SS 2021

Algorithmen und Datenstrukturen

12. Kapitel

Dynamische Programmierung

Martin Dietzfelbinger

Juli 2021

Kapitel 12: Dynamische Programmierung (DP)

Algorithmenparadigma für **Optimierungsprobleme**. Typische Schritte:

- Definiere (viele) **„Teilprobleme“** (einer Instanz)
- Identifiziere einfache **Basisfälle**
- Formuliere eine Version der Eigenschaft

> **Substrukturen optimaler Strukturen sind optimal**

- Finde Rekursionsgleichungen für Werte optimaler Lösungen:

> **Bellmansche Optimalitätsgleichungen**

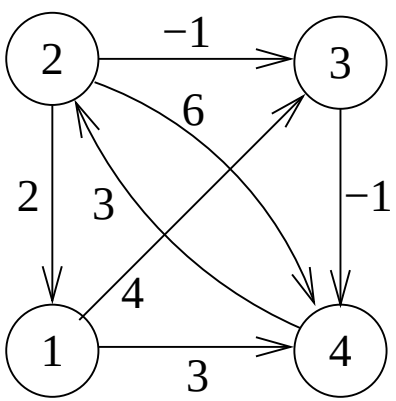
- Berechne optimale Werte (und Strukturen) **iterativ**.

12.1 Das All-Pairs-Shortest-Paths-Problem

Das „**APSP-Problem**“ ist zentrales Beispiel für die Strategie „Dynamische Programmierung“.

„Kürzeste Wege zwischen allen Paaren von Knoten“.

Eingabe:



Digraph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ und $E \subseteq \{(v, w) \mid 1 \leq v, w \leq n, v \neq w\}$,

$c\colon E \to \mathbb{R} \cup \{+\infty\}$: **Gewichts-/Kosten-/Längenfunktion**.

Die **Länge** eines Kantenzugs $p = (v = v_0, v_1, \ldots, v_r = w)$ ist

$$c(p) = \sum_{1 \le s \le r} c(v_{s-1}, v_s).$$

Gesucht: für alle v, w, $1 \le v, w \le n$:
Kantenzug p von v nach w mit minimalem $c(p)$ („kürzester Weg“).

Erinnerung: **Single-Source-Shortest-Paths-Problem (SSSP-Problem)**.
Im Fall **nichtnegativer** Kantengewichte löst n-maliges Aufrufen des Algorithmus von **Dijkstra** das APSP-Problem. Rechenzeit: $n \cdot O(m \log n) = O(nm \log n)$.

Der im Folgenden beschriebene **Algorithmus von Floyd–Warshall*** kann auch mit negativen Kantengewichten umgehen.

* Drei unabh. Publikationen: 1962 R. W. Floyd (1936–2001), amer. Informatiker; 1959 Stephen Warshall (1935–2006), amer. Informatiker; 1959 Bernard Roy (1934–2017), frz. Mathematiker

Wir verlangen aber: Es gibt keine **Kreise** mit (strikt) negativer Gesamtlänge, d. h.

$$(*) \qquad v = v_0, v_1, \ldots, v_r = v \text{ Kreis } \Rightarrow \sum_{1 \le s \le r} c(v_{s-1}, v_s) \geq 0.$$

Grund für diese Annahme: Wenn es einen solchen Kreis von v nach v gibt, und irgendein Weg von v nach w existiert, dann gibt es Kantenzüge von v nach w mit beliebig stark negativer Länge. Das heißt: Die Frage nach einem „kürzesten Weg“ von v nach w ist sinnlos.

Konsequenzen aus $(*)$:

(1) Wenn p Kantenzug von v nach w ist, dann existiert auch ein (einfacher) **Weg** p' von v nach w mit $c(p') \leq c(p)$.

(Begründung: Wenn p Segment $(u, \ldots, u)$ enthält, ersetze es durch (u). Hierdurch kann sich die Länge des Kantenzugs nicht vergrößern, weil es keine negativen Kreise gibt. Iteriere dies, bis alle Knotenwiederholungen beseitigt sind.)

(2) Wenn es einen Kantenzug von v nach w gibt, dann auch einen mit minimaler Länge (einen **„kürzesten Weg“**).

(Begründung: Wegen (1) muss man nur unter den (einfachen) **Wegen** von v nach w (mit nicht mehr als $n - 1$ Kanten) einen kürzesten finden; davon gibt es nur endlich viele.)

Bemerkungen:

- Es kann mehrere „kürzeste Wege“ von v nach w geben.
  (Selbst wenn alle Kantenlängen 1 sind, kann es mehrere Wege von v nach w mit gleich vielen „hops“ geben.)
- Deshalb: „Ein“ kürzester Weg, nicht „der“ kürzeste Weg.
- Wenn $G = (V, E, c)$ einen Kreis $(v, \ldots, v)$ mit Kosten 0 hat, der mindestens eine Kante hat, dann gibt es sogar unendlich viele Kantenzüge von v nach w von minimaler Länge.
  (Durchlaufe den Kreis mehrfach.)

Für Digraphen $G = (V, E, c)$, mit $(*)$, definieren wir:

$$\boxed{\boldsymbol{S(v,w)} := \text{die Länge eines kürzesten Weges von } v \text{ nach } w.}$$

(Wenn $v \not\rightsquigarrow w$, setzen wir $S(v, w) := \infty$.)

APSP-Problem, vereinfacht: Bestimme $S(v, w)$, für alle $v, w \in V$.

Gewünschte **Ausgabe:** Matrix $S = (S(v, w))_{1 \leq v, w \leq n}$.

O. B. d. A.: $E = V \times V - \{(v,v) \mid v \in V\}$. (Wenn $(v,w) \notin E$, setzen wir $c(v,w) = \infty$.)

Erster Schritt: Identifiziere geeignete **Teilprobleme**.

Gegeben ein k, $0 \leq k \leq n$, betrachte nur (einfache) Wege von v nach w, die **unterwegs** (also vom Start- und Endpunkt abgesehen) nur Knoten in $\{1, \ldots, k\}$ besuchen (nur Wege $(v, v_1, \ldots, v_{r-1}, w)$ mit $v_1, \ldots, v_{r-1} \in \{1, \ldots, k\}$).

$k = 0$: Kein Knoten darf unterwegs besucht werden; es handelt sich nur um die Kanten (v,w) und um Wege (v) (keine Kante, Kosten 0), wenn $v = w$.

$k = 1$: Die Wege aus dem Fall „$k = 0$“ und zusätzlich Wege $(v, 1, w)$.

$k = 2$: Die Wege aus dem Fall „$k = 1$“ und Wege $(v, 1, 2, w)$, $(v, 2, 1, w)$, $(v, 2, w)$.

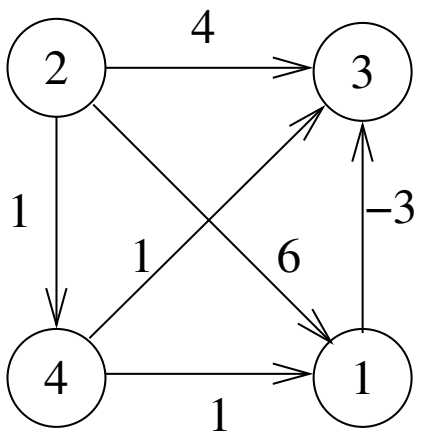
Solche Wege nennen wir ***k*-Wege** von v nach w.

Für $1 \leq v, w \leq n$ und $0 \leq k \leq n$ definieren wir:

$$\boldsymbol{S(v,w,k)} := \text{minimale Länge } \sum_{1 \leq i \leq r} c(v_{i-1}, v_i) \text{ eines } k\text{-Weges } (v = v_0, \ldots, v_r = w) \text{ von } v \text{ nach } w.$$

Klar ist: $S(v,w) = S(v,w,n)$, für alle v, w.

Beispiel: (Die nicht eingezeichneten Kanten haben Länge ∞.)



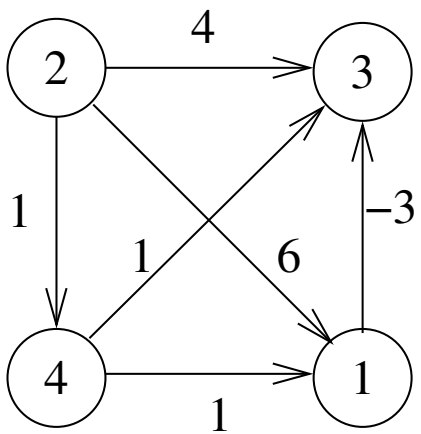
0-Weg von 2 nach 3: $(2, 3)$.

1-Wege: zusätzlich $(2, 1, 3)$.

2-Wege und 3-Wege: Keine Änderung; Endpunkte können auf Wegen nicht im Innern vorkommen.

4-Wege: zusätzlich $(2, 4, 3)$ und $(2, 4, 1, 3)$.

Beispiel: (Die nicht eingezeichneten Kanten haben Länge ∞.)



$S(2,3,0) = 4$
$S(2,3,1) = 3$
$S(2,3,2) = 3$
$S(2,3,3) = 3$
$S(2,3,4) = -1$

monoton fallend

0-Weg von 2 nach 3: $(2, 3)$.

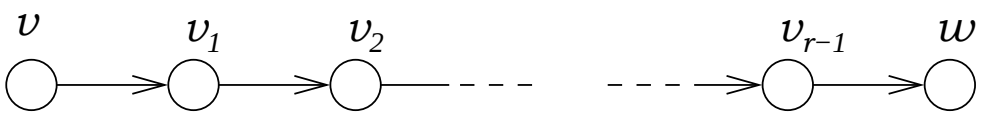
1-Wege: zusätzlich $(2, 1, 3)$.

2-Wege und 3-Wege: Keine Änderung; Endpunkte können auf Wegen nicht im Innern vorkommen.

4-Wege: zusätzlich $(2, 4, 3)$ und $(2, 4, 1, 3)$.

Wir entwickeln jetzt „Bellmansche Optimalitätsgleichungen“ für dieses Problem.

Für $k \geq 1$ betrachte k-Weg p von v nach w mit minimalen Kosten („**kürzester k-Weg**“):



Knoten k kommt im Inneren von p entweder einmal oder gar nicht vor.

1) Falls k in p **nicht** vorkommt, ist p kürzester $(k-1)$-Weg von v nach w.
2) Falls k im Inneren von p **vorkommt**, zerfällt p in zwei Teile

die beide kürzeste $(k-1)$-Wege sind. (Sonst könnte ein Teilweg durch einen billigeren ersetzt werden, im Widerspruch zur Optimalität von p.)

Punkte 1) und 2) drücken aus, dass Substrukturen (Teilwege) einer optimalen Lösung für ein Teilproblem (hier: kürzester k-Weg) selbst wieder optimal sind, für ein anderes Teilproblem.

Um beide Fälle zu erfassen, nehmen wir das Minimum der beiden Möglichkeiten.

Die **Bellmanschen Optimalitätsgleichungen** für den Algorithmus von Floyd-Warshall lauten dann:

$$S(v,w,k) = \min\{S(v,w,k-1), S(v,k,k-1) + S(k,w,k-1)\},$$
$$\text{für } 1 \leq v,w \leq n, 1 \leq k \leq n.$$

Basisfälle:

$S(v,v,0) = 0.$ (Weg (v), Länge 0.)

$S(v,w,0) = c(v,w)$, für $v \neq w.$ (Kante (v,w).)

Obgleich die Bellmanschen Optimalitätsgleichungen scheinbar ein rekursives Programm nahelegen, wird das Programm iterativ aufgeschrieben. Zur Aufbewahrung der Zwischenergebnisse benutzen wir ein Array $\texttt{S}[1..n,1..n,0..n]$.

Eine iterative Programmstruktur mit einem Array als Datenstruktur für die Zwischenergebnisse ist ganz typisch für das Paradigma „dynamische Programmierung“. Naive Umsetzung in ein rekursives Programm würde dazu führen, dass Zwischenergebnisse wiederholt ausgerechnet werden, mit katastrophalen Folgen für die Rechenzeit.

Initialisierung:

$$\begin{aligned} &\texttt{S}[v,v,0] \leftarrow 0, && 1 \le v \le n; \\ &\texttt{S}[v,w,0] \leftarrow c(v,w), && 1 \le v,w \le n, \quad v \neq w. \end{aligned}$$

Der Algorithmus füllt dann das Array S gemäß wachsendem k aus:

```
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w,k] ← min{S[v,w,k-1], S[v,k,k-1] + S[k,w,k-1]}.
```

Korrektheit: Folgt aus der Vorüberlegung. Im Array S werden genau die Werte $S(v, w, k)$ gemäß den Basisbedingungen und den Bellmanschen Optimalitätsgleichungen berechnet.

Rechenzeit: Drei geschachtelte Schleifen: $\Theta(n^3)$.

Bei dieser simplen Implementierung wird noch Platz verschwendet, da für den k-ten Schleifendurchlauf offenbar nur die $(k-1)$-Komponenten des Arrays S benötigt werden.

Man überlegt sich leicht, dass man mit zwei Matrizen auskommt, einer mit den „alten“ Werten ($(k-1)$-Version) und einer mit den „neuen“ Werten (k-Version), zwischen denen dann immer passend umzuschalten ist.

Der Platzaufwand beträgt dann $O(n^2)$, der Zeitbedarf $O(n^3)$.

Wir können den Speicherplatzbedarf und die Berechnung selbst noch vereinfachen. Es gilt nämlich

$$S(v,k,k) = S(v,k,k-1) \text{ und } S(k,w,k) = S(k,w,k-1), \text{ für } 1 \leq v,w,k \leq n.$$

(k kann nicht im Inneren eines k-Weges von v nach k bzw. von k nach w vorkommen.)

Es ist also gleichgültig, ob man in der Iteration die alten Werte $S(v,k,k-1)$ und $S(k,w,k-1)$ oder die neuen Werte $S(v,k,k)$ und $S(k,w,k)$ benutzt. Man braucht also gar kein „altes“ und „neues“ Array, und erhält mit einem zweidimensionalen Array `S[1..n,1..n]`:

```
S[v,v] ← 0, für 1 ≤ v ≤ n;
S[v,w] ← c(v,w), für 1 ≤ v,w ≤ n, v ≠ w;
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w] ← min{S[v,w], S[v,k] + S[k,w]}.
```

Satz 12.1.1

Der bislang dargestellte Algorithmus berechnet in Zeit $O(n^3)$ und Platz $O(n^2)$ die Längen der kürzesten Wege zwischen allen Paaren von Knoten des gewichteten Digraphen $G = (V, E, c)$.

Wir wollen aber nicht nur die **Kosten** eines kürzesten Weges wissen, sondern auch zu gegebenen v, w einen **kürzesten Weg konstruieren** können.

Hierfür: Zusätzliches Array `I[1..n,1..n]`, in dem durch die Runden $k = 0, 1, \ldots, n$ hindurch für jedes Paar (v, w) die folgende Information gehalten wird:
Was ist der Knoten mit der größten Nummer, der für die Konstruktion eines kürzesten k-Weges von v nach w verwendet worden ist?

In Runde k: Wenn sich `S[`v, w`]` nicht ändert, bleibt `I[`v, w`]` gleich.
Wenn $S(v, w, k) < S(v, w, k-1)$, wird `I[`$v, w$`]` auf k gesetzt, weil Knoten k für einen kürzesten k-Weg notwendig ist.

Diese Erweiterung liefert den „vollen“ Floyd-Warshall-Algorithmus.

12.1.2 Algorithmus Floyd-Warshall(C[1..n,1..n])

Eingabe: C[1..n,1..n]: Matrix der Kantenkosten/-längen $c(v,w)$ in $\mathbb{R} \cup \{+\infty\}$

Ausgabe: S[1..n,1..n]: Kosten $S(v,w)$ eines kürzesten v-w-Weges

I[1..n,1..n]: minimaler maximaler Knoten auf einem kürzesten v-w-Weg

```
(1)   for v from 1 to n do
(2)     for w from 1 to n do
(3)       if v = w then S[v,v] ← 0; I[v,v] ← 0
(4)         else S[v,w] ← C[v,w];
(5)              if S[v,w] < ∞ then I[v,w] ← 0 else I[v,w] ← −1;
(6)   for k from 1 to n do
(7)     for v from 1 to n do
(8)       for w from 1 to n do
(9)         if S[v,k] + S[k,w] < S[v,w] then
(10)          S[v,w] ← S[v,k] + S[k,w]; I[v,w] ← k;
(11)  Ausgabe: S[1..n,1..n] und I[1..n,1..n].
```

Korrektheit: Eigentlich „nach Konstruktion“. Formal beweist man die folgende Schleifeninvariante:

Nach dem Schleifendurchlauf mit k in `k` ist $\texttt{S}[v, w] = S(v, w, k)$.

(Durch vollständige Induktion über $k = 0, \ldots, n$, unter Ausnutzung der Bellman-Gleichungen und Eigenschaft $(*)$, der Abwesenheit von strikt negativen Kreisen.)

Zudem, wenn auch die „roten“ Teile für $\texttt{I}[1..n, 1..n]$ ausgeführt werden:

$\texttt{I}[v, w]$ ist das kleinste ℓ mit folgender Eigenschaft:
Unter den kürzesten Wegen von v nach w mit Zwischenknoten in $1, \ldots, k$ gibt es einen, dessen **maximaler** Eintrag ℓ ist.

Spezialfälle: $\texttt{I}[v, v] = 0$ für alle v. Wenn $v \neq w$ und die Kante (v, w) ein kürzester Weg ist, dann ist $\texttt{I}[v, w] = 0$; wenn $v \not\rightsquigarrow w$, ist $\texttt{I}[v, w] = -1$.

(Beweis durch Induktion über $k = 0, \ldots, n$.)

Ein kürzester Weg von v nach w kann dann ausgehend von $\texttt{I}[1..n, 1..n]$ mit einer einfachen rekursiven Prozedur „`printPathInner`“ gefunden werden.

12.1.2a Algorithmus printPathInner(v, w)
Global: `I[1..n,1..n]`: Matrix der „minimalen maximalen“ inneren Knoten
Eingabe: $v, w \in V$, $v \neq w$
Ausgabe: drucke Folge der „inneren Knoten“ eines kürzesten v-w-Weges, falls $v \rightsquigarrow w$.

```
(1)     k ← I[v,w];
(2)     if k > 0 then
(3)       printPathInner(v, k); print(k); printPathInner(k, w);
```

12.1.2b Algorithmus printPath(v, w)
Global: `I[1..n,1..n]`: Matrix der „minimalen maximalen“ inneren Knoten
Eingabe: $v, w \in V$
Ausgabe: drucke kürzesten v-w-Weg, falls $v \rightsquigarrow w$, sonst gib „kein Weg“ aus

```
(1)   if v = w
(2)     then print(v, „Länge 0“)
(3)     elseif I[v,w] < 0 then print(„kein Weg“)
(4)     else print(v); printPathInner(v, w); print(w);
```

Übung: (a) Man führe dies an einem Beispiel durch. (b) Man beweise, dass die Ausgabe korrekt ist.

12.1.3 Satz

Der Algorithmus von Floyd-Warshall löst das All-Pairs-Shortest-Paths-Problem in Rechenzeit $O(n^3)$ und Speicherplatz $O(n^2)$.

Das Ergebnis ist eine Datenstruktur der Größe $O(n^2)$, mit der sich zum Argument (v, w) mit Algorithmus **printPath** ein kürzester Weg von v nach w in Zeit $O(\#(\text{Kanten auf dem Weg}))$ ausgeben lässt.

Bemerkung: Wenn man den Algorithmus auf einer Eingabe ausführt, die Bedingung $(*)$ („es gibt keine negativen Kreise“) nicht erfüllt, dann wird am Ende mindestens einer der Werte $\mathtt{S}[v, v]$ negativ sein. Dies kann man als Kriterium für die Existenz negativer Kreise benutzen. Achtung: Wir werden in 12.2 eine andere Methode kennenlernen, um dies festzustellen.

Rekapitulation: Dynamische Programmierung (DP)

Algorithmenparadigma für **Optimierungsprobleme**. Typische Schritte:

- Gegeben eine Instanz, definiere (viele) **„Teilprobleme“**.
- Identifiziere einfache **Basisfälle** („kleine“, „leichte“ Teilprobleme).
- Formuliere Version der Eigenschaft

  Substrukturen optimaler Lösungen für Teilprobleme sind optimale Lösungen für (kleinere) Teilprobleme.

- Finde Rekursionsgleichungen für **Werte** optimaler Lösungen:

  Bellmansche Optimalitätsgleichungen

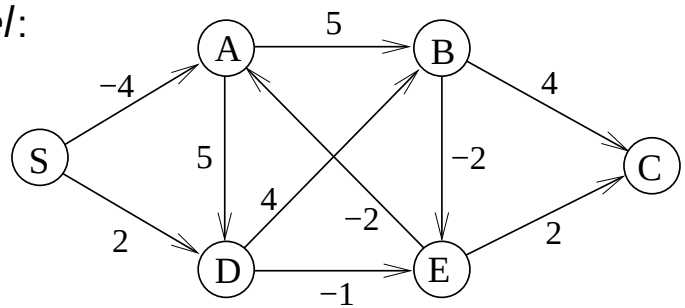
- Berechne optimale Werte **iterativ**. (Strukturen eventuell in separatem Durchgang.)

12.2 Der Bellman–Ford-Algorithmus

Zweck: Kürzeste Wege **von einem Startknoten** s aus
in gewichtetem Digraphen (V, E, c). (**SSSP**: „**S**ingle **S**ource **S**hortest **P**aths“).

Anders als beim Algorithmus von Dijkstra sind **negative Kantenkosten** zugelassen.

Beispiel:



$$c((\mathrm{S},\mathrm{A},\mathrm{B},\mathrm{C})) = 5,$$
$$c((\mathrm{S},\mathrm{D},\mathrm{E},\mathrm{C})) = 3,$$
$$c((\mathrm{S},\mathrm{A},\mathrm{D},\mathrm{E},\mathrm{C})) = 2,$$
$$c((\mathrm{S},\mathrm{A},\mathrm{B},\mathrm{E},\mathrm{C})) = 1 = d(\mathrm{S},\mathrm{C}).$$

Wir sagen, dass von $v_0 = s$ aus **ein Kreis mit negativer Gesamtlänge erreichbar ist**, wenn es einen Kantenzug $(v_0, \ldots, v_q, \ldots, v_r)$ gibt mit $q < r$ und $v_q = v_r$, wobei $c(v_q, v_{q+1}) + \ldots + c(v_{r-1}, v_r) < 0$ gilt. Wenn dies der Fall ist, kann man durch wiederholtes Durchlaufen des Kreises $(v_q, \ldots, v_r)$ Kantenzüge mit beliebig stark negativen Kosten von s nach v_r erhalten – die Frage nach einem „kürzesten Weg“ wird sinnlos.
In diesem Fall soll unser Algorithmus „negativer Kreis“ ausgeben.

Knotenmenge: $V = \{1, \ldots, n\}$, Startknoten $s \in V$,
Menge E von gerichteten Kanten, mit Kantengewichten $c\colon E \to \mathbb{R}$.

Das Format des Inputs ist recht frei:
Adjazenzlisten oder auch nur eine (unstrukturierte) Liste aller Kanten mit Gewichten.

Für die Überlegungen nehmen wir zuerst an, dass von s aus kein Kreis negativer Länge erreichbar ist.

Datenstrukturen: Distanzarray `dist`$[1..n]$
Vorgängerarray `p`$[1..n]$

Idee: `dist`$[v]$ enthält stets einen **Schätzwert** (eine obere Schranke) für die Länge eines kürzesten Weges von s nach v. Wert ∞ ist zugelassen, ist Startwert.

`p`$[v]$ enthält den direkten Vorgänger u von v auf einem Weg von s nach v der Länge $\leq$ `dist`$[v]$, wenn dies $< \infty$ ist. Dabei bilden die Kanten $(\texttt{p}[v], v)$ stets einen Baum, am Ende einen **Baum von kürzesten Wegen**.

Durch eine Reihe von Operationen sollen Werte `dist`$[v]$ und `p`$[v]$ sukzessive so gesetzt werden, dass sie kürzesten Wegen entsprechen.

Grundoperation, schon aus dem Dijkstra-Algorithmus bekannt:

update(u, v) // für $(u, v) \in E$

(1) **if** `dist[`u`]` $+ c(u, v) <$ `dist[`v`]` **then**

(2) `p[`v`]` $\leftarrow u$;

(3) `dist[`v`]` $\leftarrow$ `dist[`u`]` $+ c(u, v)$;

Folgende Aussage benutzen wir als Invariante:

$(*)$ Für alle $w \in V$ gilt: $d(s, w) \leq$ `dist[`w`]`.
D. h.: Wenn `dist[`w`]` $< \infty$, dann gibt es einen Weg von s nach w der Länge $\leq$ `dist[`w`]`.

Lemma 12.2.1

Wenn $(*)$ gilt und **update**(u, v) ausgeführt wird, dann gilt $(*)$ weiter.

(„Mit **update**(u, v) macht man nichts falsch.“)

Lemma 12.2.1

Wenn $(*)$ gilt und **update**(u, v) ausgeführt wird, dann gilt $(*)$ weiter.

Beweis: Sei $w \in V$, sei $(u, v) \in E$.

Wenn sich bei **update**(u, v) der Wert `dist[`w`]` nicht ändert, ist nichts zu zeigen.

Damit bleibt nur der Fall $v = w$ und `dist[`v`]` $< \infty$ (nach der **update**-Operation).

Dann muss `dist[`u`]` $< \infty$ gelten. Wir wissen nach Voraussetzung, dass $d(s, u) \leq$ `dist[`u`]` gilt, es also einen Weg p der Länge $\leq$ `dist[`u`]` von s nach u gibt.

Anhängen der Kante (u, v) an p liefert einen Kantenzug der Länge $\leq$ `dist[`u`]` $+ c(u, v)$ von s nach v.

Nach unserer vorläufigen Annahme kann dieser Kantenzug keinen negativen Kreis enthalten.

Also gibt es einen Weg von s nach v der Länge $\leq$ `dist[`u`]` $+ c(u, v)$. □

Algorithmischer Ansatz: Wir beginnen mit $\texttt{dist}[s] \leftarrow 0$ und $\texttt{dist}[v] \leftarrow \infty$ für alle anderen $v \in V$.

Dann gilt $(*)$.

Nun führen wir viele **update**(u, v)-Operationen aus (auch mehrfach an derselben Kante).

Lemma 12.2.2

$(s = v_0, v_1, \ldots, v_t = w)$ sei ein kürzester Weg von $s = v_0$ nach $w = v_t$ (also ohne Knotenwiederholung). Am Anfang gelte $\texttt{dist}[s] = 0$. In der Folge der ausgeführten **update**(u, v)-Operationen sollen

$$\textbf{update}(v_0, v_1), \ldots, \textbf{update}(v_{t-1}, v_t)$$

in dieser Reihenfolge vorkommen.

Dann gilt am Ende: $\texttt{dist}[w] = d(s, w)$.

Beweis: Durch Induktion über $i = 0, \ldots, t$ zeigen wir:
Nach Ausführung von **update**$(v_0, v_1), \ldots,$ **update**(v_{i-1}, v_i) gilt $\texttt{dist}[v_i] = d(s, v_i)$.

I.A.: $i = 0$. Es ist $v_0 = s$ und nach der Initialisierung gilt $\texttt{dist}[s] = 0 = d(s, s)$.

I.V.: Aussage ist richtig für $i - 1$.

I.S.: Nach I.V. gilt nach Ausführung von **update**$(v_0, v_1), \ldots,$ **update**(v_{i-2}, v_{i-1}) die Gleichung $\texttt{dist}[v_{i-1}] = d(s, v_{i-1})$.

Nun werden weitere **update**-Operationen und dann **update**(v_{i-1}, v_i) ausgeführt. Wegen Lemma 12.2.1 kann sich dabei $\texttt{dist}[v_{i-1}]$ nicht mehr ändern (verringern).

Nach der Operation **update**(v_{i-1}, v_i) gilt (wegen der Struktur der **update**-Operation):

$$\texttt{dist}[v_i] \leq \texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i) \overset{\text{I.V.}}{=} d(s, v_{i-1}) + c(v_{i-1}, v_i) \overset{(+)}{=} d(s, v_i).$$

(Für $(+)$ wird benutzt, dass $(s = v_0, v_1, \ldots, v_\ell)$ ein kürzester Weg von s nach v_ℓ ist, für $\ell = i - 1$ und $\ell = i$. Dies muss so sein, weil $(v_0, \ldots, v_\ell)$ ein Teil eines kürzesten Weges von s nach w ist.)
Mit Lemma 12.2.1 folgt $\texttt{dist}[v_i] = d(s, v_i)$. □

12.2.3 Algorithmus Bellman-Ford$(((V, E, c), s))$

Eingabe: (V, E, c): Digraph mit Kantengewichten in $\mathbb{R}$, $s \in V$: Startknoten

Ausgabe: Wenn G keine Kreise mit negativem Gewicht hat:
In `dist[`v`]`: Abstand $d(s, v)$;
In `p[`$1..n$`]`: Baum von kürzesten Wegen (wie bei Dijkstra);

Initialisierung:

(1) **for** `v` **from** 1 **to** n **do**
(2) `dist[v]` $\leftarrow \infty$; `p[v]` $\leftarrow -1$;
(3) `dist[`s`]` $\leftarrow 0$; `p[`s`]` $\leftarrow -2$; // wird nie geändert, wenn kein negativer Kreis existiert

Hauptschleife:

(4) **for** `i` **from** 1 **to** $n-1$ **do** // heißt nur: wiederhole $(n-1)$-mal
(5) **forall** $(u, v) \in E$ **do update**(u, v); // beliebige Reihenfolge

Zyklentest:

(6) **forall** $(u, v) \in E$ **do**
(7) **if** `dist[`u`]` $+ c(u, v) <$ `dist[`v`]` **then return** „negativer Kreis“;

Ergebnis:

(8) **return** `dist[`$1..n$`]`, `p[`$1..n$`]`.

Satz 12.2.4

Der Bellman-Ford-Algorithmus hat folgendes Verhalten:

(a) Wenn es keinen von s aus erreichbaren Kreis mit negativer Gesamtlänge gibt („negativer Kreis“), steht am Ende des Algorithmus in $\texttt{dist}[v]$ die Länge eines kürzesten Weges von s nach v, und die Kanten $(\texttt{p}[w], w)$ (mit $\texttt{p}[w] > 0$) bilden einen Baum der kürzesten Wege von s aus zu den erreichbaren Knoten.

(b) Der Algorithmus gibt „negativer Kreis“ aus genau dann wenn es einen von s aus erreichbaren negativen Kreis gibt.

(c) Die Rechenzeit ist $O(nm)$.

Beweis: (c) ist klar.

(a) Da es keine negativen Kreise gibt, die von s aus erreichbar sind, kann durch Herausschneiden von Kreisen aus Kantenzügen, die in s starten, die Länge nicht steigen.

Wie in 12.1 folgert man, dass es für jeden von s aus erreichbaren Knoten v einen kürzesten **Weg** von s nach v gibt, auf dem sich also keine Knoten wiederholen. Dieser Weg hat höchstens $n-1$ Kanten.
Damit ist auch das Konzept „Länge $d(s,v)$ eines kürzesten Weges von s nach v“ wohldefiniert.

Sei $v \in V$ mit $d(s,v) < \infty$ beliebig. Wähle einen kürzesten Weg $(s = v_0, \ldots, v_t = v)$ von s nach v. Dann ist $t \leq n-1$.
Wir können Lemma 12.2.2 auf den Ablauf des Algorithmus anwenden, da die Initialisierung ist wie dort vorgeschrieben und auf die Kanten $(v_0, v_1), \ldots, (v_{t-1}, v_t)$ nacheinander die **update**-Operation angewendet wird.
Also gilt am Ende $\texttt{dist}[v] = d(s,v)$.
Der Beweis der Behauptung über die $(\texttt{p}[v], v)$-Kanten ist nicht ganz einfach.
Er ist **nicht prüfungsrelevant**, aber für Interessierte auf den Druckfolien dargestellt.

Behauptung: Die p-Werte beschreiben einen Kürzeste-Wege-Baum.

(i) Die Kanten $(\texttt{p}[w], w)$ mit $\texttt{p}[w] \geq 1$ bilden zu jedem Zeitpunkt einen **Baum** T mit Wurzel s, der alle Knoten w mit $0 \leq \texttt{dist}[w] < \infty$ enthält.

Dies wird durch Induktion über update-Operationen gezeigt.

Am Anfang ist es wahr – T hat nur den Knoten s, und es gibt keine Kante $(\texttt{p}[w], w)$ mit $0 \leq \texttt{dist}[w] < \infty$.

Bei einer $\texttt{update}(u, v)$-Operation, die $\texttt{p}[v]$ verändert, gibt es zwei Möglichkeiten. Entweder wird Knoten v als neues Blatt an T angehängt (nämlich wenn vorher $\texttt{p}[v] = -1$ war) oder der Unterbaum mit Wurzel v wird von seinem bisherigen Vorgänger abgehängt und als Unterbaum an u angehängt.

Wichtig: Wir müssen uns vergewissern, dass es nicht möglich ist, dass in dieser Situation u selbst im Unterbaum unter v sitzt. (Sonst würden wir einen Kreis erzeugen, und die $(\texttt{p}[w], w)$-Kanten wären kein Baum mehr.) Annahme: $(v = v_0, v_1, \ldots, v_r = u)$ ist ein Weg mit $\texttt{p}[v_i] = v_{i-1}$ für $i = 1, \ldots, r$, es wird $\texttt{update}(u, v)$ ausgeführt, wobei sich $\texttt{dist}[u] + c(u, v) < \texttt{dist}[v]$ ergibt. Wir haben $\texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i) \leq \texttt{dist}[v_i]$ für $i = 1, \ldots, r$. (Das gilt, weil zum Zeitpunkt, wo $\texttt{p}[v_i] = v_{i-1}$ gesetzt wurde, Gleichheit herrscht und sich $\texttt{dist}[v_i]$ nachher nicht mehr geändert hat.)

Die Ungleichung für u und v kann man auch als $\texttt{dist}[v_r] + c(v_r, v_0) < \texttt{dist}[v_0]$ lesen. Wenn wir nun diese $r+1$ Ungleichungen addieren, dann fallen sämtliche $\texttt{dist}[v_i]$-Werte heraus, und es bleibt $c(v_0, v_1) + \cdots + c(v_{r-1}, v_r) + c(v_r, v_0) < 0$ stehen. Das heißt aber, dass $(v = v_0, v_1, \ldots, v_r, v_0)$ ein von s aus erreichbarer negativer Kreis ist, im Widerspruch zu unserer Annahme.

(ii) Wenn am Ende $p = (s = v_0, v_1, v_2, \ldots, v_t = v)$ ein Weg im Baum T ist, dann gilt $d(s, v) = \texttt{dist}[v] = \sum_{1 \le i \le t} c(v_{i-1}, v_i)$. D. h.: p ist ein kürzester Weg von s nach v.
(In dem Moment, in dem $\texttt{dist}[v_i]$ in der Operation $\texttt{update}(v_{i-1}, v_i)$ seinen endgültigen Wert erhält und $\texttt{p}[v_i]$ auf v_{i-1} gesetzt wird, gilt $\texttt{dist}[v_i] = \texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i)$. Nach diesem Zeitpunkt kann sich nur noch $\texttt{dist}[v_{i-1}]$ verringern, also gilt am Ende $\texttt{dist}[v_i] \ge \texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i)$.)
Addition dieser Ungleichungen, für $1 \le i \le t$, ergibt $d(s, v) = \texttt{dist}[v] \ge \sum_{1 \le i \le t} c(v_{i-1}, v_i)$.
Das bedeutet, dass p ein Weg der Länge $d(s, v)$ ist, also optimal ist. □

Beweis von (b): „$\Rightarrow$“: Indirekt. Annahme: Von s aus ist kein negativer Kreis erreichbar (also ist $d(s, v)$ definiert für alle v), aber in Zeile (7) wird ein v mit $\texttt{dist}[u] + c(u, v) < \texttt{dist}[v]$ gefunden. – Nach (a) folgt $d(s, u) + c(u, v) < d(s, v)$, was offensichtlich unmöglich ist.

„$\Leftarrow$“: Sei $(s = v_0, \ldots, v_r, \ldots, v_t)$ mit $r < t$ und $v_r = v_t$ ein Kantenzug, der mit einem Kreis endet. Wenn der Test in Zeile (7) stets „$\geq$“ ergibt, gilt:

$$\begin{aligned}
\texttt{dist}[v_r] + c(v_r, v_{r+1}) &\geq \texttt{dist}[v_{r+1}] \\
\texttt{dist}[v_{r+1}] + c(v_{r+1}, v_{r+2}) &\geq \texttt{dist}[v_{r+2}] \\
&\vdots \\
\texttt{dist}[v_{t-1}] + c(v_{t-1}, v_t) &\geq \texttt{dist}[\underbrace{v_t}_{=v_r}]
\end{aligned}$$

Addition dieser Ungleichungen und Vereinfachen liefert:

$$c(v_r, v_{r+1}) + \cdots + c(v_{t-1}, v_t) \geq 0.$$

Das bedeutet: Jeder von s aus erreichbare Kreis hat nichtnegative Kosten. □

Bemerkung: Die Entwicklung des Bellman-Ford-Algorithmus folgt nur scheinbar nicht dem DP-Paradigma (Teilprobleme, Basisfälle, Bellmansche Gleichungen). Dies sieht man wie folgt:

Wir definieren, für $v \in V$ und $0 \leq k < n$:

$$S(v,k) := d_k(s,v) := \min(\text{Länge eines } s\text{-}v\text{-Weges mit} \leq k \text{ Kanten}).$$

Damit: Basisfälle $S(s,0) = 0$ und $S(v,0) = \infty$ für $v \neq s$.

Bellmansche Optimalitätsgleichungen:
Sei $k \geq 1$. Für einen kürzesten s-v-Weg p mit $\leq k$ Kanten gibt es zwei Fälle:

- p hat $\leq k-1$ Kanten. Dann hat er Länge $S(v,k-1)$.
- p hat k Kanten. Sei w der vorletzte Knoten auf p. Dann hat p Länge $S(w,k-1) + c(w,v)$.

Da w nicht bekannt ist, muss man über alle Möglichkeiten minimieren, und man erhält:

$$S(v,k) = \min(\{S(v,k-1)\} \cup \{S(w,k-1) + c(w,v) \mid (w,v) \in E\}).$$

Dies führt zur folgenden „DP-Version“ des Algorithmus von Bellman und Ford.

12.2.3 Algorithmus Bellman-Ford (DP-Version) $(((V, E, c), s))$

Eingabe, Ausgabe: wie vorher

Datenstruktur: `dist`$[1..n, 0..n-1]$: `real`; `p`$[1..n]$: `integer`;

Initialisierung:

(1) **for** v **from** 1 **to** n **do**

(2) `dist[v,0]` $\leftarrow \infty$; `p[v]` $\leftarrow -1$;

(3) `dist`$[s, 0] \leftarrow 0$; `p`$[s] \leftarrow -2$;

Hauptschleife:

(4) **for** k **from** 1 **to** $n-1$ **do**

(5) **forall** v $\in V$ **do**

(6) `dist[v,k]` $\leftarrow$ `dist[v,k-1]`

(7) **forall** $(\mathtt{w}, \mathtt{v}) \in E$ **do**

(8) **if** `dist[w,k-1]` $+ c(\mathtt{w}, \mathtt{v}) <$ `dist[v,k]` **then**

(9) „aktualisiere `dist[v,k]` und `p[v]`“.

Ergebnis:

(10) **return** `dist`$[1..n, n-1]$, `p`$[1..n]$.

Aktualisierungen (4)–(9) wie im gewöhnlichen Bellman-Ford-Algorithmus, aber in spezieller Reihenfolge und „schichtenweise“ über $k = 1, \ldots, n-1$. Zwei Ebenen des `dist`-Arrays genügen.

Alternativer Ansatz (zu Floyd-Warshall) zur Lösung des APSP-Problems in Digraphen ohne negative Kreise:

(1) Löse **eine** **SSSP-Instanz** in G mit neuem Knoten s per **Bellman-Ford-Algorithmus**. – Kosten: $O(mn)$.

(2) Definiere modifizierte Kantengewichte $\bar{c}(v,w) \geq 0$, $(v,w) \in E$.

(3) Löse n **viele SSSP-Instanzen** in G mit $\bar{c}$, mittels **Dijkstra-Algorithmus**! Kosten (mit Binärheaps): $n \cdot O(m \log n) = O(mn \log n)$.

Gesamtkosten: $O(mn \log n)$.

Viel besser als das $O(n^3)$ des Floyd-Warshall-Algorithmus, wenn $m \ll n^2 / \log n$.

Details:

Zu (1): Füge zu $G = (V, E, c)$ einen **neuen Knoten** s und alle Kanten (s, v), $v \in V$, hinzu und setze $c(s, v) := 0$, für $v \in V$. Dies ergibt erweiterten Digraphen $G' = (V \cup \{s\}, E', c')$.

Anwenden des Algorithmus von Bellman und Ford auf G' liefert $\pi(v) := d_{G'}(s, v)$, für $v \in V$ (und prüft, ob es in G negative Kreise gibt).

Zu (2): Für $(v, w) \in E$ definiere:

$$\bar{c}(v, w) := c(v, w) + \pi(v) - \pi(w).$$

(Man sagt: Modifiziere c mittels „Knotenpotenzialen“ π.)

Für eine Kante $(v, w) \in E$ beobachten wir:

$\pi(w) = d_{G'}(s, w) \leq d_{G'}(s, v) + c(v, w) = \pi(v) + c(v, w)$, also $\bar{c}(v, w) \geq 0$.

Behauptung: Wenn $p = (v = v_0, v_1, \ldots, v_t = w)$ ein beliebiger v-w-Weg in G ist, dann gilt für die „c-Länge“ $c(p)$ und die „$\bar{c}$-Länge“ $\bar{c}(p)$:

$$\bar{c}(p) = c(p) + \pi(v) - \pi(w).$$

Beweis: Mit Teleskopsumme:

$$\begin{aligned}\bar{c}(p) = \sum_{1 \le i \le t} \bar{c}(v_{i-1}, v_i) &= \sum_{1 \le i \le t} (c(v_{i-1}, v_i) + \pi(v_{i-1}) - \pi(v_i)) \\ &= \sum_{1 \le i \le t} c(v_{i-1}, v_i) + \pi(v_0) - \pi(v_t) = c(p) + \pi(v_0) - \pi(v_t).\end{aligned}$$

D. h.: Für v-w-Wege p und p' gilt $c(p) \le c(p')$ genau dann wenn $\bar{c}(p) \le \bar{c}(p')$.

Insbesondere: Kürzeste Wege in G bezüglich c sind dieselben wie kürzeste Wege bezüglich $\bar{c}$.

Zu (3): Für jeden Knoten $v \in V$ gilt: Ein Aufruf des Dijkstra-Algorithmus liefert alle Distanzen $\bar{d}(v, w)$ zwischen Knoten in G mit $\bar{c}$ und einen Kürzeste-Wege-Baum für Startknoten v.

c-Distanzen: $d_G(v, w) = \bar{d}(v, w) - \pi(v) + \pi(w)$, für $(v, w) \in V \times V$.

Der Kürzeste-Wege-Baum mit Startknoten v für c ist derselbe wie für $\bar{c}$.

Diese Konstruktion wird mit jedem $v \in V$ als Startknoten ausgeführt.

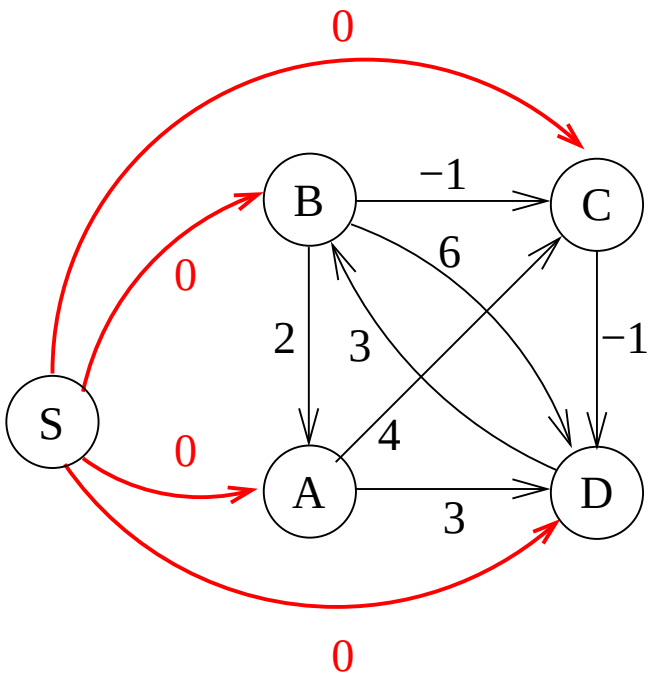
Dies liefert alle Distanzen $d_G(v, w)$ und n viele Kürzeste-Wege-Bäume T_v, $v \in V$.

Rechenzeit: $O(nm \log n)$.

Um einen kürzesten Weg von v nach w zu finden, benutze T_v.

Rechenzeit: $O(\#(\text{Kanten auf einem kürzesten } v\text{-}w\text{-Weg}))$.

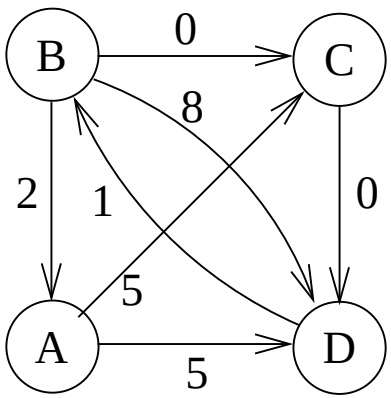
Beispiel:



(1) Bellman-Ford angewendet auf den um S erweiterten Graphen liefert:

$\pi(\mathrm{A}) = 0,\ \pi(\mathrm{B}) = 0,\ \pi(\mathrm{C}) = -1,\ \pi(\mathrm{D}) = -2.$

Auf den Graphen $\bar{G} = (V, E, \bar{c})$ mit Kantenlängen $\bar{c}(v, w)$
(und $\pi(\mathrm{A}) = 0$, $\pi(\mathrm{B}) = 0$, $\pi(\mathrm{C}) = -1$, $\pi(\mathrm{D}) = -2$)



wendet man viermal den Algorithmus von Dijkstra an, mit Startknoten A, B, C und D. – Einige Ergebnisse:

$\bar{d}(\mathrm{B}, \mathrm{C}) = 0$, $\bar{d}(\mathrm{B}, \mathrm{D}) = 0$, $\bar{d}(\mathrm{B}, \mathrm{A}) = 2$, $\bar{d}(\mathrm{C}, \mathrm{A}) = 3$.

Durch Korrektur $d(v, w) = \bar{d}(v, w) - \pi(v) + \pi(w)$ erhält man:

$d(\mathrm{B}, \mathrm{C}) = 0-0+(-1) = -1$, $d(\mathrm{B}, \mathrm{D}) = 0-0+(-2) = -2$, $d(\mathrm{B}, \mathrm{A}) = 2-0+0 = 2$, $d(\mathrm{C}, \mathrm{A}) = 3 - (-1) + 0 = 4$.

12.3 Editierdistanz

Problemstellung: Sei A ein Alphabet. (*Bsp.*: Lat. Alphabet, ASCII-Alphabet, {A,C,G,T}.)

Wenn $x = a_1 \dots a_m \in A^*$ und $y = b_1 \dots b_n \in A^*$ zwei Zeichenreihen über A sind, möchte man herausfinden, wie **ähnlich** (oder unähnlich) sie sind.

„`Arbeit`“ und „`Freizeit`“ sind nicht identisch, aber intuitiv einander ähnlicher als „`Informatik`“ und „`Camping`“. Wie können wir „Ähnlichkeit“ messen?

Wir definieren Elementarschritte (Editier-Operationen), die einen String verändern:

- Lösche einen Buchstaben aus einem String: Aus uav wird uv $(u, v \in A^*, a \in A)$.
  Beispiel: `Haut` $\rightarrow$ `Hut`.
- Füge einen Buchstaben in einen String ein: Aus uv wird uav $(u, v \in A^*, a \in A)$.
  Beispiel: `Hut` $\rightarrow$ `Haut`.
- Ersetze einen Buchstaben: Aus uav wird ubv $(u, v \in A^*, a, b \in A)$.
  Beispiel: `Haut` $\rightarrow$ `Hast`.

Der „Abstand“ oder die **Editierdistanz $d(x, y)$** von x und y ist die minimale Anzahl von Editieroperationen, die benötigt werden, um x in y zu transformieren.

Offenbar: $d(x, y) = d(y, x)$. (Einfügen/Löschen invers zueinander, Ersetzen invers zu Ersetzen.)

Überlege: Äquivalenz Transformation/Zuordnung:

A	r	b	e	i	-	-	-	t		
F	r	-	e	i	z	e	i	t		
I	n	f	o	r	m	a	t	-	i	k	-
C	-	-	-	-	-	a	m	p	i	n	g

Man schreibt Strings aus Buchstaben und dem Sonderzeichen - untereinander, wobei die beiden Zeilen jeweils das Wort x bzw. y ergeben, wenn man die -'s ignoriert.
Die **Kosten** einer solchen Anordnung: Die Anzahl der Positionen, an denen die oberen und unteren Einträge nicht übereinstimmen.

Die Kosten, die eine solche Anordnung erzeugt, sind gleich der Anzahl der Editierschritte in einer Transformation. (Im Beispiel: 5 bzw. 10.) Daher: Die **minimalen** Kosten einer solchen Anordnung sind die Editierdistanz. (Im Beispiel: 4 (!) bzw. 10.)

Das Problem „Editierdistanz“ wird folgendermaßen beschrieben:

Input: Strings $x = a_1 \dots a_m$, $y = b_1 \dots b_n$ aus A^*.

Aufgabe: Berechne $d(x, y)$ (und eine Editierfolge, die x in y transformiert).

Ansatz: Dynamische Programmierung

Unser Beispiel: $x = \texttt{Exponentiell}$ und $y = \texttt{Polynomiell}$.

1. Schritt: Relevante **Teilprobleme** identifizieren!

Betrachte Präfixe $x[1..i] = a_1 \dots a_i$ und $y[1..j] = b_1 \dots b_j$, und setze

$$\boxed{\boldsymbol{E(i,j)} := d(x[1..i], y[1..j]), \text{ für } 0 \leq i \leq m,\ 0 \leq j \leq n}$$

Im Beispiel bedeutet $\boldsymbol{E(7,6)}$: Berechne $d(\texttt{Exponen}, \texttt{Polyno})$
(wegen $x[1..7] = \texttt{Exponen}$ und $y[1..6] = \texttt{Polyno}$).

Um Bellmansche Gleichungen zu erhalten, betrachten wir Beziehungen zwischen Teilproblemen. Dabei benutzen wir die Vorstellung der Anordnungen von Wörtern untereinander wie auf Folie 38.

Betrachte nichtleere Präfixe $x[1..i] = a_1 \dots a_i$ und $y[1..j] = b_1 \dots b_j$, mit $1 \leq i \leq m$, $1 \leq j \leq n$. Wenn $x[1..i]$ und $y[1..j]$ optimal untereinander angeordnet sind, mit Kosten $E(i,j)$, gibt es drei Möglichkeiten für die Behandlung der letzten Stelle:

1. Fall (Ersetzung)	2. Fall (Löschung)	3. Fall (Einfügung)
$x[1..i-1]$ a_i	$x[1..i-1]$ a_i	$x[1..i]$ –
$y[1..j-1]$ b_j	$y[1..j]$ –	$y[1..j-1]$ b_j

1. Fall (Ersetzung)	2. Fall (Löschung)	3. Fall (Einfügung)
$x[1..i-1]$ a_i	$x[1..i-1]$ a_i	$x[1..i]$ -
$y[1..j-1]$ b_j	$y[1..j]$ -	$y[1..j-1]$ b_j

1. Fall: Die letzte Stelle kostet $\text{diff}(a_i, b_j) := [a_i \neq b_j]$ $\left(= \begin{cases} 1 & \text{falls } a_i \neq b_j \\ 0 & \text{falls } a_i = b_j \end{cases}\right)$

$x[1..i-1]$ und $y[1..j-1]$ sind einander optimal zugeordnet („optimale Substruktur“).

$$\Rightarrow \quad E(i,j) = E(i-1, j-1) + \text{diff}(a_i, b_j).$$

2. Fall: In der letzten Stelle erfolgt eine Löschung, mit Kosten 1.

$x[1..i-1]$ und $y[1..j]$ sind einander optimal zugeordnet.

$$\Rightarrow \quad E(i,j) = E(i-1, j) + 1.$$

3. Fall: In der letzten Stelle erfolgt eine Einfügung, mit Kosten 1.

$x[1..i]$ und $y[1..j-1]$ sind einander optimal zugeordnet.

$$\Rightarrow \quad E(i,j) = E(i, j-1) + 1.$$

Unter den drei möglichen Fällen ist derjenige (sind diejenigen) mit den kleinsten Kosten relevant. Wir erhalten zusammengefasst:

Bellmansche Optimalitätsgleichungen für Editierdistanz:

$$E(i,j) = \min\{E(i-1,j-1) + \mathsf{diff}(a_i, b_j), E(i-1,j) + 1, E(i,j-1) + 1\}.$$

Basisfälle: Leere Präfixe $\varepsilon = x[1..0]$ und $\varepsilon = y[1..0]$.

Klar: $E(i,0) = d(x[1..i], \varepsilon) = i$, für $0 \leq i \leq m$, und
$E(0,j) = d(\varepsilon, y[1..j]) = j$, für $0 \leq j \leq n$.

(Man muss alle Buchstaben streichen bzw. einfügen.)

Die Zahlen $E(i,j)$ berechnen wir iterativ, indem wir sie in eine Matrix $\texttt{E}[0..m,0..n]$ eintragen. Dies liefert den DP-Algorithmus für die Editierdistanz.

Initialisierung:

$\texttt{E}[i,0] \leftarrow i$, für $i = 0, \ldots, m$;
$\texttt{E}[0,j] \leftarrow j$, für $j = 0, \ldots, n$.

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellmanschen Optimalitätsgleichungen:

for $\texttt{i}$ **from** 1 **to** m **do**
    for $\texttt{j}$ **from** 1 **to** n **do**
        $\texttt{E[i,j]} \leftarrow \min\{\texttt{E[i-1,j-1]} + \text{diff}(a_{\texttt{i}}, b_{\texttt{j}}), \texttt{E[i-1,j]} + 1, \texttt{E[i,j-1]} + 1\}$;
return $\texttt{E}[m,n]$.

Rechenzeit: $O(m \cdot n)$

Bemerkung: Andere Reihenfolgen für das Ausfüllen sind möglich. Für die Ausführung von Hand ist es geschickt, zuerst die Werte (i,j) mit $\text{diff}(a_i, b_j) = 0$, also $a_i = b_j$ zu identifizieren (rot, unterstrichen).

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1											
x	2											
p	3											
o	4		–				–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

Basiswerte $E(i, 0)$ und $E(0, j)$, Markierungen für diff$(a_i, b_j) = 0$.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	–				–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

Zeilenweise ausgefüllt bis $E(4, 1)$.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	**3**				–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

$E(4,2) = \min\{3+0, 3+1, 4+1\} = 3.$

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	**3**	**3**	4	5	6	7	8	9	10	11
o	4	4	**3**	**4**			–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

$E(4,3) = \min\{3+1, 3+1, 3+1\} = 4$.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	**3**	4	4	5	**5**	6	7	8	9	10
n	5	5	4	4	5	**4**	5	6	7	8	9	10
e	6	6	5	5	5	5	5	6	7	**7**	8	9
n	7	7	6	6	6	**5**	6	6	7	8	8	9
t	8	8	7	7	7	6	6	7	7	8	9	9
i	9	9	8	8	8	7	7	7	**7**	8	9	10
e	10	10	9	9	9	8	8	8	8	**7**	8	9
l	11	11	10	**9**	10	9	9	9	9	8	**7**	**8**
l	12	12	11	**10**	10	10	10	10	10	9	**8**	**7**

Matrix ist komplett ausgefüllt. Ergebnis: $d(x, y) = E(12, 11) = 7$, rechte untere Ecke!

Ermittlung der Editier-Operationen in Zeit $O(m+n)$

Gegeben x und y, möchte man natürlich nicht nur die Editierdistanz $d(x,y)$, sondern auch die Schritte einer Transformation ermitteln, die mit $d(x,y)$ Schritten x in y überführt.
Wenn die Matrix `E[0..m,0..n]` mit den Werten $E(i,j)$ vorliegt, ist das leicht:

Wir wandern einen Weg von (m,n) nach $(0,0)$. Dabei gehen wir von (i,j) nach

$$\begin{array}{ll} (i-1,j-1), & \text{falls } E(i,j) = E(i-1,j-1) + \mathsf{diff}(a_i,b_j), \\ (i-1,j), & \text{falls } E(i,j) = E(i-1,j) + 1, \\ (i,j-1), & \text{falls } E(i,j) = E(i,j-1) + 1. \end{array}$$

Wenn mehr als eine Gleichung zutrifft, können wir einen der möglichen Schritte beliebig wählen.
Der gefundene Weg endet in $(0,0)$. Diesen Weg lesen wir dann rückwärts von $(0,0)$ nach (m,n). Dieser Weg repräsentiert in offensichtlicher Weise eine Transformation von x in y, wobei man Buchstaben gleich lässt, ersetzt, einfügt oder löscht. Man liest $x = x[1..m]$ von links nach rechts und generiert $y[1..n]$ von links nach rechts.

(1) $(i-1,j-1) \xrightarrow{b} (i,j)$ entspricht dem Übernehmen von a_i $(b=0)$ oder der Ersetzung durch b_j $(b=1)$;

(2) $(i-1,j) \xrightarrow{1} (i,j)$ entspricht dem Streichen von a_i;

(3) $(i,j-1) \xrightarrow{1} (i,j)$ entspricht dem Einfügen von b_j.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	**1**	1	2	3	4	5	6	7	8	9	10	11
x	**2**	2	2	3	4	5	6	7	8	9	10	11
p	3	**3**	3	3	4	5	6	7	8	9	10	11
o	4	4	**3**	4	4	5	**5**	6	7	8	9	10
n	5	5	4	**4**	5	**4**	5	6	7	8	9	10
e	6	6	5	5	**5**	5	5	6	7	**7**	8	9
n	7	7	6	6	6	**5**	6	6	7	8	8	9
t	8	8	7	7	7	6	**6**	**7**	7	8	9	9
i	9	9	8	8	8	7	7	7	**7**	8	9	10
e	10	10	9	9	9	8	8	8	8	**7**	8	9
l	11	11	10	**9**	10	9	9	9	9	8	**7**	**8**
l	12	12	11	**10**	10	10	10	10	10	9	**8**	**7**

Im Beispiel: $(0,0) \xrightarrow{1} (1,0) \xrightarrow{1} (2,0) \xrightarrow{1} (3,1) \xrightarrow{0} (4,2) \xrightarrow{1} (5,3) \xrightarrow{1} (6,4) \xrightarrow{0} (7,5) \xrightarrow{1} (8,6) \xrightarrow{1} (8,7) \xrightarrow{0} (9,8) \xrightarrow{0} (10,9) \xrightarrow{0} (11,10) \xrightarrow{0} (12,11)$, Kosten 7.

Dies führt zu der folgenden Zuordnung/Transformation, wobei die sieben „kostenpflichtigen“ Positionen rot markiert sind:

E	x	p	o	n	e	n	t	-	i	e	l	l
-	-	P	o	l	y	n	o	m	i	e	l	l

12.4 Optimale Matrizenmultiplikation

Multiplikation von zwei Matrizen über einem Ring R: Gegeben

$$A_1 = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1r_1} \\ \vdots & \vdots & \vdots \\ \alpha_{r_0 1} & \cdots & \alpha_{r_0 r_1} \end{pmatrix} \in R^{r_0 \times r_1}, A_2 = \begin{pmatrix} \beta_{11} & \cdots & \beta_{1r_2} \\ \vdots & \vdots & \vdots \\ \beta_{r_1 1} & \cdots & \beta_{r_1 r_2} \end{pmatrix} \in R^{r_1 \times r_2},$$

berechne Produkt $C := A_1 \cdot A_2$ mit

$$C = \begin{pmatrix} \gamma_{11} & \cdots & \gamma_{1r_2} \\ \vdots & \vdots & \vdots \\ \gamma_{r_0 1} & \cdots & \gamma_{r_0 r_2} \end{pmatrix} \in R^{r_0 \times r_2}, \text{ wobei } \gamma_{ik} = \sum_{1 \le j \le r_1} \alpha_{ij}\beta_{jk}, \text{ für } 1 \le i \le r_0,\ 1 \le k \le r_2.$$

Standardmethode benötigt $r_0 r_1 r_2$ R-Multiplikationen, $r_0(r_1 - 1)r_2$ R-Additionen.
Gesamtrechenzeit: $\Theta(r_0 r_1 r_2)$. **„Kosten“: $c(r_0, r_1, r_2) := r_0 r_1 r_2$.**

Wir betrachten „Iterierte Matrixmultiplikation“:

Gegeben sind $A_1, A_2, \ldots, A_k$, wobei A_i eine $r_{i-1} \times r_i$-Matrix über R ist.

Aufgabe: Berechne $A_1 \cdot A_2 \cdot \cdots \cdot A_k$, **möglichst kostengünstig**.

Die Matrizenmultiplikation ist **assoziativ**.

Beispiel $k = 3$: Wir betrachten die Anzahl der R-Multiplikationen, die bei der Berechnung von $A_1A_2A_3$ anfallen:

Kosten $r_0r_1r_2 + r_0r_2r_3$ bei der Klammerung $(A_1A_2)A_3$ und
Kosten $r_1r_2r_3 + r_0r_1r_3$ bei der Klammerung $A_1(A_2A_3)$.

Beispiel: $(r_0, r_1, r_2, r_3) = (5, 10, 3, 10)$.
Die erste Klammerung führt zu Kosten $5 \cdot 10 \cdot 3 + 5 \cdot 3 \cdot 10 = 300$, die zweite zu Kosten $10 \cdot 3 \cdot 10 + 5 \cdot 10 \cdot 10 = 800$.

Wir wollen für beliebiges k und beliebige $(r_0, r_1, \ldots, r_k)$ die Klammerung ermitteln, die die geringsten Kosten verursacht.

Methode: Dynamische Programmierung.

Problem:

Eingabe: Dimensionen $r_0 \times r_1$, $r_1 \times r_2$, ..., $r_{k-1} \times r_k$ von Matrizen $A_1, \ldots, A_k$.

Kurz: $(r_0, r_1, \ldots, r_k)$ für ein $k \geq 2$.

Ausgabe: Eine („*optimale*“) Klammerung, die bei der Berechnung von $A_1 \cdot \cdots \cdot A_k$ die **Gesamtkosten** (= Anzahl aller R-Multiplikationen) **minimiert**. Und: diese Kosten.

Seien $\boldsymbol{c(r_0, \ldots, r_k)}$ diese Kosten bei optimaler Klammerung, wobei $k \geq 2$ und $r_0, r_1, \ldots, r_k \geq 1$ beliebige ganze Zahlen sind.

Für $k = 1$ setzen wir $\boldsymbol{c(r_0, r_1)} := 0$ für alle r_0, r_1. (Es ist nichts zu tun.)

Klar: $c(r_0, r_1, r_2) = r_0 r_1 r_2$.

Sei $(r_0, r_1, \ldots, r_k)$ für ein $k \geq 3$ gegeben. Die optimale Lösung des Klammerungsproblems hat die Form

$$\underbrace{(A_1 \cdots A_\ell)}_{\substack{\text{irgendwie} \\ \text{geklammert}}} \underbrace{(A_{\ell+1} \cdots A_k)}_{\substack{\text{irgendwie} \\ \text{geklammert}}}$$

für ein $1 \leq \ell < k$ (das wir nicht kennen!).

Zentrale Beobachtung („Optimale Substruktur“):

Die Klammerungen in den Teilen $A_1 \cdots A_\ell$ bzw. $A_{\ell+1} \cdots A_k$ müssen optimale Kosten für die Teilprobleme liefern.

Wäre z. B. die Teilklammerung von $A_1 \cdots A_\ell$ nicht optimal, dann könnte man durch **Verbesserung dieses Teils** die **Gesamtkosten senken**.

Aus dieser Beobachtung ergibt sich: Es gibt ein ℓ, $1 \leq \ell < k$, mit

$$c(r_0, \ldots, r_k) = c(r_0, \ldots, r_\ell) + c(r_\ell, \ldots, r_k) + \underbrace{r_0 r_\ell r_k}_{\substack{\text{äußere} \\ \text{Matrizenmult.}}} .$$

Der letzte Summand stellt die Kosten für die abschließende Multiplikation einer $r_0 \times r_\ell$- mit einer $r_\ell \times r_k$-Matrix dar.

Wir suchen also ein ℓ, das die Summe auf der rechten Seite minimiert.

Wir benötigen optimale Klammerungen bzw. ihre Kosten für die Teilprobleme $A_1 \cdots A_\ell$ und $A_{\ell+1} \cdots A_k$.

Ansatz: Betrachte die **Teilprobleme** $TP(i, j)$, die minimalen Kosten für die Berechnung des Teilprodukts $A_i \cdots A_j, 1 \leq i \leq j \leq k$.

Parametersatz zu $TP(i, j)$: $(r_{i-1}, \ldots, r_j)$.

Beobachtung („Optimale Substruktur“):

Optimale Klammerung für $A_i \cdots A_j$ ist $\underbrace{(A_i \cdots A_\ell)}_{\text{irgendwie geklammert}}\underbrace{(A_{\ell+1} \cdots A_j)}_{\text{irgendwie geklammert}}$, wobei die Klammerungen in den beiden Teilen optimal sein müssen.

„Bellmansche Optimalitätsgleichungen“:

$$c(r_{i-1}, \ldots, r_j) = \min\left\{ c(r_{i-1}, \ldots, r_\ell) + c(r_\ell, \ldots, r_j) + r_{i-1} r_\ell r_j \;\middle|\; i \leq \ell < j \right\},$$

für $1 \leq i < j \leq k$. (Gilt auch für $j = i + 1$, trivialerweise.)

Basisfall: $c(r_{i-1}, r_i) = 0$, für $1 \leq i \leq k$.

Um die optimalen Werte zu berechnen, müssen wir nur die Rekursionsformel iterativ auswerten. Erweiterung zur Ermittlung der optimalen Klammerung führt zu folgendem Programm.

MatrixOptimal$((r_0, \ldots, r_k))$
Eingabe: Dimensionsvektor $(r_0, \ldots, r_k)$
Ausgabe: Kosten $c(r_0, \ldots, r_k)$ bei der optimalen Klammerung
`l[1..k,1..k]`: Plan zur Ermittlung der optimalen Unterteilung;
Datenstruktur: Matrizen `C[1..k,1..k]`, `l[1..k,1..k]`
Ziel: `C[i,j]` enthält $c(r_{i-1}, \ldots, r_j)$
`l[i,j]` enthält Index zur Unterteilung der Multiplikation bei $A_i \cdots A_j$

(1) **for** `i` **from** 1 **to** k **do** `C[i,i]` $\leftarrow 0$;
(2) **for** `i` **from** 1 **to** $k-1$ **do** `C[i,i+1]` $\leftarrow r_{\texttt{i}-1} \cdot r_{\texttt{i}} \cdot r_{\texttt{i}+1}$;
(3) **for** `d` **from** 2 **to** $k-1$ **do**
(4) **for** `i` **from** 1 **to** $k-$ `d` **do**
(5) bestimme das ℓ, `i` $\leq \ell <$ `i+d`,
(6) das $C =$ `C[i,`ℓ`]` $+$ `C[`$\ell+1$`,i+d]` $+ r_{\texttt{i}-1} \cdot r_\ell \cdot r_{\texttt{i+d}}$ minimiert;
(7) `l[i,i+d]` $\leftarrow$ dieses ℓ;
(8) `C[i,i+d]` $\leftarrow$ das minimale C;
(9) **Ausgabe:** `C[1..k,1..k]` und `l[1..k,1..k]`.

Korrektheit: Man beweist durch Induktion über $d = 0, 1, \ldots, k-1$ mit Hilfe der Bellmanschen Optimalitätsgleichungen, dass $\texttt{C}[i, i+d]$ die Zahl $c(r_{i-1}, \ldots, r_{i+d})$ enthält.

Zur Übung: Man zeige, dass mit Hilfe der $\texttt{l}[i, j]$-Werte die optimale Klammerung in Linearzeit ermittelt werden kann.

Laufzeit: Die Minimumssuche in Zeilen (5)–(6) kostet Zeit $O(k)$; mit den geschachtelten Schleifen (3)–(8) und (4)–(8) ergibt sich eine Rechenzeit von $\Theta(k^3)$.

Beispiel: $(r_0, \ldots, r_4) = (3, 2, 2, 5, 6)$.

`c`:

i \ j	1	2	3	4
1	0	12	42	108
2		0	20	80
3			0	60
4				0

`l`:

i \ j	1	2	3	4
1	$*$	$*$	2	2
2		$*$	$*$	3
3			$*$	$*$
4				$*$

$12 = 3 \cdot 2 \cdot 2 \qquad 20 = 2 \cdot 2 \cdot 5 \qquad 60 = 2 \cdot 5 \cdot 6$

$42 = 12 + 3 \cdot 2 \cdot 5 \quad 80 = 20 + 2 \cdot 5 \cdot 6 \; (< 60 + 2 \cdot 2 \cdot 6) \ldots$

Optimale Klammerung: $(A_1 A_2)(A_3 A_4)$.

Kosten: 108 Multiplikationen.

ENDE

Viel Erfolg bei der Prüfungsvorbereitung und in der Prüfung!