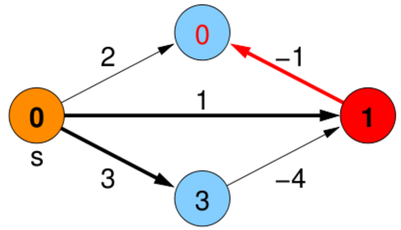
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



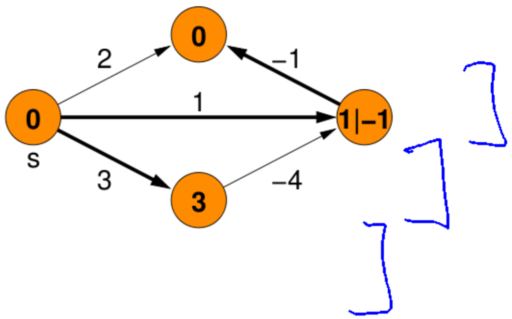
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Bellman-Ford-Algorithmus

Folgerung

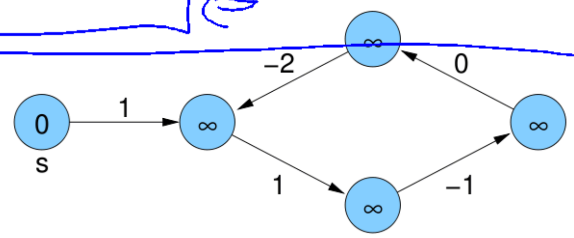
In einem Graph mit n Knoten gibt es für jeden erreichbaren Knoten v mit $d(s, v) > -\infty$ einen kürzesten Weg bestehend aus $< n$ Kanten zwischen s und v.

Strategie:

- anstatt kürzeste Pfade in Reihenfolge wachsender Gewichtssumme zu berechnen, betrachte sie in Reihenfolge steigender Kantenanzahl
- durchlaufe (n-1)-mal alle Kanten im Graph und aktualisiere die Distanz
- dann alle kürzesten Wege berücksichtigt

Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



$d[w] \le d[v] + c(v,w) \le d(v) + \boxed{c[v,v] + c[v,w]}$

Bellman-Ford-Algorithmus

Zusammenfassung:

- keine Distanzerniedrigung mehr möglich
  ($d[v] + c(v,w) \geq d[w]$ für alle w):
  fertig, alle $d[w]$ korrekt für alle w
- Distanzerniedrigung möglich selbst noch in n-ter Runde
  ($d[v] + c(v,w) < d[w]$ für ein w):
  Es gibt einen negativen Kreis, also Knoten w mit Distanz $-\infty$.

$s \to v_1 \to v_2 \to v_3 \to \dots \to v_i \to \to v$

Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  d[s] = 0;   parent[s] = s;
  for (int i = 0; i < n − 1; i++) {   // n − 1 Runden
    foreach (e = (v, w) ∈ E)
      if (d[v] + c(e) < d[w]) {   // kürzerer Weg?
        d[w] = d[v] + c(e);
        parent[w] = v;
      }
  }
  foreach (e = (v, w) ∈ E)
    if (d[v] + c(e) < d[w]) {   // kürzerer Weg in n-ter Runde?
      infect(w);
}
```


Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  d[s] = 0;   parent[s] = s;
  for (int i = 0; i < n − 1; i++) {   // n − 1 Runden
    foreach (e = (v, w) ∈ E)
      if (d[v] + c(e) < d[w]) {   // kürzerer Weg?
        d[w] = d[v] + c(e);
        parent[w] = v;
      }
  }
  foreach (e = (v, w) ∈ E)
    if (d[v] + c(e) < d[w]) {   // kürzerer Weg in n-ter Runde?
      infect(w);
}
```

Bellman-Ford-Algorithmus

```
infect(Node v) {   // −∞-Knoten
  if (d[v] > −∞) {
    d[v] = −∞;
    foreach (e = (v, w) ∈ E)
      infect(w);
  }
}
```

Gesamtlaufzeit: $\boxed{O(m \cdot n)}$

Bellman-Ford-Algorithmus

Bestimmung der Knoten mit Distanz $-\infty$:

- betrachte alle Knoten, die in der n-ten Phase noch Distanzverbesserung erfahren
- aus jedem Kreis mit negativem Gesamtgewicht muss mindestens ein Knoten dabei sein
- jeder von diesen Knoten aus erreichbare Knoten muss Distanz $-\infty$ bekommen
- das erledigt hier die infect-Funktion
- wenn ein Knoten zweimal auftritt (d.h. der Wert ist schon $-\infty$), wird die Rekursion abgebrochen

Bellman-Ford-Algorithmus

Ursprüngliche Idee der Updates vorläufiger Distanzwerte stammt von Lester R. Ford Jr.

Verbesserung (Richard E. Bellman / Edward F. Moore):

- verwalte FIFO-Queue von Knoten, zu denen ein kürzerer Pfad gefunden wurde und deren Nachbarn am anderen Ende ausgehender Kanten noch auf kürzere Wege geprüft werden müssen
- wiederhole: nimm ersten Knoten aus der Queue und prüfe für jede ausgehende Kante die Distanz des Nachbarn

  falls kürzerer Weg gefunden, aktualisiere Distanzwert des Nachbarn und hänge ihn an Queue an (falls nicht schon enthalten)
- Phase besteht immer aus Bearbeitung der Knoten, die am Anfang des Algorithmus (bzw. der Phase) in der Queue sind (dabei kommen während der Phase schon neue Knoten ans Ende der Queue)

Kürzeste einfache Pfade bei beliebigen Kantengewichten

Achtung!

Fakt

Die Suche nach kürzesten einfachen Pfaden (also ohne Knotenwiederholungen / Kreise) in Graphen mit beliebigen Kantengewichten (also möglichen negativen Kreisen) ist ein $\mathcal{NP}$-vollständiges Problem.

(Man könnte Hamilton-Pfad-Suche damit lösen.)

All Pairs Shortest Paths

gegeben:

- Graph mit beliebigen Kantengewichten, der aber keine negativen Kreise enthält

gesucht:

- Distanzen / kürzeste Pfade zwischen allen Knotenpaaren

Naive Strategie:

- n-mal Bellman-Ford-Algorithmus (jeder Knoten einmal als Startknoten)

$\boxed{\Rightarrow O(n^2 \cdot m)}$

$(\log n \, n + m)\, n$

All Pairs Shortest Paths

Bessere Strategie:

- reduziere n Aufrufe des Bellman-Ford-Algorithmus auf n Aufrufe des Dijkstra-Algorithmus

Problem:

- Dijkstra-Algorithmus funktioniert nur für nichtnegative Kantengewichte

Lösung:

- Umwandlung in nichtnegative Kantenkosten ohne Verfälschung der kürzesten Wege

All Pairs Shortest Paths

Sei $\Phi : V \mapsto \mathbb{R}$ eine Funktion, die jedem Knoten ein Potential zuordnet.

Modifizierte Kantenkosten von $e = (v, w)$:

$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$

Lemma

Seien p und q Wege von v nach w in G.

$c(p)$ und $c(q)$ bzw. $\bar{c}(p)$ und $\bar{c}(q)$ seien die aufsummierten Kosten bzw. modifizierten Kosten der Kanten des jeweiligen Pfads.

Dann gilt für jedes Potential Φ:

$$\boxed{\bar{c}(p) < \bar{c}(q)} \quad \Leftrightarrow \quad c(p) < c(q)$$

All Pairs Shortest Paths

Beweis.

Sei $p = (v_1, \ldots, v_k)$ beliebiger Weg und $\forall i : \; e_i = (v_i, v_{i+1}) \in E$

Es gilt:

$$\begin{aligned}
\bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) \\
&= \sum_{i=1}^{k-1} \left(\Phi(v_i) + c(e_i) - \Phi(v_{i+1})\right) \\
&= \Phi(v_1) + \underline{c(p)} - \Phi(v_k)
\end{aligned}$$

d.h. modifizierte Kosten eines Pfads hängen nur von ursprünglichen Pfadkosten und vom Potential des Anfangs- und Endknotens ab. (Im Lemma ist $v_1 = v$ und $v_k = w$) □

All Pairs Shortest Paths

Lemma

Annahme:

- *Graph hat keine negativen Kreise*
- *alle Knoten von s aus erreichbar*

Sei für alle Knoten v das Potential $\Phi(v) = d(s, v)$.

Dann gilt für alle Kanten e: $\boxed{\bar{c}(e) \geq 0}$

Beweis.

- für alle Knoten v gilt nach Annahme: $d(s, v) \in \mathbb{R}$ (also $\neq \pm\infty$)
- für jede Kante $e = (v, w)$ ist

$$\begin{aligned} d(s,v) + c(e) &\geq d(s,w) \quad \Delta \\ \boxed{d(s,v) + c(e) - d(s,w) \geq 0} \end{aligned}$$

□

All Pairs Shortest Paths / Johnson-Algorithmus

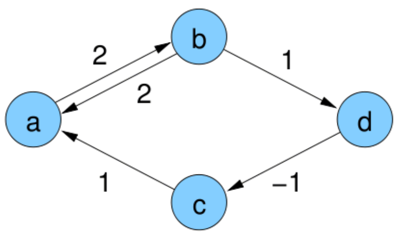
- füge neuen Knoten s und Kanten (s, v) für alle v hinzu mit $c(s, v) = 0$

⇒ alle Knoten erreichbar ✓

- berechne $\boxed{d(s, v)}$ mit Bellman-Ford-Algorithmus
- setze $\Phi(v) = d(s, v)$ für alle v
- berechne modifizierte Kosten $\bar{c}(e)$ ✓
- berechne für alle Knoten v die Distanzen $\boxed{\bar{d}(v, w)}$ mittels Dijkstra-Algorithmus mit modifizierten Kantenkosten auf dem Graph ohne Knoten s
- berechne korrekte Distanzen $d(v, w) = \bar{d}(v, w) + \Phi(w) - \Phi(v)$ ✓

All Pairs Shortest Paths / Johnson-Algorithmus

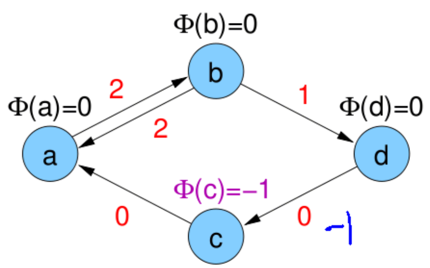
Beispiel:



All Pairs Shortest Paths / Johnson-Algorithmus

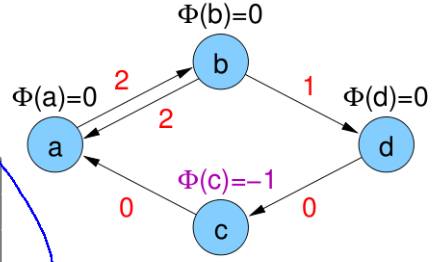
3. $\bar{c}(e)$-Werte für alle $e = (v, w)$ berechnen:

$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$



All Pairs Shortest Paths / Johnson-Algorithmus

4. Distanzen $\bar{d}$ mit modifizierten Kantengewichten via Dijkstra:

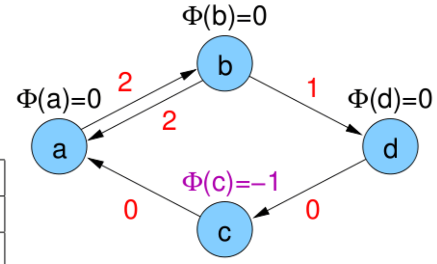


$\bar{d}$	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0

All Pairs Shortest Paths / Johnson-Algorithmus

5. korrekte Distanzen berechnen mit Formel

$$d(v,w) = \underline{\bar{d}(v,w) + \Phi(w) - \Phi(v)}$$



d	a	b	c	d
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	−1	0

All Pairs Shortest Paths / Johnson-Algorithmus

Laufzeit:

$$\begin{aligned} T(APSP) &= O(\boxed{T_{\text{Bellman-Ford}}(n+1, m+n)} + n \cdot \boxed{T_{\text{Dijkstra}}(n,m)}) \\ &= O((m+n)\cdot(n+1) + n(n\log n + m)) \\ &= O(m \cdot n + n^2 \log n) \end{aligned}$$

(bei Verwendung von Fibonacci Heaps)

APSP / Floyd-Warshall-Algorithmus

Grundlage:

- geht der kürzeste Weg von u nach w über v, dann sind auch die beiden Teile von u nach v und von v nach w kürzeste Pfade zwischen diesen Knoten
- Annahme: alle kürzesten Wege bekannt, die nur über Zwischenknoten mit Index kleiner als k gehen

⇒ kürzeste Wege über Zwischenknoten mit Indizes bis einschließlich k können leicht berechnet werden:

  - entweder der schon bekannte Weg über Knoten mit Indizes kleiner als k
  - oder über den Knoten mit Index k (hier im Algorithmus der Knoten v)