Script **generated by TTT**

Title: Seidl: GAD (21.06.2016)

Date: Tue Jun 21 14:32:39 CEST 2016

Duration: 79:24 min

Pages: 50

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

- beliebiger Graph
  (gerichtet oder ungerichtet, muss diesmal kein DAG sein)
- mit nicht-negativen Kantengewichten

$\Rightarrow$ keine Knoten mit Distanz $-\infty$

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- wie bei Breitensuche, jedoch diesmal auch mit Distanzen $\neq 1$

Lösung:

- besuche Knoten in der Reihenfolge der kürzesten Distanz zum Startknoten s

Kürzeste Pfade: SSSP / Dijkstra

Algorithmus Dijkstra1: löst SSSP-Problem

Eingabe : $G = (V, E), \quad c : E \mapsto \mathbb{R}, \quad s \in V$
Ausgabe : Distanzen $d(s, v)$ zu allen $v \in V$

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

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

- beliebiger Graph
  (gerichtet oder ungerichtet, muss diesmal kein DAG sein)
- mit nicht-negativen Kantengewichten

$\Rightarrow$ keine Knoten mit Distanz $-\infty$

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- wie bei Breitensuche, jedoch diesmal auch mit Distanzen $\neq 1$

Lösung:

- besuche Knoten in der Reihenfolge der kürzesten Distanz zum Startknoten s

Kürzeste Pfade: SSSP / Dijkstra

Algorithmus Dijkstra1: löst SSSP-Problem

Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}$, $s \in V$
Ausgabe : Distanzen $d(s, v)$ zu allen $v \in V$

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

Graphen | Kürzeste Wege

Kürzeste Pfade: SSSP / Dijkstra

Algorithmus Dijkstra1: löst SSSP-Problem

Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}$, $s \in V$
Ausgabe : Distanzen $d(s, v)$ zu allen $v \in V$

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

H. Seidl (TUM) GAD SS'16 457

Algorithmus Dijkstra2: löst SSSP-Problem

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

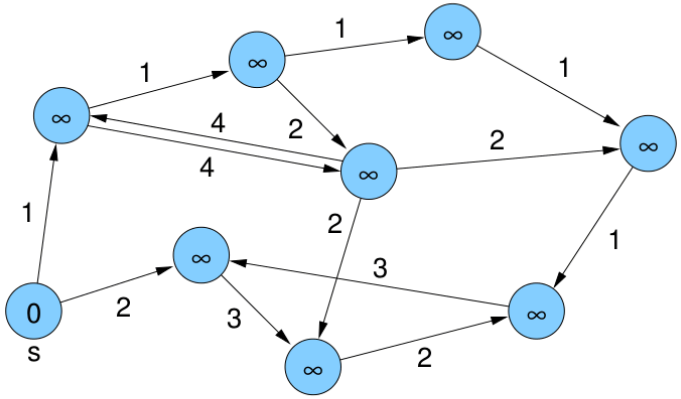
forall $v \in V \setminus s$ **do**
  $d[v] = \infty$
;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $(\neg pq.empty())$ **do**
  $v = pq$.deleteMin();
  forall $(v, w) \in E$ **do**
    newDist $= d[v] + c(v, w)$;
    if $(newDist < d[w])$ **then**
      $pred[w] =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

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

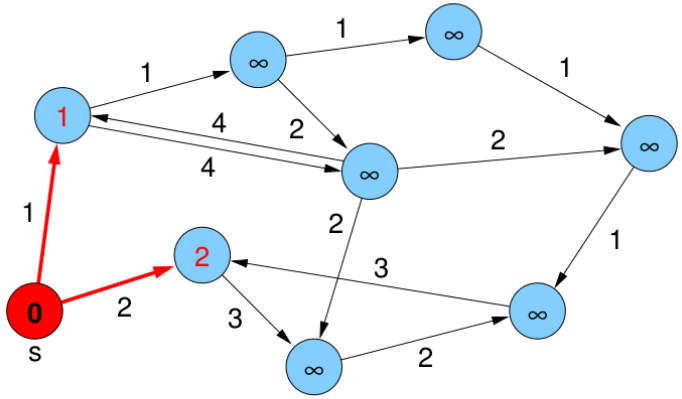
Dijkstra-Algorithmus

Beispiel:



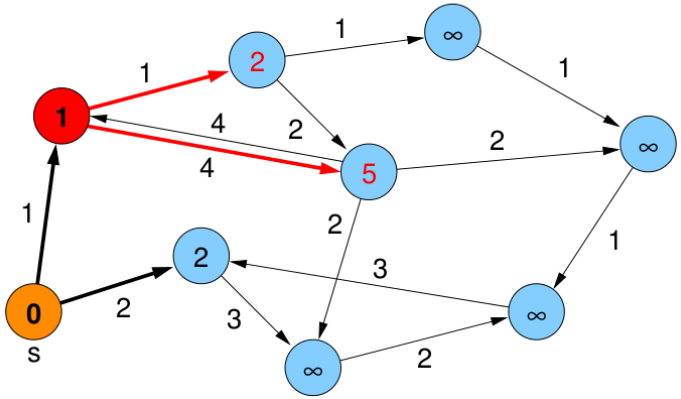
Dijkstra-Algorithmus

Beispiel:



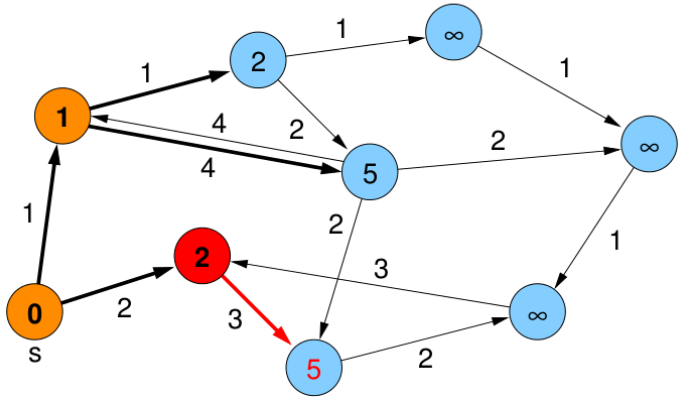
Dijkstra-Algorithmus

Beispiel:



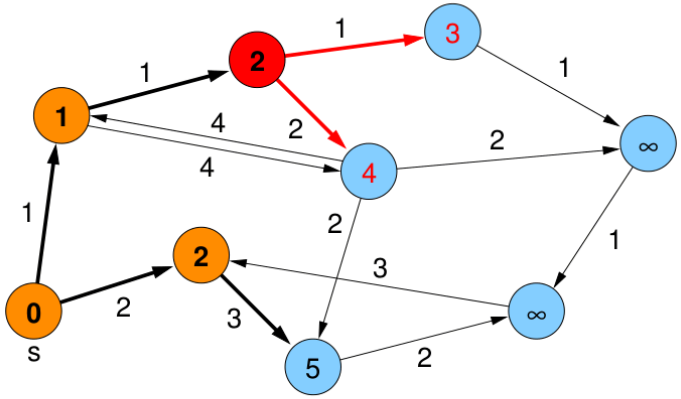
Dijkstra-Algorithmus

Beispiel:



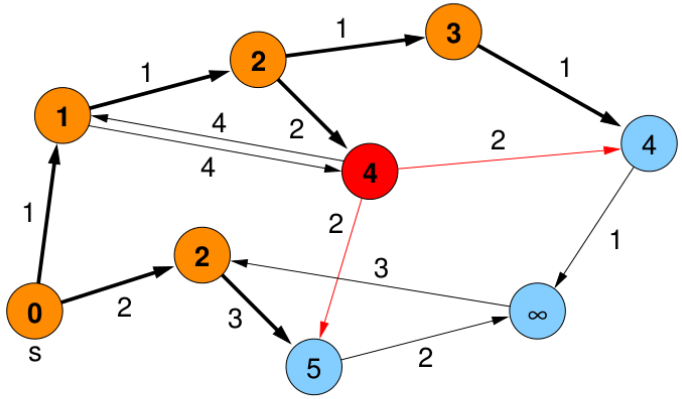
Dijkstra-Algorithmus

Beispiel:



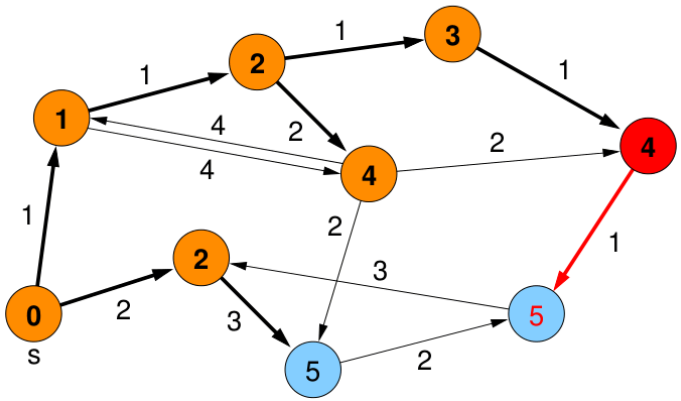
Dijkstra-Algorithmus

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

- Annahme: Algorithmus liefert für w einen zu großen Wert $d(s,w)$
- sei w der Knoten mit der kleinsten (wirklichen) Distanz, für den der Wert $d(s,w)$ falsch festgelegt wird (wenn es davon mehrere gibt, der Knoten, für den die Distanz zuletzt festgelegt wird)
- kann nicht sein, weil $d(s,w)$ immer 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 (dabei steht $d(s,v)$ immer schon fest, so dass auch die Länge eines kürzesten Wegs über v zu w richtig berechnet wird)
- d.h., entweder wurde auch der Wert von v falsch berechnet (Widerspruch zur Def. von w) oder die Distanz von v wurde noch nicht festgesetzt
- weil die berechneten Distanzwerte monoton wachsen, kann letzteres nur passieren, wenn v die gleiche Distanz hat wie w (auch Widerspruch zur Def. von w)

Dijkstra-Algorithmus

- Datenstruktur: Prioritätswarteschlange
  (z.B. Fibonacci Heap: amortisierte Komplexität $O(1)$ für `insert` und `decreaseKey`, $O(\log n)$ `deleteMin`)
- Komplexität:
  - $n \times O(1)$ `insert`
  - $n \times O(\log n)$ `deleteMin`
  - $m \times O(1)$ `decreaseKey`

  $\Rightarrow O(m + n \log n)$
- aber: nur für nichtnegative Kantengewichte(!)

Monotone Priority Queues

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

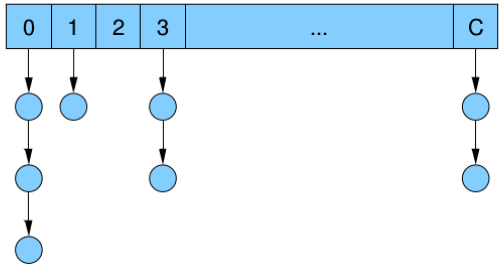
Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

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

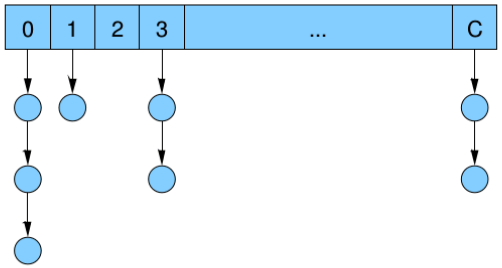
Bucket Queue

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



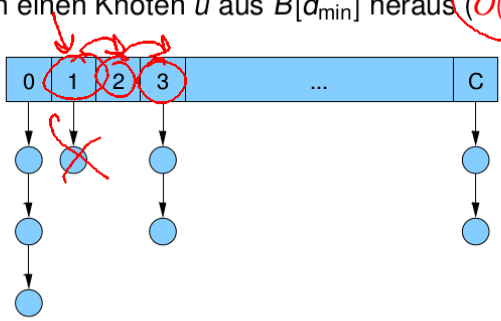
Bucket Queue

- 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



Bucket Queue / Operationen

- insert(v): fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($O(1)$)
- decreaseKey(v): entfernt v aus momentaner Liste ($O(1)$ falls Handle auf Listenelement in v gespeichert) und fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($O(1)$)
- deleteMin(): solange $B[d_{\min}] = \emptyset$, setze $d_{\min} = (d_{\min}+1) \bmod (C+1)$. Nimm dann einen Knoten u aus $B[d_{\min}]$ heraus ($O(C)$)



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)

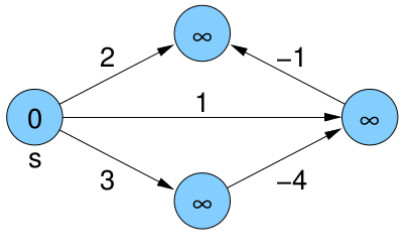
$\Rightarrow$ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

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

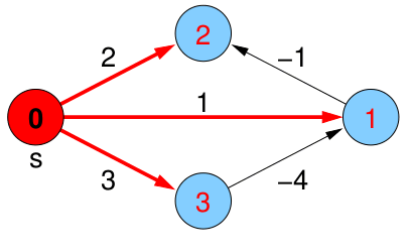
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



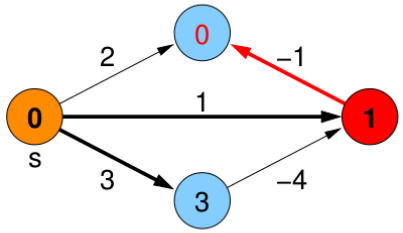
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



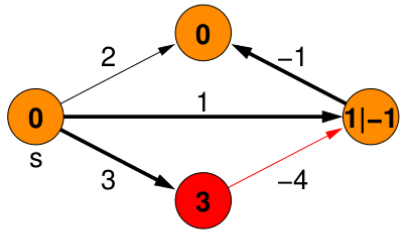
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Lemma

Für jeden von s erreichbaren Knoten v mit $d(s,v) > -\infty$ gibt es einen einfachen Pfad (ohne Kreis) von s nach v der Länge $d(s,v)$.

Beweis.

Betrachte kürzesten Weg mit Kreis(en):

- Kreis mit Kantengewichtssumme > 0 nicht enthalten:
  Entfernen des Kreises würde Kosten verringern
- Kreis mit Kantengewichtssumme $= 0$:
  Entfernen des Kreises lässt Kosten unverändert
- Kreis mit Kantengewichtssumme < 0:
  Distanz von s ist $-\infty$

□

Bellman-Ford-Algorithmus

Folgerung

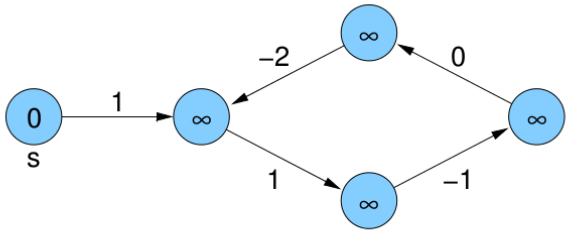
In einem Graph mit n Knoten gibt es für jeden erreichbaren Knoten v mit $d(s,v) > -\infty$ einen kürzesten Weg bestehend aus $< n$ Kanten zwischen s und v.

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

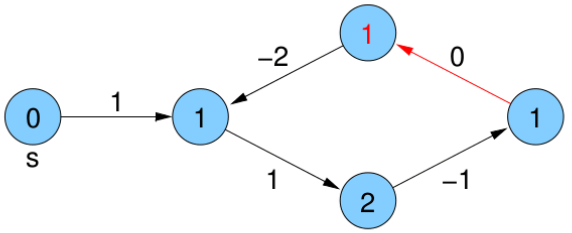
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



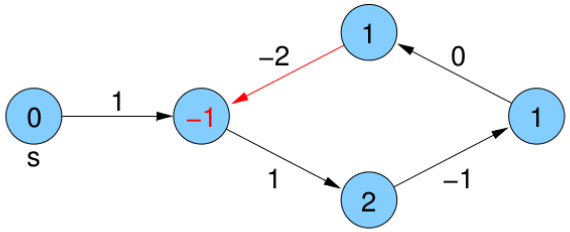
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

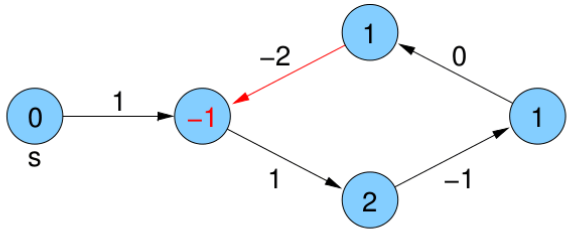
Problem: Erkennung negativer Kreise



Graphen Kürzeste Wege

Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



H. Seidl (TUM) GAD SS'16 494

Graphen Kürzeste Wege

Bellman-Ford-Algorithmus

Keine Distanzverringerung mehr möglich:

- Annahme: zu einem Zeitpunkt gilt für alle Kanten (v, w) $d[v] + c(v, w) \geq d[w]$

$\Rightarrow$ (per Induktion) für alle Knoten w und jeden Weg p von s nach w gilt: $d[s] + c(p) \geq d[w]$

- falls sichergestellt, dass zu jedem Zeitpunkt für kürzesten Weg p von s nach w gilt $d[w] \geq c(p)$, dann ist $d[w]$ zum Schluss genau die Länge eines kürzesten Pfades von s nach w (also korrekte Distanz)

Graphen Kürzeste Wege

Bellman-Ford-Algorithmus

Zusammenfassung:

- keine Distanzverringerung mehr möglich
  $(d[v] + c(v, w) \geq d[w]$ für alle $w)$:
  fertig, alle $d[w]$ korrekt für alle w

- Distanzverringerung 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 $-\infty$.

Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[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?
      parent[w] = v;
      infect(w);
}
```


Graphen Kürzeste Wege

Bellman-Ford-Algorithmus

BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[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?
      parent[w] = v;
      infect(w);
}

H. Seidl (TUM) GAD SS'16 498

Graphen Kürzeste Wege

Bellman-Ford-Algorithmus

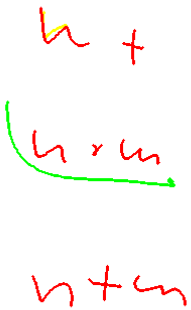
```
infect(Node v) {   // −∞-Knoten
  if (d[v] > −∞) {
    d[v] = −∞;
    foreach (e = (v, w) ∈ E)
      infect(w);
  }
}
```

Gesamtlaufzeit: $O(m \cdot n)$

H. Seidl (TUM) GAD SS'16 499

Graphen Kürzeste Wege

Bellman-Ford-Algorithmus



BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[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?
      parent[w] = v;
      infect(w);
}

H. Seidl (TUM) GAD SS'16 498

Bellman-Ford-Algorithmus

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

Gesamtlaufzeit: $O(m \cdot n)$

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

Bestimmung eines negativen Zyklus:

- bei den oben genannten Knoten sind vielleicht auch Knoten, die nur an negativen Kreisen über ausgehende Kanten angeschlossen sind, die selbst aber nicht Teil eines negativen Kreises sind
- Rückwärtsverfolgung der parent-Werte, bis sich ein Knoten wiederholt
- Kanten vom ersten bis zum zweiten Auftreten bilden einen negativen Zyklus

Bellman-Ford-Algorithmus

Idee der Updates vorläufiger Distanzwerte: Lester R. Ford Jr.

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

- benutze Queue von Knoten, zu denen ein kürzerer Pfad gefunden wurde und deren Nachbarn an ausgehenden 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) $\Rightarrow$ $\leq n-1$ Phasen

Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[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?
      parent[w] = v;
      infect(w);
}
```


Bellman-Ford-Algorithmus

Idee der Updates vorläufiger Distanzwerte: Lester R. Ford Jr.

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

- benutze Queue von Knoten, zu denen ein kürzerer Pfad gefunden wurde und deren Nachbarn an ausgehenden 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) $\Rightarrow$ $\leq n-1$ Phasen

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 (APSP)

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)

$\Rightarrow$ $O(n^2 \cdot m)$

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 (APSP)

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)

$\Rightarrow O(n^2 \cdot m)$

APSP / Kantengewichte

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