Graphen und Algorithmen

Vorlesung #3: Kürzeste Wege

Dr. Armin Fügenschuh
Technische Universität Darmstadt
WS 2007/2008

Übersicht

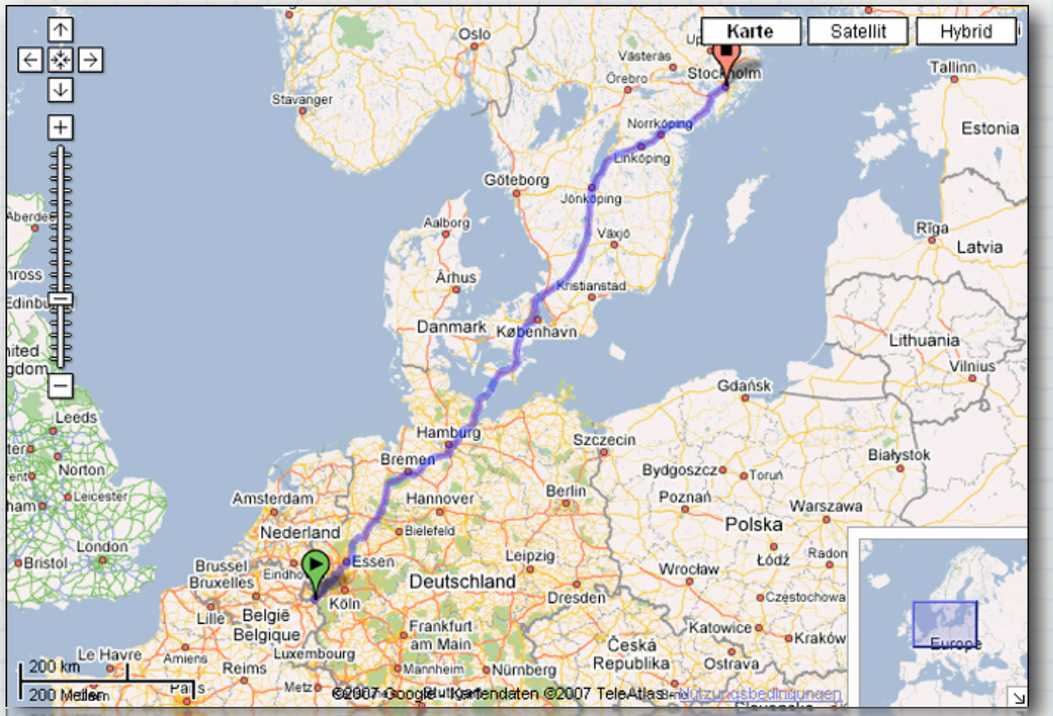
- Problemstellung
- Anwendungen
- Bellmans Optimalitätsbedingung und kürzeste Wege in azyklischen Digraphen
- Algorithmus von Dijkstra
- Algorithmus von Bellman-Ford
- Alle-Paare-Algorithmus von Floyd-Warshall
- Aufspannende Bäume und minimale Maximalkosten-Wege
- Steiner-Bäume

Das Kürzeste-Wege-Problem

- **Definition 1**:
  Sei $D = (V, A, c)$ ein bewerteter Digraph. Seien $s, t \in V$ Knoten in D. Die Aufgabe, einen Weg von s nach t mit minimaler Länge zu finden, wird als **Problem des kürzesten Weg** bezeichnet.
- In dieser Allgemeinheit gestellt, ist das Problem schwer zu lösen. Daher beschränken wir uns auf Spezialfälle.
- **Definition 2**:
  Sei $D = (V, A, c)$ ein bewerteter Digraph. Die Gewichte c heißen **konservativ**, falls es keinen Kreis mit negativem Gesamtgewicht gibt.
- Mit dieser Einschränkung wird das Problem zumindest für Digraphen einfach.
- Im Falle von ungerichteten Graphen beschränken wir uns zudem auf nicht-negative Gewichte.
- Besonders einfache (und schnelle) Algorithmen erhält man, wenn man weitere Einschränkungen ausnutzt, zum Beispiel für planare Graphen, oder Graphen, in denen alle Gewichte gleich sind.
- **Definition 3**:
  Sei $D = (V, A, c)$ ein konservativ bewerteter Digraph bzw. ein nicht-negativ bewerteter Graph. Das **alle-Paare-Kürzeste-Wege-Problem** fragt nach der Berechnung von Matrizen $(l_{i,j})_{i,j \in V}$ und $(p_{i,j})_{i,j \in V}$, wobei $l_{i,j}$ die Länge eines kürzesten Weges von i nach j ist und $p_{i,j}$ der letzte Knoten vor j auf einem solchen kürzesten Weg ist (falls er existiert).

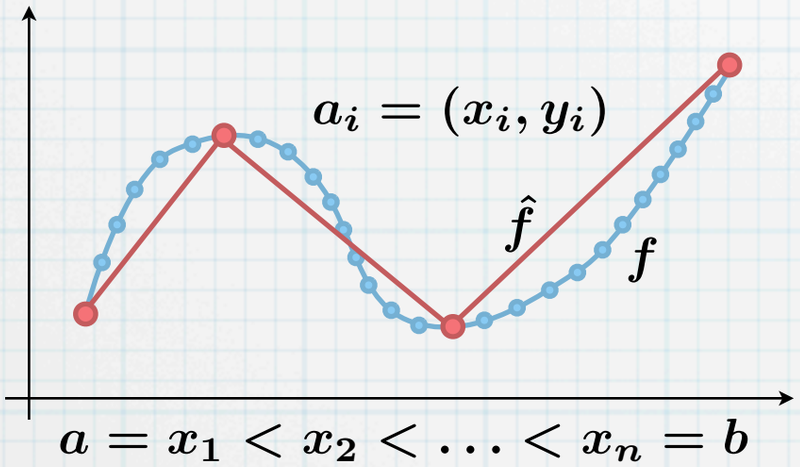
Anwendungen (1)

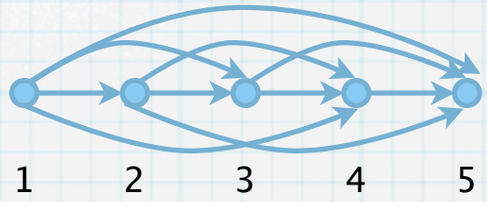
- Routenplanung im Straßenverkehr
- Digraph $D = (V, A, c)$ repräsentiert das Verkehrsnetz
  - Knoten sind Kreuzungen, Einmündungen
  - Bögen sind Straßen
  - Gewichte sind z.B. Fahrzeiten, Entfernungen oder Kosten (Maut)
- Praktische Herausfordungen:
  - Ein Internet-Dienst (wie z.B. GoogleMaps, Map24, Michelin) bekommt pro Sekunde ca. 1000 Anfragen.
  - Der Nutzer erwartet eine Antwort praktisch „sofort“ (< 1 Sekunde).
  - Der europäische Straßengraph besteht aus ca. 10 Mio. Knoten und 40 Mio. Bögen.



Anwendungen (2)

* Approximation stückweise linearer Funktionen (Imai u. Iri, 1986).
* Gegeben sei eine stetige Funktion $f : [a, b] \to \mathbb{R}$ in Form einer Wertetabelle. Zwischen zwei benachbarten Werten der Tabelle wird eine lineare Interpolation angenommen.
* Gesucht ist eine stückweise lineare Funktion $\hat{f} : [a, b] \to \mathbb{R}$, welche f möglichst gut approximiert, aber mit einer wesentlich kleineren Wertetabelle auskommt.
* Beispiel:



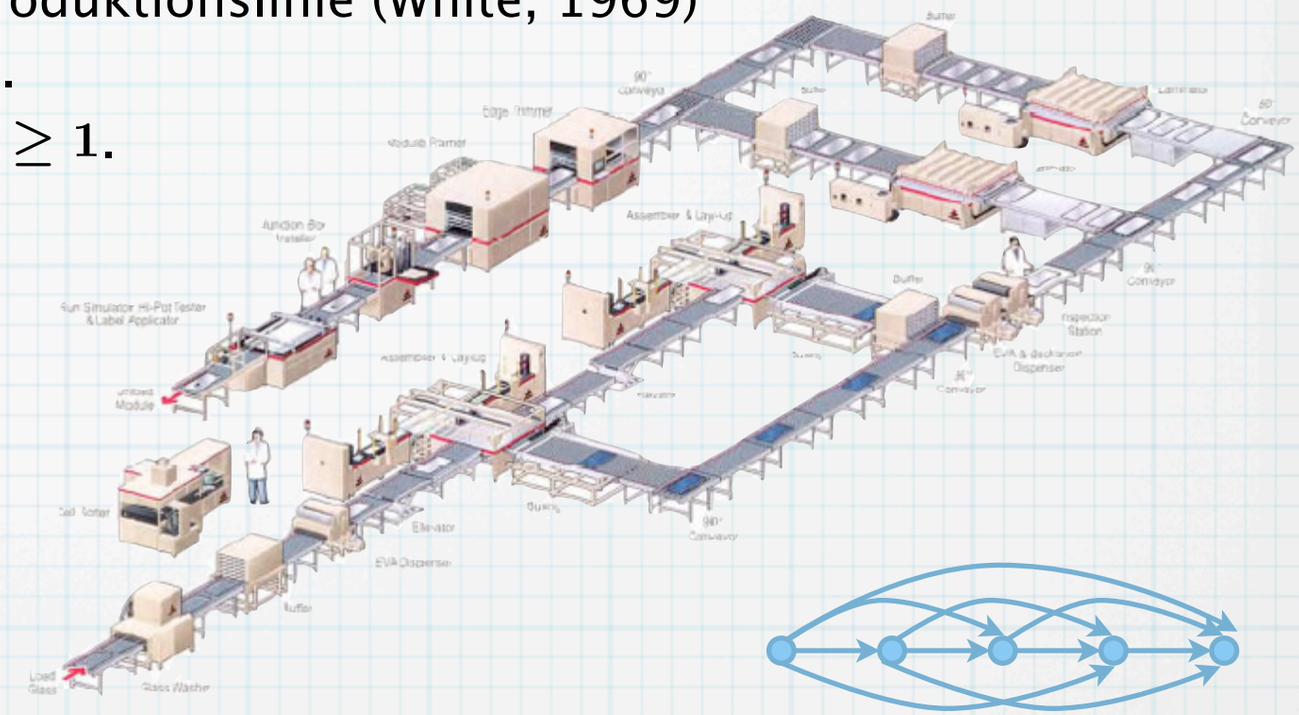


* Gewichteter Digraphen $D = (V, A, c)$, wobei $V = \{1, \ldots, n\}$ und $A = \{(i, j) : i < j\}$.
* Wähle Konstanten $\alpha, \beta > 0$ (α sind Kosten für das Segment und β sind Kosten für den Approximationsfehler) und definiere Gewichte

$$c_{i,j} := \alpha + \beta \left(\sum_{k=i}^{j} (f(x_k) - \hat{f}(x_k))^2 \right)$$

* Kürzester Weg von a nach b in D ist gesuchte Approximation.

Anwendungen (3)



- Planung von Inspektionen entlang einer Produktionslinie (White, 1969)
- n Fertigungsstationen innerhalb der Linie.
- Fertigung in Losen (batches) der Größe $B \geq 1$.
- Wahrscheinlichkeit für Produktionsfehler in Station i sei α_i; Fehler können nicht behoben werden.
- Nach jeder Station kann eine Prüfung des gesamten Loses erfolgen; die Prüfung erkennt fehlerhafte Teile.
- Am Ende der Linie erfolgt grundsätzlich eine Prüfung aller Teile.
- Problem: jede Prüfstation kostet Geld, Restfertigung von Schadteilen kostet ebenfalls Geld.
- Ziel: Finde das optimale Verhältnis (trade-off) zwischen diesen beiden Kostenfaktoren.
- Gegebene Größen: Herstellungskosten p_i pro Stück in Station i.
  Inspektionskosten $f_{i,j}$ pro Los nach Station j, falls zuletzt nach Station i inspiziert wurde.
  Inspektionskosten $g_{i,j}$ pro Stück nach Station j, falls zuletzt nach Station i inspiziert wurde.
- Gewichteter Digraphen $D = (V, A, c)$, wobei $V = \{0, 1, \ldots, n\}$ und $A = \{(i, j) : i < j\}$.
- Sei B_i die Anzahl heiler Teile im Los nach Station i, d.h. $B_i := B \cdot \prod_{k=1}^{i} (1 - \alpha_k)$.
- Ein kürzester Weg von 0 nach n bzgl. der Bogenkosten $c_{i,j}$ definiert eine optimale Verteilung von Inspektionsstellen, wobei

$$c_{i,j} := f_{i,j} + B_i \cdot \left(g_{i,j} + \sum_{k=i+1}^{j} p_k \right).$$

Bellmans Optimalitätskriterium

* **Satz 3**:
  Sei $D = (V, A, c)$ ein konservativ bewerteter Digraph. Seien $s, t \in V$ Knoten in D. Sei $P := (a_1, \ldots, a_{n-1}, a_n)$ ein kürzester Weg von s nach t, und sei $a_n = (u, t)$ der letzter Bogen in diesem Weg. Dann ist $P' := (a_1, \ldots, a_{n-1})$ ein kürzester Weg von s nach u.
* Beweis:

  Angenommen, $Q' := (b_1, \ldots, b_m)$ ist ein kürzerer Weg als P' von s nach u.

  Dann gilt $c(Q') + c_{a_n} < c(P)$.

  1. Fall, t ist nicht in Q' enthalten.

     Dann ist $(b_1, \ldots, b_m, a_n)$ ein kürzerer Weg von s nach t (im Vergleich zu P).

     Widerspruch zur Voraussetzung, dass P ein kürzester Weg ist.

  2. Fall, t ist in Q' enthalten.

     Sei b_k der Bogen von Q' mit Endknoten t.

     Teile Q' in Wege $R := (b_1, \ldots, b_k)$ von s nach t, und $S := (b_k, \ldots, b_m)$ von t nach u.

     Definiere $K := (b_k, \ldots, b_m, a_n)$, so ist K ein Kreis.

     Es gilt $c(K) \geq 0$, da der Digraph D konservativ ist.

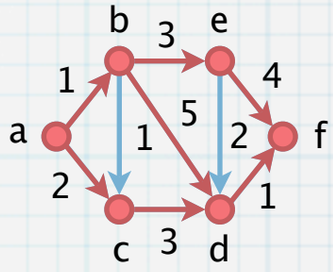
     Also gilt: $c(R) = c(Q') + c_{a_n} - c(K) < c(P) - c(K) \leq c(P)$.

     Ebenso im Widerspruch zur Voraussetzung, dass P ein kürzester Weg ist.

* **Folgerung 4**:
  Gegeben sei ein bewerteter azyklischer Digraph $D = (V, A, c)$ sowie ein Knoten $s \in V$.
  Wir definieren rekursiv die Funktion $dist : V \times V \to \mathbb{R}$ durch $dist(s, s) := 0$ und $dist(s, w) := \min\{dist(s, v) + c_{v,w} : (v, w) \in A\}$ für alle $w \in V \backslash \{s\}$.
  Dann liefert $dist$ den Abstand eines jeden Knoten des Digraphen von s.
* Bemerkung: Nach einer topologischen Sortierung erfolgt die Berechnung von $dist$ in $O(|A|)$.

Kürzeste-Wege-Algorithmus von Dijkstra

* Eingabe: gewichteter Graph $D = (V, A, c)$ mit $c_{i,j} \geq 0 \; \forall \, (i,j) \in A$, Startknoten $s \in V$.
* Ausgabe: Funktionen $l : V \to \mathbb{R} \cup \{\infty\}, p : V \to V$, wobei $l(v)$ die Länge eines kürzesten s–v–Weges und $p(v)$ der Vorgängerknoten von v auf diesem Weg ist.



$D = (V, A, c)$

$s = a$

```
(1)  algorithm dijkstra
(2)      l(s) := 0, l(v) := +∞ ∀v ∈ V\{s}, R := ∅
(3)      repeat
(4)          v := arg min{l(w) : w ∈ V\R}
(5)          R := R ∪ {v}
(6)          for alle w ∈ V\R mit (v, w) ∈ A do
(7)              if l(w) > l(v) + c_{v,w} then
(8)                  l(w) := l(v) + c_{v,w}
(9)                  p(w) := v
(10)             end if
(11)         end for
(12)     until R = V
(13) end algorithm
```

$v = a$

$R = \{\}$

$w = b$

$l(b) = \infty > l(a) + c_{a,b} = 0 + 1 = 1$

v	a	b	c	d	e	f
$l(v)$						

v	a	b	c	d	e	f
$p(v)$						

Korrektheit von Dijkstras Algorithmus (1)

* **Satz 5**:
  Wenn in Dijkstras Algorithmus für einen Knoten $v \in V$ in einer beliebigen Iteration der Wert $l(v)$ endlich ist, dann gibt es einen Weg von s nach v, dessen Länge $l(v)$ ist.
* Beweis:

  Für $v = s$ ist $l(s) = 0$, also ist die Aussage trivial.

  Sei nun $v \neq s$.

  Da $l(v)$ endlich, gibt es einen Knoten u_1 mit $(u_1, v) \in A$, $l(u_1) < \infty$ und $l(v) = l(u_1) + c_{u_1,v}$ (Schritt 8).

  u_1 ist dann in R (Schritt 5), und $l(u_1)$ ändert sich auch im weiteren Verlauf nicht mehr.

  Da $l(u_1)$ endlich, gibt es einen Knoten u_2 mit $(u_2, u_1) \in A$ und $l(u_1) = l(u_2) + c_{u_2,u_1}$.

  Dieses wiederholen wir so oft, bis wir bei s angekommen sind.

  Auf diese Weise erhalten wir einen Weg von s nach v der Länge $l(v)$, denn:

$$\begin{aligned} l(v) &= l(u_1) + c_{u_1,v} \\ &= l(u_2) + c_{u_2,u_1} + c_{u_1,v} \\ &= \dots \\ &= c_{s,u_n} + c_{u_n,u_{n-1}} + \dots + c_{u_2,u_1} + c_{u_1,v} \end{aligned}$$

  D.h., $l(v)$ ist genau die Summe der Bögen, also die Länge des Weges von s nach v.

  Dieses ist tatsächlich ein Weg (keine Wiederholung von Knoten), da u_1 nach u_2, u_2 nach u_3 usw. zu R hinzugefügt wurde.

Korrektheit von Dijkstras Algorithmus (2)

* **Satz 6**:
  Wenn ein Knoten v in Dijkstras Algorithmus in Schritt (4) gewählt wird, dann ist $l(v)$ die Länge $dist(s, v)$ eines kürzesten Weges von s nach v.
* Beweis:

  Klar ist (nach Satz 5): $l(v)$ ist eine obere Schranke auf die Länge, $l(v) \geq dist(s, v)$.

  Zeigen per Induktion über die Anzahl der Elemente in R, dass $l(v)$ genau diese Länge ist.

  Am Anfang ist diese Aussage trivial, $l(s) = 0$.

  Angenommen, wir wählen ein u in Schritt (4), und für alle vorher gewählten Knoten ist die Aussage richtig.

  1. Fall, $l(u) = \infty$.

  Da $l(s)$ endlich, muss es einen vorher gewählten Knoten $u' \neq s$ mit $l(u') = \infty$ geben.
  Sei u' zudem der erste derartige Knoten.
  Dann gilt wg. Schritt (8) für alle weiteren Knoten $v \in V \backslash R$ ebenso $l(v) = \infty$.
  Alle in Schritt (4) vorher gewählten Knoten $v' \in R \backslash \{u'\}$ haben ein endliches $l(v')$.
  Also gibt es keine Kante von einem v' mit $l(v') < \infty$ zu einem v mit $l(v) = \infty$,
  da sonst $l(v) = \mathbf{min}\{l(v), l(v') + c_{v',v}\} < \infty$ wäre, was ein Widerspruch ist.
  Also gibt es keinen Weg von s nach u, und die Setzung $l(u) = \infty$ ist korrekt.

Korrektheit von Dijkstras Algorithmus (3)

* **Satz 6**:
  Wenn ein Knoten v in Dijkstras Algorithmus in Schritt (4) gewählt wird, dann ist $l(v)$ die Länge $dist(s, v)$ eines kürzesten Weges von s nach v.
* Beweis:

  2. Fall, $l(u) < \infty$.

     Es gibt (nach Satz 5) einen Weg mit Länge $l(u)$. Also ist $l(u) \geq dist(s, u)$.
     Angenommen, es gibt einen kürzeren Weg $s = v_0, v_1, \ldots, v_k = u$.
     Dessen Länge ist dann $c_{v_0,v_1} + \ldots + c_{v_{k-1},v_k} = dist(s, u)$.
     Sei v_j der letzte Knoten auf diesem Weg, der vor u in Schritt (4) gewählt wurde.
     Nach Induktionsvoraussetzung gilt $l(v_j) = dist(s, v_j) = c_{v_0,v_1} + \ldots + c_{v_{j-1},v_j}$.

     1. Fall, $v_{j+1} \neq u\ (= v_k)$.

        Nachdem $l(v_j)$ den endgültigen Wert bekommen hat, ist $l(v_{j+1}) \leq l(v_j) + c_{v_j,v_{j+1}}$.
        Diese Beziehung gilt auch, nachdem $l(u)$ den endgültigen Wert hat. Also:
        $l(v_{j+1}) \leq l(v_j) + c_{v_j,v_{j+1}} = dist(s, v_j) + c_{v_j,v_{j+1}} \leq dist(s, u) < l(u)$.
        Es ist also $l(v_{j+1}) < l(u)$, aber u wurde nach Annahme vor v_{j+1} in Schritt (4) gewählt, was ein Widerspruch ist.

     2. Fall, $v_{j+1} = u$.

        Wie zuvor gilt, sobald $l(v_j)$ den endgültigen Wert hat:
        $l(u) = l(v_{j+1}) \leq l(v_j) + c_{v_j,v_{j+1}} = dist(s, v_j) + c_{v_j,v_{j+1}} = dist(s, u)$.
        Zusammen gilt $l(u) = dist(s, u)$.

Laufzeit von Dijkstras Algorithmus

* **Satz 7**:
  Der Algorithmus von Dijkstra hat eine Laufzeit in $O(|V|^2)$.
* Beweis:

  Die Knotenwahl in Schritt (4) wird $|V|$ mal durchgeführt, und es werden dabei alle Knoten in $V \backslash R$ geprüft. $V \backslash R$ wird in jedem Schritt ein Element kleiner.

  Damit ist die Komplexität $|V| + (|V| - 1) + (|V| - 2) + \ldots + 1 = O(|V|^2)$.

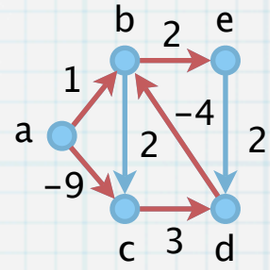
  Die for-Schleife in den Schritten (6)-(11) wird $\deg_{out}(v)$ oft durchlaufen für Knoten v.

  Damit ist die Komplexität $\sum_{v \in V} \deg_{out}(v) = |A|$.

  Da $|A| \leq |V|^2$, ist die Komplexität damit $O(|V|^2)$.
* Bemerkung: Dijkstras Algorithmus funktioniert genauso für Graphen.

Der Shimbel- (Moore-Bellman-Ford)-Algorithmus

* Eingabe: konservativ-gewichteter Digraph $D = (V, A, c)$, Startknoten $s \in V$.
* Ausgabe: Funktionen $l : V \to \mathbb{R} \cup \{\infty\}, p : V \to V$, wobei $l(v)$ die Länge eines kürzesten s–v–Weges und $p(v)$ der Vorgängerknoten von v auf diesem Weg ist.



$D = (V, A, c)$

$s = a$

```
(1)  algorithm mooreBellmanFord
(2)     l(s) := 0, l(v) := +∞ ∀ v ∈ V\{s}
(3)     for i from 1 to |V| − 1 do
(4)        for alle Bögen (v, w) ∈ A do
(5)           if l(w) > l(v) + c_{v,w} then
(6)              l(w) := l(v) + c_{v,w}
(7)              p(w) := v
(8)           end if
(9)        end for
(10)    end for
(11) end algorithm
```

$i = 1$

(a, b)

$l(b) = \infty > l(a) + c_{a,b} = 0 + 1 = 1$

v	a	b	c	d	e
$l(v)$					

v	a	b	c	d	e
$p(v)$					

Laufzeit und Korrektheit von Moore-Bellman-Ford

* **Satz 8**:
  Der Algorithmus von Moore–Bellman–Ford arbeitet korrekt. Seine Laufzeit ist in $O(|V| \cdot |A|)$.
* Beweis:
  * Aussage über Laufzeit ist klar, da (3) $|V|$ Mal und (4) $|A|$ Mal durchlaufen wird.
  * Sei $u \in V$. Wir zeigen, dass der Algorithmus einen kürzesten Weg von s nach u berechnet.
    Sei $P = (v_0, v_1, \ldots, v_k)$ die Knotensequenz eines solchen Weges.
    Da die Gewichte des Graphen konservativ sind, ist P ein einfacher Weg, d.h. $k \leq |V| - 1$.
    Zeige nun per Induktion, dass nach der i–ten Iteration von (3) ein kürzester Weg bis v_i berechnet ist (dies kann P sein, muss es aber nicht).
    Induktionsanfang: für $v_0 = s$ ist dies sicher richtig.
    Angenommen nun, die Aussage gilt für $i - 1$. Dann hat der Algorithmus einen kürzesten Weg bis v_{i-1} der Länge $l(v_{i-1})$ berechnet.
    Dann gilt nach der i–ten Iteration: $l(v_i) \leq l(v_{i-1}) + c_{v_{i-1},v_i}$.
    Da $l(v_{i-1})$ die Länge eines kürzesten Weges nach v_{i-1} ist, und c_{v_{i-1},v_i} die kürzeste Verbindung von v_{i-1} nach v_i (da in P, einem kürzesten Weg), ist $l(v_i) < l(v_{i-1}) + c_{v_{i-1},v_i}$ nicht möglich.
    Also ist $l(v_i) = l(v_{i-1}) + c_{v_{i-1},v_i}$ damit die Länge eines kürzesten Weges nach v_i.

Wie man negative Kreise findet

* Eingabe: gewichteter Digraph $D = (V, A, c)$, Startknoten $s \in V$, Funktion $l : V \to \mathbb{R} \cup \{\infty\}$ aus Moore-Bellman-Ford-Algorithmus.
* Ausgabe: Meldung, dass negativer Kreis von s aus erreichbar oder Meldung, dass kein solcher Kreis existiert.

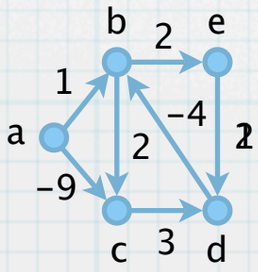
```
(1)  algorithm negativeCycle
(2)     for alle Bögen (v, w) ∈ A do
(3)       if l(w) > l(v) + c_{v,w} then
(4)          Meldung: „negativer Kreis“
(5)          goto (9)
(6)       end if
(7)     end for
(8)     Meldung: „kein negativer Kreis“
(9)  end algorithm
```

(a, b)

$$l(b) = -11 > l(a) + c_{a,b} = 0 + 1 = 1$$

negativer Kreis

$s = a$



$D = (V, A, c)$

v	a	b	c	d	e
$l(v)$	0	-10	-9	-6	-8

Korrektheitsbeweis

* **Satz 9**:
  Der Algorithmus negativeCycle arbeitet korrekt. Seine Laufzeit ist in $O(|A|)$.
* Beweis:

  Angenommen, der Algorithmus arbeitet nicht korrekt.

  Dann gibt es einen negativen, von s aus erreichbaren Kreis, aber der Algorithmus gibt die Meldung „kein Kreis gefunden“ aus.

  Sei $C := \{v_0, v_1, \ldots, v_k\}$ ein solcher Kreis. Dann gilt $\sum_{i=1}^{k} c_{v_{i-1},v_i} < 0$.

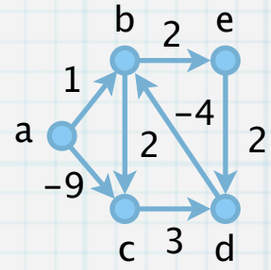
  Da negativeCycle keinen negativen Kreis gefunden hat, gilt $l(v_i) \leq l(v_{i-1}) + c_{v_{i-1},v_i}$ für alle $i = 1, \ldots, k$.

  Summiert man diese auf, so erhält man $\sum_{i=1}^{k} l(v_i) \leq \sum_{i=1}^{k} l(v_{i-1}) + \sum_{i=1}^{k} c_{v_{i-1},v_i}$.

  Es folgt $0 \leq \sum_{i=1}^{k} c_{v_{i-1},v_i}$. Widerspruch!

Alle-Paare-Kürzeste-Wege-Algorithmus von Floyd-Warshall

* Eingabe: konservativ–gewichteter Digraph $D = (V, A, c)$
* Ausgabe: Matrizen $(l_{i,j})_{i,j \in V}$ und $(p_{i,j})_{i,j \in V}$, wobei $l_{i,j}$ die Länge eines kürzesten Weges von i nach j ist und $p_{i,j}$ der letzte Knoten vor j auf einem solchen kürzesten Weg ist.



$D = (V, A, c)$

```
(1)  algorithm floydWarshall
(2)      l_{i,j} := c_{i,j} ∀ (i,j) ∈ A
(3)      l_{i,j} := +∞ ∀ (i,j) ∈ V × V\A, i ≠ j
(4)      l_{i,i} := 0 ∀ i ∈ V
(5)      p_{i,j} := i ∀ (i,j) ∈ A
(6)      for alle Knoten k ∈ V do
(7)        for alle Knoten i ∈ V\{k} do
(8)          for alle Knoten j ∈ V\{i,k} do
(9)            if l_{i,j} > l_{i,k} + l_{k,j} then
(10)             l_{i,j} := l_{i,k} + l_{k,j}
(11)             p_{i,j} := p_{k,j}
(12)           end if
(13)         end for
(14)       end for
(15)     end for
(16) end algorithm
```

$k = a$

$i = b$

$j = c$

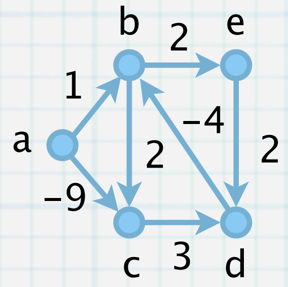
$l_{b,c} = 2 > l_{b,a} + l_{a,c} = \infty - 9 = \infty$

$l_{i,j}$	a	b	c	d	e
a					
b					
c					
d					
e					

$p_{i,j}$	a	b	c	d	e
a					
b					
c					
d					
e					

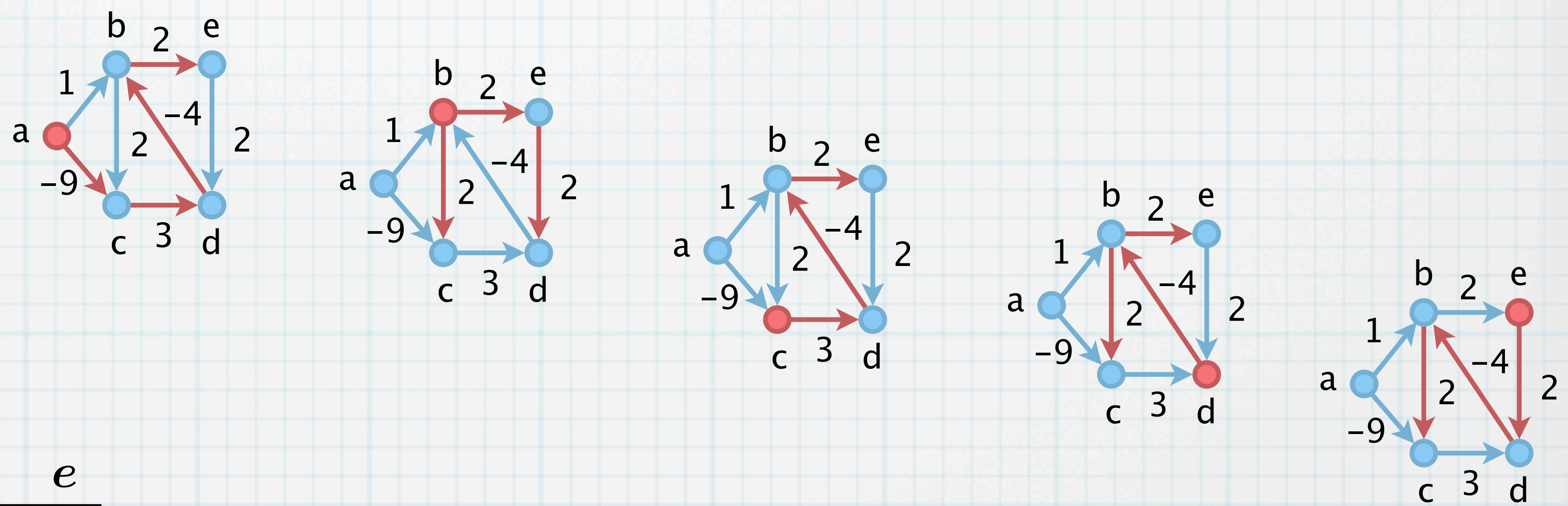
Ausgabe des Floyd-Warshall-Algorithmus

* Jede Zeile der Matrix $(p_{i,j})_{i,j \in V}$ beschreibt einen kürzeste-Wege-Baum.



$D = (V, A, c)$

$p_{i,j}$	a	b	c	d	e
a		d	a	c	b
b			b	e	b
c		d		c	b
d		d	b		b
e		d	b	e	



$l_{i,j}$	a	b	c	d	e
a	0	-10	-9	-6	-8
b	∞	0	2	4	2
c	∞	-1	0	3	1
d	∞	-4	-2	0	-2
e	∞	-2	0	2	0

Korrektheitsbeweis zu Floyd-Warshall

* **Satz 10**:
  Der Algorithmus von Floyd–Warshall arbeitet korrekt. Seine Laufzeit ist in $O(|V|^3)$.
* Beweis:
  * Komplexitätsaussage klar, da in (6), (7), (8) jeweils eine for–Schleife mit $O(|V|)$ Schritten ausgeführt wird, und diese Schleifen ineinander geschachtelt sind.
  * Per Induktion zeigen wir: nach Iteration k von Schleife (6) enthält $(l_{i,j})_{i,j \in V}$ die Längen von kürzesten i–j–Wegen mit Zwischenknoten $v \in \{1, \ldots, k\}$ und $(p_{i,j})_{i,j \in V}$ enthält die jeweils letzten Knoten vor j auf einem solchen Weg. (O.B.d.A. sei $V = \{1, \ldots, n\}$.)

    Für $k = 0$ (vor dem ersten Schleifendurchlauf) stimmt die Aussage; für $k = n$ folgt die Korrektheit des Algorithmus. Sei die Aussage wahr für $k - 1$.

    Während des k–ten Durchlaufs von Schleife (6) wird $l_{i,j}$ durch $l_{i,k} + l_{k,j}$ ersetzt, wenn letzteres kleiner ist.

    $l_{i,k}$ ist Länge eines kürzesten i–k–Weges P und $l_{k,j}$ ist Länge eines kürzesten k–j–Weges Q über die Zwischenknoten $1, \ldots, k-1$.

    Zu zeigen bleibt: diese beiden Wege P, Q haben keine gemeinsamen Knoten (außer k).

    Angenommen, es gibt einen solchen Zwischenknoten v. Betrachte den i–j–Weg über v.

    Dieser hat nur Zwischenknoten $1, \ldots, k-1$, aber (da die Gewichte konservativ) eine Länge von höchstens $l_{i,k} + l_{k,j}$, und ist insbesondere kürzer als $l_{i,j}$ nach der $k-1$ Iteration.

    Widerspruch zur Induktionsannahme.

Das alle-Paare minimax Wege Problem in Graphen

* **Definition 11**:
  Sei $G = (V, E, c)$ ein bewerteter Graph. Im **alle-Paare minimax Wege Problem** ist der Wert eines Weges P zwischen Knoten i und j definiert als der Wert der teuersten Kante in P (Maximalkosten). Gesucht ist für alle Knotenpaare i, j ein Weg mit minimalen Maximalkosten.
* Anwendungen:
  * Routenplanung für Rollstuhlfahrer (minimiere die maximale Steigung).
  * Reise durch die Wüste (minimiere den maximalen Weg zwischen zwei Wasserstellen).
  * Wiedereintritt einer Weltraumkapsel in die Erdatmosphäre (minimiere die maximale Temp.).
* **Satz 12**:
  Die eindeutigen Pfade zwischen allen Knotenpaaren i, j in einem minimalen Spannbaum T für G sind minimax Wege.
* Beweis:

  Sei P der eindeutige i–j–Weg in T und $\{p, q\}$ die Kante mit maximalen Kosten $c_{p,q}$ in P.

  Löscht man die Kante $\{p, q\}$ in T, so wird ein Fundamentalschnitt $p \in V_1, q \in V_2$ definiert.

  Dieser Schnitt hat nach der Schnitt–Optimalitätsbedingung (VL2, Satz 18) die Eigenschaft, dass Kante $\{p, q\}$ die billigste Kante ist, also $c_{p,q} \leq c_{s,t}$ für alle $s \in V_1, t \in V_2$.

  Betrachte einen beliebigen i–j–Weg P' in G. Dieser Weg muss eine Kante aus dem Fundamentalschnitt V_1, V_2 enthalten.

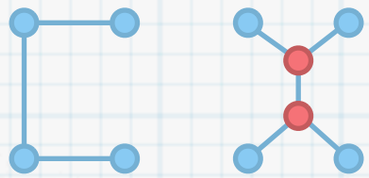
  Also ist der Wert von P' mindestens $c_{p,q}$.

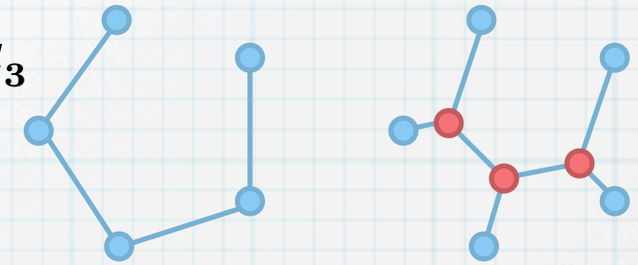
  Da der Wert genau $c_{p,q}$ für Weg P ist, ist letzterer also ein optimaler minimax–Weg.

Euklidische Steinerbäume und Steinerbäume in Graphen

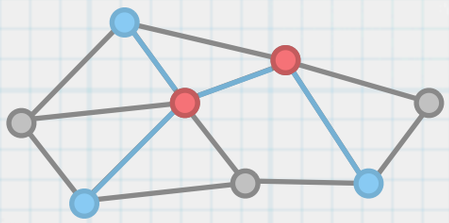
* **Definition 13**:
  Gegeben seien n Punkte in der Ebene (mit dem euklidischen Abstand). Das Problem, diese Punkte zu einem Netz minimaler Gesamtlänge zu verbinden, wobei zusätzlich beliebig viele weitere Punkte (**Steinerpunkte** genannt) eingeführt werden dürfen, wird als **euklidisches Steinerbaum-Problem** bezeichnet.
* Beispiele:

G_1 

G_2 

G_3 

* Steinerbaum-Verhältnis: $\varrho =$ Gesamtlänge Steinerbaum / Gesamtlänge min. Spannbaum
* Satz von Du und Hwang (1990): $\varrho \geq \sqrt{3}/2 \approx 0,86..$
* **Definition 14**:
  Gegeben sei ein bewerteter Graph $G = (V, E, c)$ mit positiven Kantengewichten. Sei $V = R \cup S, R \cap S = \emptyset$. Beim **Graphen-Steinerbaum-Problem** ist eine Teilmenge $S' \subseteq S$ gesucht, so dass der minimale aufspannende Baum auf dem induzierten Untergraphen $G|(R \cup S')$ minimales Gewicht hat. Die Knoten in S' heißen **Steinerknoten**.
* Beispiel:



$G = (V, E)$

$R =$
$S' =$
$S \backslash S' =$

* Spezialfälle: Für $S = \emptyset$ ist der Steinerbaum ein minimaler aufspannender Baum.
* Für $|R| = 2$ ist der Steinerbaum ein kürzester Weg zwischen den beiden Knoten von R.

Abschätzung für die Anzahl der Steinerpunkte

* **Definition 15**:
  Ein **metrischer Graph** ist ein bewerteter Graph $G = (V, E, c)$ mit positiven Gewichten, die der **Dreiecksungleichung** genügen, d.h. für alle Knoten $i, j, k \in V$ mit $\{i, j\}, \{i, k\}, \{k, j\} \in E$ gilt $c_{i,j} \leq c_{i,k} + c_{k,j}$.
* **Satz 16**:
  Sei $G = (V, E, c)$ ein vollständiger metrischer Graph und $V = R \cup S, R \cap S = \emptyset$. Dann gibt es einen Steinerbaum T für G, der höchstens $|R| - 2$ Steinerknoten enthält (also $|S'| \leq |R| - 2$).
* Beweis:

  Sei x bzw. y der durchschnittliche Grad eines Knotens in R bzw. S'.

  Trivialerweise ist $x \geq 1$.

  Jeder Steinerknoten in S' liegt auf mindestens drei Kanten, da der Graph vollständig und metrisch ist. Also $y \geq 3$.

  Für die Anzahl Kanten in T gilt $|R| + |S'| - 1 = \frac{1}{2}(|R| \cdot x + |S'| \cdot y) \geq \frac{1}{2}(|R| + 3 \cdot |S'|)$.
* **Satz 17**:
  Sei $G = (V, E, c)$ ein bewerteter Graph mit positiven Gewichten. Sei $(d_{i,j})_{i,j \in V}$ die Matrix der Längen der kürzesten Wege. Sei $V = R \cup S, R \cap S = \emptyset$. Dann ist das Gewicht eines Steinerbaums T' in (K_V, d) gleich dem Gewicht eines Steinerbaums T in G.
* Beweis:

  Da $c_{i,j} \geq d_{i,j}$, gilt $c(T) \geq d(T')$.

  Umgekehrt: ersetzt man in T' jede Kante $\{i, j\}$ durch die Kanten eines kürzesten i–j–Weges in G, so erhält man einen Steinerbaum T'' gleichen Gewichts für G.

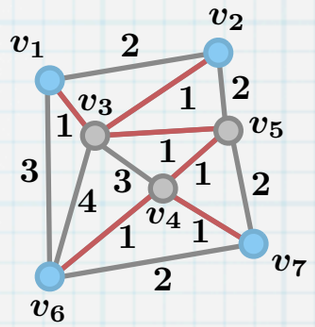
  Kantendopplungen und Kreise können dabei nicht auftreten: ließe man solche Kanten weg, würde man einen Steinerbaum mit $c(T'') < d(T')$ erhalten, Widerspruch.

  Also gilt $c(T) \geq d(T') = c(T'')$, und da $c(T) = c(T'')$, folgt $c(T) = d(T')$.

Lawlers Algorithmus für Steinerbäume in Graphen

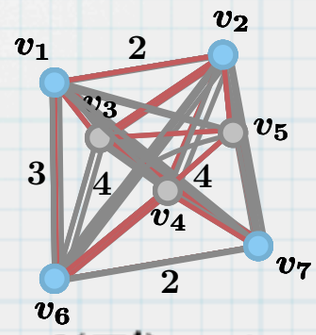
* Eingabe: gewichteter Graph $G = (V, E, c)$ mit $c_{i,j} > 0 \; \forall \; \{i, j\} \in E$, Aufteilung der Knotenmenge $V = R \cup S, R \cap S = \emptyset$.
* Ausgabe: minimaler Steinerbaum T für R in G.

(1) **algorithm** graphSteinerTree
(2) $W := +\infty$
(3) $((d_{i,j}), (p_{i,j})) :=$ floydWarshall(G)
(4) **for** i **from** 0 **to** $|R| - 2$ **do**
(5) **for** alle $S' \subseteq S$ mit $|S'| = i$ **do**
(6) $T' :=$ jarnik($K_V|(R \cup S'), d$)
(7) **if** $d(T') < W$ **then** $W := d(T'), T := T'$ **end if**
(8) **end for**
(9) **end for**
(10) **for** $\{i, j\} \in T$ **do**
(11) **if** $\{i, j\} \notin E$ or $c_{i,j} > d_{i,j}$ **then**
(12) $E(T) := E(T) \backslash \{\{i, j\}\}$
(13) $E(T) := E(T) \cup \{\{u, v\}$ Kante eines kürzesten i–j–Weges in $G\}$
(14) **end if**
(15) **end for**
(16) **end algorithm**



	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1		2	1			3	
v_2	2		1		2		
v_3	1	1		3	1	4	
v_4			3		1	1	1
v_5		2	1	1			2
v_6	3		4	1			2
v_7				1	2	2	

$W = \infty$
$i = 0$
$S' = \{\}$
$\{v_1, v_3\}$



$d(T') = 7$

Korrektheit und Komplexität

* **Satz 18**:
  Lawlers Algorithmus arbeitet korrekt. Seine Laufzeitkomplexität ist in $O(|V|^3 + 2^{|S|} \cdot |R|^2)$.
* Beweis:

  Die Korrektheit folgt unmittelbar aus Satz 16 und dem Beweis von Satz 17.

  Der Aufruf des Floyd-Warshall-Algorithmus trägt $O(|V|^3)$ zur Laufzeit bei.

  Jeder Aufruf des Algorithmus von Jarnik (Prim) hat eine Komplexität von $O(|R|^2)$, da wegen Satz 16 dieser Algorithmus auf $O(|R|)$ viele Punkte angewandt wird.

  Die Anzahl der Aufruf (Anzahl der Iterationen von Schritt 4 und 5 zusammen) kann abgeschätzt werden durch:
  $\sum_{i=0}^{|R|-2} \binom{|S|}{i} \leq 2^{|S|}$.
  Daraus ergibt sich die Aussage über die Komplexität.
* Folgerung: Für eine feste Anzahl von $|S|$ bzw. $|R|$ ist das Steinerbaumproblem in Graphen polynomial lösbar (bzw. ist Lawlers Algorithmus effizient).

Literaturquellen

* R.K. Ahuja, T.L. Magnanti, J.B. Orlin, **Network Flows – Theory, Algorithms, and Applications**, Prentice Hall, Upper Saddle River, 1993. (Kapitel 4&5, Seite 93–165)
* J. Clark, D.A. Holton, **Graphentheorie**, Spektrum Akademischer Verlag, Heidelberg, 1994. (Kapitel 2, Seite 75–85)
* W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, A. Schrijver, **Combinatorial Optimization**, Wiley Interscience, New York, 1998. (Kapitel 2, Seite 19–36)
* T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, **Introduction to Algorithms**, 2nd Edition, McGraw-Hill Book Company, Boston, 2001. (Kapitel 24&25, Seite 580–642)
* J. Erickson, **Algorithms**, Lecture Notes, University of Illinois at Urbana-Champaign, 2007. (Kapitel 13&14)
* D. Jungnickel: Graphen, **Netzwerke und Algorithmen**, BI Wissenschaftsverlag, Mannheim, 1994. (Kapitel 3, Seite 89–132)
* B. Korte, J. Vygen, **Combinatorial Optimization – Theory and Algorithms**, 2nd Edition, Springer Verlag, Berlin, 2001. (Kapitel 7, Seite 139–152)
* A. Schrijver, **Combinatorial Optimization – Polyhedra and Efficiency**, Springer Verlag, Berlin, 2003. (Kapitel 6–8, Seite 87–130)