10 Dynamische Programmierung

Das Prinzip der *Dynamischen Programmierung* wird häufig bei Fragestellungen auf Worten angewendet.

10.1 Längste gemeinsame Teilfolge

Wir betrachten Worte der Art $w = a_1 a_2 \dots a_m$, wobei a_i ein Zeichen aus einem Alphabet Σ ist. Das Alphabet kann zum Beispiel $\Sigma = \{\text{a, b, } \dots, \text{ z}\}$ sein.

Ein Wort lässt sich als einfaches Array

$$w[1, \dots, m] \quad \text{mit } w[i] = a_i$$

darstellen. So können wir direkt auf die einzelnen Zeichen zugreifen. Zunächst müssen wir noch klären, was wir unter einer Teilfolge verstehen wollen.

Definition 10.1(Teilfolge):

Das Wort v ist eine Teilfolge von $w = a_1 a_2 \dots a_m \iff$ es gibt eine Folge von Indizes $1 \leq i_1 \lneq i_2 \lneq \dots \lneq i_k \leq m$ so dass $v = a_{i_1} a_{i_2} \dots a_{i_k}$.

Oder anders: Eine Teilfolge v von w entsteht dadurch, dass wir aus w einzelne Zeichen(positionen) auswählen und dann in der *gleichen Reihenfolge*, wie sie in w auftreten, als v hinschreiben.

Beispiel 10.1: *Ist $w = abcd$, so sind*

$$a, b, c, d, \quad ab, ac, ad, bc, bd, cd, \quad abc, acd$$

und abcd sowie das leere Wort ε alles Teilworte von w.

Ist $w = a_1 a_2 \dots a_m$, dann entspricht eine Teilfolge von w einer Folge über $\{0, 1\}$ der Länge m. Die Anzahl der Teilworte von w ist $\leq 2^m$.

Beispiel 10.2:

$$\begin{aligned} w = abcd &\Rightarrow \#\,Teilfolgen = 16, \quad aber \\ w = aaaa &\Rightarrow \#\,Teilfolgen = 5. \end{aligned}$$

Beim Problem der längsten gemeinsamen Teilfolge haben wir zwei Worte v und w gegeben und suchen ein Wort u so dass:

- Das Wort u Teilfolge von v *und* w ist.

- Es gibt kein s mit $|s| \gneq |u|$ und s ist Teilfolge von v und w.

Hier ist $|w|$ = #Zeichen von w, die Länge von w. Wir setzen $|\varepsilon| = 0$. Bezeichnung $u = \mathrm{lgT}(v, w)$.

Beispiel 10.3:

1. *Ist etwa* $v = abab$ *und* $w = baba$*, so sind* aba *und* bab *jeweils* $\mathrm{lgT}(v, w)$.
2. *Ist* $w = aba$ *und* $v = acacac$*, so ist* aa *eine* $\mathrm{lgT}(v, w)$.

Also: Die Zeichen der längsten gemeinsamen Teilfolge brauchen in v und w nicht direkt hintereinander stehen.

Man kann $\mathrm{lgt}(v, w)$ auf $\mathrm{lgT}(v', w')$ mit $|v'| + |w'| \lneq |v| + |w|$ zurückführen.

Satz 10.1: *Ist* $v = a_1 \dots a_m$ *und* $w = b_1 \dots b_n$*, so gilt:*

1. *Ist* $a_m = b_n = a$*, so ist jede* $\mathrm{lgT}(v, w)$ *von der Form* ua*. Und* u *ist eine* $\mathrm{lgT}(a_1 \dots a_{m-1}, b_1 \dots b_{m-1})$.
2. *Ist* $a_m \neq b_n$*, so ist jede* $\mathrm{lgT}(v, w)$ *der Art*

$$u_1 = \mathrm{lgT}(a_1 \dots a_{m-1}, w) \quad \textit{oder} \quad u_2 = \mathrm{lgT}(v, b_1 \dots b_{n-1}).$$

Beweis. 1. Ist r eine Teilfolge von v und w ohne a am Ende, so ist ra eine längere gemeinsame Teilfolge. (Beispiel: $v = aa$, $w = aaa \to \mathrm{lgT}(v, w) = \mathrm{lgT}(a, aa)a$.)

Ist ua eine gemeinsame Teilfolge von v und w aber u keine $\mathrm{lgT}(a_1 \dots a_{m-1}, b_1 \dots b_{n-1})$, so ist ua auch keine $\mathrm{lgT}(v, w)$.

2. Für jede gemeinsame Teilfolge u von v und w gilt eine der Möglichkeiten:
   - u ist Teilfolge von $a_1 \dots a_m$ und $b_1 \dots b_{n-1}$
   - u ist Teilfolge von $a_1 \dots a_{m-1}$ und $b_1 \dots b_n$
   - u ist Teilfolge von $a_1 \dots a_{m-1}$ und $b_1 \dots b_{n-1}$

   (Beispiel: $v = abba, w = baab \quad \Rightarrow \quad ab$ ist $\mathrm{lgT}(abb, baab)$, ba ist $\mathrm{lgT}(abba, baa)$.)

□

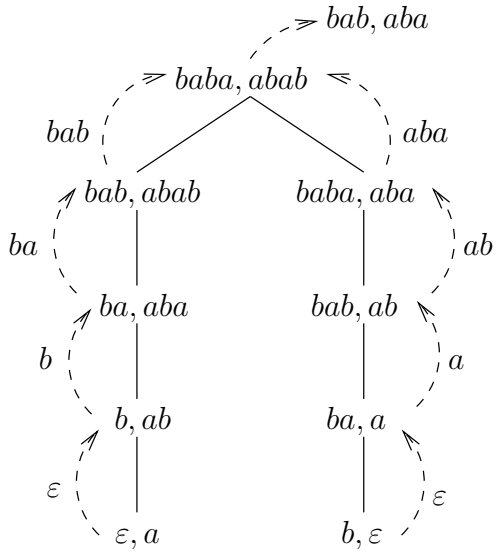
Hier wird das Prinzip der *Reduktion auf optimale Lösungen von Teilproblemen* genutzt. Wir betrachten zunächst den rekursiven Algorithmus.

Algorithmus 18: $\mathrm{lgT}(v, w)$

Input : $v = a_1 \dots a_m,\ w = b_1 \dots b_n$
Output : längste gemeinsame Teilfolge von v und w

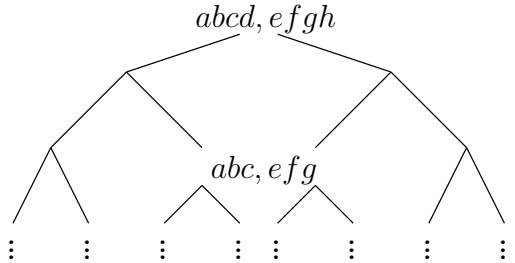
```
1  if m == 0 oder n == 0 then
2  |  return ε
3  end
4  if a_m == b_n then
5  |  d = lgT(a_1 ... a_{m-1}, b_1 ... b_{n-1});
6  |  return „d verlängert um a_m“;
7  else
8  |  d = lgT(a_1 ... a_{m-1}, b_1 ... b_n);
9  |  e = lgT(a_1 ... a_m, b_1 ... b_{n-1});
10 |  return „Längeres von d, e“ ;        /* bei |d| = |e| beliebig */
11 end
```

Betrachten wir nun den Aufrufbaum für $v = baba$ und $w = abab$.

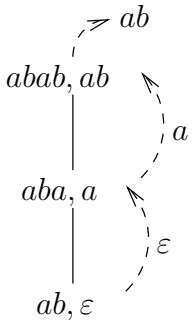


Die längste gemeinsame Teilfolge ist also *aba* uder *bab*. Die Struktur des Baumes ist *nicht* statisch. Sie hängt von der Eingabe ab.

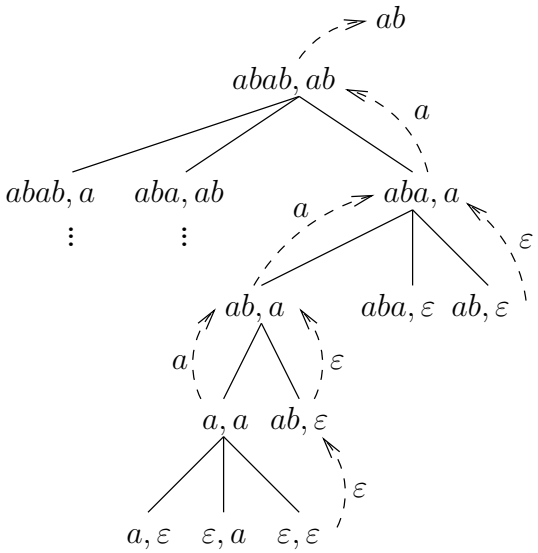
Die Größe des Baumes ist $\geq 2^n$ bei $|v| = |w| = n$.



Beispiel 10.4:



So finden wir den Teil ab $\underline{ab}$ *und* $\underline{ab}$ *als längste gemeinsame Teilfolge. Wie bekommen wir alle Möglichkeiten?*



⇒ *Immer alle Wege gehen. Bei gleicher maximalen Länge beide merken.*

Sei jetzt wieder $|v| = m$, $|w| = n$. Wieviele *verschiedene* haben wir?

$$\text{Aufruf} \Longleftrightarrow \text{zwei Anfangsstücke von } v \text{ und } w$$

Das Wort v hat $m+1$ Anfangsstücke (inklusive v und ε), w hat $n+1$ Anfangsstücke. Also #Anfangsstücke $\leq (m+1) \cdot (n+1)$.

Wir merken uns die Ergebnisse der verschiedenen Aufrufe in einer Tabelle $T[k, h]$, $0 \leq k \leq m$, $0 \leq h \leq n$.

$$T[k, h] = \text{Länge einer lgT}(a_1 \ldots a_k, b_1 \ldots b_h)$$

Dann ist

$$\begin{aligned} T[0,h] &= 0 \quad \text{für } 0 \le h \le m \\ T[k,0] &= 0 \quad \text{für } 0 \le k \le n \end{aligned}$$

und weiter für $h, k \gneq 0$

$$T[k,h] = \begin{cases} T[k-1,h-1]+1 & \text{falls } a_k = b_h \\ \max\{T[k,h-1], T[k-1,h]\} & \text{falls } a_k \neq b_h. \end{cases}$$

Das Mitführen von Pointern zu den Maxima erlaubt die Ermittlung aller $\mathrm{lgT}(v,w)$ mit ihren Positionen.

Laufzeit: $\Theta(m \cdot n)$, da pro Eintrag $O(1)$ anfällt.

Beispiel 10.5: *$v = abc$, $w = abcd$* $\quad T[k,h]$ *mit* $0 \le k \le 3$, $0 \le h \le 4$

$k \backslash h$	T	*0*	*a* *1*	*b* *2*	*c* *3*	*d* *4*
	0	*0*	*0*	*0*	*0*	*0*
a	*1*	*0*	*1*	*1*	*1*	*1*
b	*2*	*0*	*1*	*2*	*2*	*2*
c	*3*	*0*	*1*	*2*	*3*	***3***
			a	*b*	*c*	

$k \backslash h$	B	*0*	*a* *1*	*b* *2*	*c* *3*	*d* *4*
	0	•	←	←	←	←
a	*1*	↑	↖	←	←	←
b	*2*	↑	↑	↖	←	←
c	*3*	↑	↑	↑	↖	←
			a	*b*	*c*	

$$\nwarrow \Leftrightarrow +1 \qquad \uparrow, \leftarrow \Leftrightarrow +0$$

Die Tabelle füllen wir spaltenweise von links nach rechts *oder zeilenweise von* oben nach unten *aus. Zusätzlich haben wir noch eine Tabelle $B[k,h]$ gleicher Dimension für die $\nwarrow, \uparrow, \leftarrow$.*

10.2 Optimaler statischer binärer Suchbaum

Wir betrachten im folgenden das Problem des *optimalen statischen binären Suchbaumes*. Uns interessieren dabei nur die Suchoperationen, Einfügen und Löschen von Elementen findet nicht statt.

Gegeben sind n verschiedene (Schlüssel-)Werte, $a_1 \lneq a_2 \lneq \cdots \lneq a_n$, mit den zugehörigen zugriffswahrscheinlichkeiten. Auf den Wert a_1 wird mit Wahrscheinlichkeit p_1, auf a_2 mit p_2, usw., zugegriffen. Dabei gilt

$$\sum_{i=1}^{n} p_i = 1.$$

Gesucht ist ein *binärer Suchbaum* T, der die Werte $a_1, \ldots, a_n$ enthält. Die Kosten für das Suchen in T mit den Häufigkeiten p_i sollen *minimal* sein.

Was sind die Kosten von T?

Definition 10.2: *(a)* $\text{Tiefe}_T(a_i)$ *bezeichnet die Tiefe von* a_i *in* T*, also die Anzahl der Kanten von der Wurzel von* T *bis zum Knoten* a_i*. Bei der Suche nach* a_i *sind demnach* $\text{Tiefe}_T(a_i) + 1$ *viele Vergleichsoperationen nötig. Wir summieren die Kosten für die einzelnen* a_i *gewichtet mit der Wahrscheinlichkeit, dass dieser Wert gesucht wird.*

$$K(T) := \sum_{i=1}^{n} p_i \cdot \big(\text{Tiefe}_T(a_i) + 1\big)$$

(b) T *ist optimal genau dann, wenn gilt:*

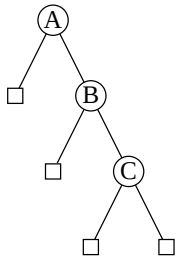
$$K(T) = \min\{K(S) \mid S \text{ ist ein binärer Suchbaum für } a_1, \ldots, a_n\}$$

Beachte: Ein optimaler Suchbaum existiert immer, da es nur endlich viele Suchbäume zu $a_1, \ldots, a_n$ gibt.

Beispiel 10.6:

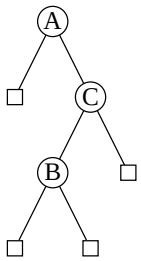
$$a_1 = A, \quad a_2 = B, \quad a_3 = C, \qquad p_1 = \frac{4}{10}, \quad p_2 = \frac{3}{10}, \quad p_3 = \frac{3}{10}$$

1.



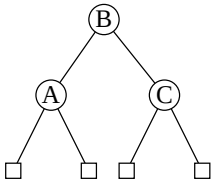
$$K = \frac{\overbrace{1 \cdot 4}^{A} + \overbrace{2 \cdot 3}^{B} + \overbrace{3 \cdot 3}^{C}}{10} = \frac{19}{10}$$

2.



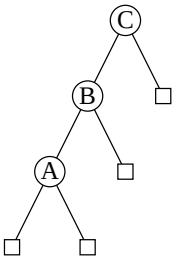
$$K = \frac{\overbrace{1 \cdot 4}^{A} + \overbrace{3 \cdot 3}^{B} + \overbrace{2 \cdot 3}^{C}}{10} = \frac{19}{10}$$

3.



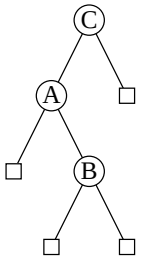
$$K = \frac{\overbrace{2 \cdot 4}^{A} + \overbrace{1 \cdot 3}^{B} + \overbrace{2 \cdot 3}^{C}}{10} = \frac{\mathbf{17}}{\mathbf{10}}$$

4.



$$K = \frac{\overbrace{3 \cdot 4}^{A} + \overbrace{2 \cdot 3}^{B} + \overbrace{1 \cdot 3}^{C}}{10} = \frac{21}{10}$$

5.

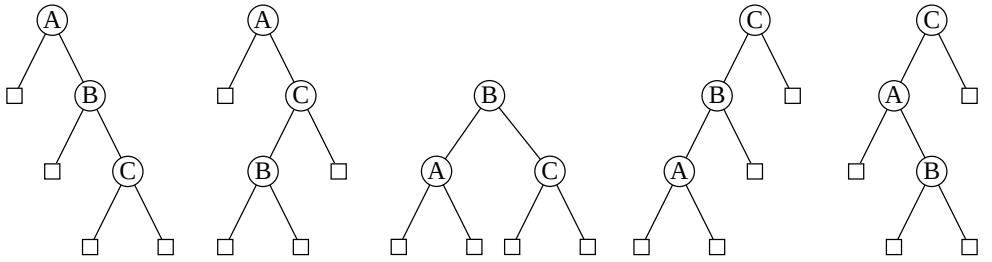


$$K = \frac{\overbrace{2 \cdot 4}^{A} + \overbrace{3 \cdot 3}^{B} + \overbrace{1 \cdot 3}^{C}}{10} = \frac{20}{10}$$

Beobachtung: Es reicht nicht aus, einfach das häufigste Element an die Wurzel zu setzen.

Beispiel 10.7: *Es müssen auch nicht zwangsläufig ausgeglichene Bäume entstehen. Wie man hier sieht.*

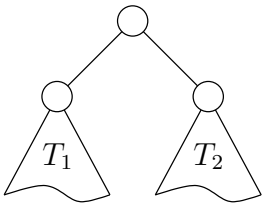
$$a_1 = A, \quad a_2 = B, \quad a_3 = C, \qquad p_1 = \frac{5}{9}, \quad p_2 = \frac{2}{9}, \quad p_3 = \frac{2}{9}$$



Kosten: $\frac{5+4+6}{9} = \mathbf{\frac{15}{9}}$ $\frac{5+6+4}{9} = \mathbf{\frac{15}{9}}$ $\frac{10+2+4}{9} = \frac{16}{9}$ $\frac{15+4+2}{9} = \frac{21}{9}$ $\frac{10+6+2}{9} = \frac{18}{9}$

Unser Ziel ist es, den optimalen Suchbaum in einer Laufzeit von $O(n^3)$ zu bestimmen.

Haben wir einen beliebigen binären Suchbaum für $a_1 \lneq a_2 \lneq \cdots \lneq a_n$ gegeben, so lassen sich seine Kosten bei Häufigkeiten p_i folgendermaßen ermitteln:



Wurzel betreten: $1 = \sum_{a_j \in T} p_j$

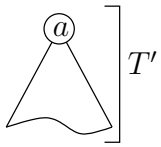
Linker Sohn der Wurzel: $\sum_{a_i \in T_1} p_i$

Rechter Sohn der Wurzel: $\sum_{a_k \in T_2} p_k$

...

Allgemein sagen wir die Kosten, die ein Teilbaum von T mit der Wurzel a zu den Gesamtkosten beiträgt, ist $K_T(a)$ mit

$$K_T(a) = \sum_{a_j \in T'} p_j$$

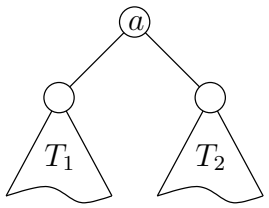
T' = Teilbaum von T mit Wurzel a. $\quad a_i \in$ 

Folgerung 10.1: *Für den binären Suchbaum S auf $a_1, \ldots, a_n$ gilt:*

$$K(S) = \sum_{i=1}^{n} K_S(a_i)$$

Beweis. Induktion über n. Für $n = 1$ ist $p_1 = 1$ und damit $K(T) = 1$ ✓.

Induktionsschluß: Sei $n \geq 1$. Wir betracheten $a_1, \ldots, a_{n+1}$ mit beliebigen p_i. Es ist

$S =$ 

$a = a_j$ für ein a_j

Dann ist

$$\begin{aligned} K(S) &= \sum_{i=1}^{n+1} p_i \cdot \big(\text{Tiefe}_S(a_i) + 1\big) \quad \text{nach Definition} \\ &= \underbrace{\sum_{i=1}^{n+1} p_i}_{=1} + \sum_{i=1}^{n+1} p_i \cdot \text{Tiefe}_S(a_i) \end{aligned}$$

$\text{Tiefe}_S(a_j) = 0$ Tiefe der Wurzel

$$\begin{aligned} K(S) &= 1 + \sum_{a_i \in T_1} p_i \cdot \text{Tiefe}_S(a_i) + \sum_{a_k \in T_2} p_k \cdot \text{Tiefe}_S(a_k) \\ &= 1 + \sum_{a_i \in T_1} p_i \cdot \big(1 + \text{Tiefe}_{T_1}(a_i)\big) + \sum_{a_k \in T_2} p_k \cdot \big(1 + \text{Tiefe}_{T_2}(a_k)\big) \end{aligned}$$

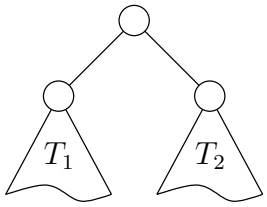
mit der Induktionsvoraussetzung für T_1 und T_2

$$\begin{aligned} &= 1 + \sum_{a_i \in T_1} K_{T_1}(a_i) + \sum_{a_k \in T_2} K_{T_2}(a_k) \\ &= K_S(a_j) + \sum_{a_i \in T_1} K_S(a_i) + \sum_{a_k \in T_2} K_S(a_k) \\ &= \sum_{i=1}^{n+1} K_S(a_i) \end{aligned}$$

□

Folgerung 10.2: *Ist für eine Menge $b_1, \ldots, b_l$ mit p_i, $\sum p_i \leq 1$*

$S =$



ein Baum, so dass $\sum K_S(b_i)$ minimal ist, so ist für T_1 und T_2

$$\sum_{b_i \in T_1} K_{T_1}(b_i), \quad \sum_{b_i \in T_2} K_{T_2}(b_i)$$

minimal.

Beweis.

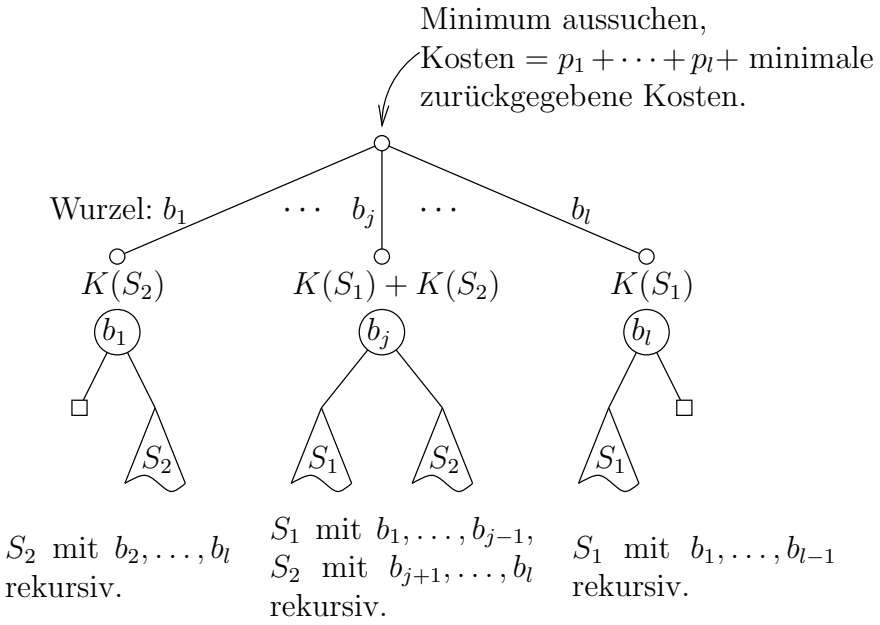
$$\sum_{i=1}^{l} K_S(b_i) = \sum_{i=1}^{l} p_i + \sum_{b_i \in T_1} K_{T_1}(b_i) + \sum_{b_i \in T_2} K_{T_2}(b_i)$$

□

Prinzip: Teile optimaler Lösungen sind optimal. Diese Eigenschaft wird beim dynamischen Programmieren immer genutzt.

Bei der Eingabe von $b_1, \ldots, b_l$, $b_1 \lneq \cdots \lneq b_l$ mit Wahrscheinlichkeiten p_i, $\sum p_i \leq 1$ ermitteln wir rekursiv einen Baum S mit

$$\sum_{i=1}^{l} K_S(b_i) \quad \text{minimal.}$$



Laufzeit: $\gg \Omega(2^l)$, 2^l Aufrufe werden alleine für ganz links und ganz rechts benötigt. Die Struktur des Aufrufbaumes ist statisch. Wieviele verschiedene Aufrufe sind darin enthalten?

$$
\begin{aligned}
\#\text{verschiedene Aufrufe} &\leq \#\text{verschiedene Aufrufe der Art} \\
&\quad b_k \lneq b_{k+1} \lneq \cdots \lneq b_{h-1} \lneq b_h, \text{ wobei } 1 \leq k \leq h \leq l \\
&= \big|\{(k,h) \mid 1 \leq k \leq h \leq l\}\big| \\
&= \underbrace{l}_{k=1} + \underbrace{(l-1)}_{k=2} + \underbrace{(l-2)}_{k=3} + \cdots + \underbrace{1}_{k=l} \quad \#\text{Möglichkeiten für } h \\
&= \frac{l(l+1)}{2} = O(l^2).
\end{aligned}
$$

Bei $a_1, \ldots, a_n$ mit p_i, $\sum p_i = 1$ Tabelle $T[k,h]$, $1 \leq k \leq h \leq n$ mit der Bedeutung

$$T[k,h] \Leftrightarrow \text{ optimale Kosten für einen Baum } S_{k,h} \text{ mit den Werten } a_k, \ldots, a_h.$$

Beispiel 10.8: *Für $n = 4$ ergeben sich die Tabelleneinträge ergeben wie folgt:*

$k \backslash h$	1	2	3	4
1				
2				
3				
4				

Für $k > h$ keine Einträge.

$$
\begin{aligned}
T[1,1] &= p_1, \qquad T[2,2] = p_2, \qquad T[3,3] = p_3, \qquad T[4,4] = p_4 \\
\\
T[1,2] &= \min\Big\{ \underbrace{(p_1 + p_2 + T[2,2])}_{a_1 \text{ als Wurzel}},\ \underbrace{(p_2 + p_1 + T[1,1])}_{a_2 \text{ als Wurzel}} \Big\} \\
T[2,3] &= \min\big\{(p_2 + p_3 + T[3,3]),\ (p_3 + p_2 + T[2,2])\big\} \\
T[3,4] &= \min\big\{(p_3 + p_4 + T[4,4]),\ (p_4 + p_3 + T[3,3])\big\} \\
\\
T[1,3] &= p_1 + p_2 + p_3 + \min\{T[2,3],\ (T[1,1] + T[3,3]),\ T[1,2]\} \\
T[2,4] &= p_1 + p_2 + p_3 + \min\{T[3,4],\ (T[2,2] + T[4,4]),\ T[2,3]\} \\
\\
T[1,4] &= p_1 + p_2 + p_3 + p_4 + \\
&\qquad + \min\{T[2,4],\ (T[1,1] + T[3,4]),\ (T[1,2] + T[4,4]),\ T[1,3]\}
\end{aligned}
$$

Allgemein: Wir haben $O(n^2)$ Einträge in T. Pro Eintrag fätllt eine Zeit von $O(n)$ an. (Das lässt sich induktiv über $h - k$ zeigen.) Also lässt sich das Problem in der Zeit $O(n^3)$ lösen.

Für alle Tabellenenträge gilt:

$$
\begin{aligned}
T[k,h] &= p_k + \cdots + p_h + \min\Big\{ \big\{T[k+1,h]\big\} \cup \big\{T[k,h-1]\big\} \cup \\
&\qquad \cup \big\{T[k,i-1] + T[i+1,h] \mid k+1 \le i \le h-1\big\} \Big\}
\end{aligned}
$$

Beispiel 10.9:

$$
a_1 = A, \quad a_2 = B, \quad a_3 = C, \qquad p_1 = \frac{4}{10}, \quad p_2 = \frac{3}{10}, \quad p_3 = \frac{3}{10}
$$

$K(S)$	A	B	C
A	$\frac{4}{10}$	$\frac{10}{10}$	$\mathbf{\frac{17}{10}}$
B	–	$\frac{3}{10}$	$\frac{9}{10}$
C	–	–	$\frac{3}{10}$

Wurzel	A	B	C
A	A	A	**B**
B	–	B	B *oder* C
C	–	–	C

$$
T[A,C] = \frac{10}{10} + \min\left\{ \underbrace{T[B,C]}_{=\frac{9}{10}},\ \underbrace{T[A,A] + T[C,C]}_{=\frac{7}{10}},\ \underbrace{T[A,B]}_{=\frac{10}{10}} \right\}
$$

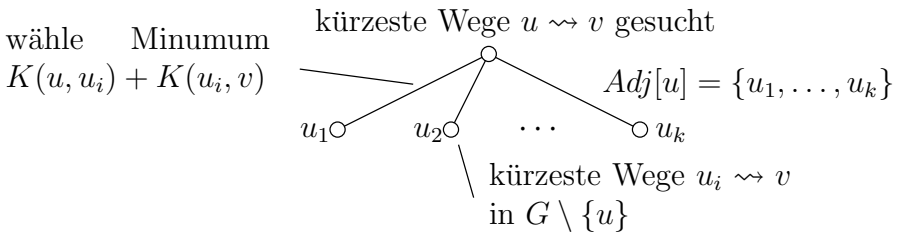
10.3 Kürzeste Wege mit negativen Kantengewichten

Ab jetzt betrachten wir wieder eine beliebige Kostenfunktion $K : E \to \mathbb{R}$.

Der Greedy-Ansatz wie bei Dijkstra funktioniert nicht mehr. Kürzeste Wege $u \circ\!\!\longrightarrow\!\!\circ\, v$ findet man durch Probieren.

Die Laufzeit ist dann $> (n-2)! \geq 2^{\Omega(n \cdot \log n)} \gg 2^n$ (!)

Es werden im Allgemeinen viele Permutationen generiert, die gar keinen Weg ergeben. Das kann mit Backtracking vermieden werden:



Dies ist korrekt, da Teilwege kürzester Wege wieder kürzeste Wege sind und wir mit kürzesten Wegen immer nur einfache Wege meinen.

Algorithmus 19: Kürzeste Wege mit Backtracking

Input : $G = (V, E)$ gerichteter Graph, $K : E \to \mathbb{R}$ beliebige Kantengewichte, u Startknoten, v Zielknoten

```
1 KW(G, u, v);
```

Das Ganze ist leicht durch Rekursion umzusetzen:

Prozedur KW(W, u, v)

Input : W noch zu betrachtende Knotenmenge, u aktueller Knoten, v Zielknoten
Output : Länge eines kürzesten Weges von u nach v

```
1  if u == v then
2  |  return 0;
3  end
4  l = ∞;
5  foreach w ∈ Adj[u] ∩ W do
6  |  l' = KW(W \ {u}, w, v);          /* Ein kürzester Weg (w, ..., v) */
7  |  l' = K(u, w) + l';              /* Die länge des Weges (u, w, ..., v) */
8  |  if l' < l then
9  |  |  l = l';                      /* Neuen kürzesten Weg merken. */
10 |  end
11 end
12 return l;                          /* Ausgabe ∞, wenn Adj[u] ∩ W = ∅ */
```

Die Korrektheit läßt sich mittels Induktion über $|W|$ zeigen.

Etwas einfacher ist es, alle einfachen Wege $a \circ\!\!\longrightarrow\!\!\circ\, b$ systematisch zu erzeugen und den Längenvergleich nur am Ende durchzuführen.

Idee: Wir speichern den Weg vom Startknoten u bis zum aktuellen Knoten. Wenn ein neuer kürzester Weg gefunden wird, dann abspeichern.

Prozedur KW(W, u, v)

Data : L, M als globale Arrays. L ist als Stack implementiert enthält den aktuell betrachteten Weg. In M steht der aktuell kürzeste Weg. Vorher initialisiert mit $L = (a)$, a Startknoten und $M = \emptyset$.

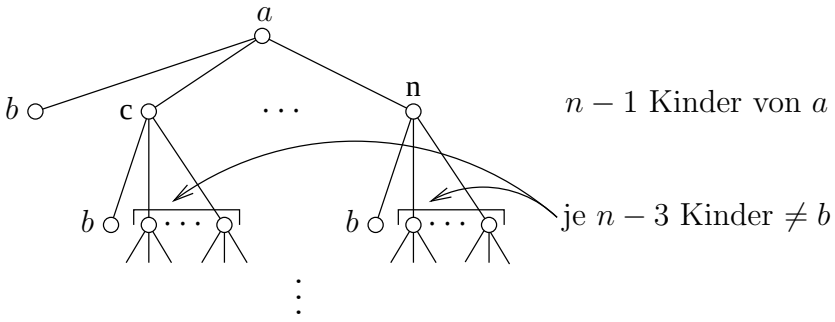
```
1  if u == v then
   |  /* K(L), K(M) Kosten des Weges in L bzw. M */
2  |  if K(L) < K(M) then
3  |  |  M = L;
4  |  end
5  else
6  |  foreach w ∈ Adj[u] ∩ W do
7  |  |  push(L, w);
8  |  |  KW(W \ {u}, w, v);
9  |  |  pop(L);
10 |  end
11 end
```

Korrektheit mit der Aussage:

> Beim Aufruf von KW(W, w, v) ist $L = (w \dots a)$. Am Ende des Aufrufs KW(W, w, v) enthält M einen kürzesten Weg der Art $M = (b \dots L)$. (L ist das L von oben)

Das Ganze wieder induktiv über $|W|$.

Laufzeit der rekursiven Verfahren? Enthält der Graph alle $n(n-1)$ Kanten, so sieht der Aufrufbaum folgendermaßen aus:



Also #Blätter $\geq (n-2) \cdot (n-3) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \geq 2^{\Omega(n \cdot \log n)}$.

Wieviele verschiedene Aufrufe haben wir höchstens? Also Aufrufe KW(W, c, d)?

$$\begin{aligned} W \subseteq V &\leq 2^n \text{ Möglichkeiten} \\ c, d &\leq n \text{ Möglichkeiten, da Endpunkt immer gleich} \end{aligned}$$

Also $n \cdot 2^n = 2^{\log n} \cdot 2^n = 2^{n+\log n} = 2^{O(n)}$ viele verschiedene Aufrufe.

Bei $2^{\Omega(n \log n)}$ Aufrufen wie oben sind viele doppelt. Es ist ja

$$\frac{2^{O(n)}}{2^{\Omega(n \log n)}} = \frac{1}{2^{\Omega(n \log n) - O(n)}} \geq \frac{1}{2^{\varepsilon n \cdot \log n - c \cdot n}} \geq \frac{1}{2^{\Omega(n \log n)}}.$$

Die Anzahl verschiedener Aufrufe ist nur ein verschwindend kleiner Anteil an allen Aufrufen.

Vermeiden der doppelten Berechnung durch Merken (Tabellieren). Dazu benutzen wir das zweidimensionale Array $T \underbrace{[1 \ldots 2^n]}_{\in \{0,1\}^n}[1 \ldots n]$ mit der Interpretation:

> Für $W \subseteq V$ mit $b, v \in W$ ist $T[W, v]$ = Länge eines kürzesten Weges $v \circ\!\!\longrightarrow\!\!\circ\, b$ nur durch W.

Füllen $T[W, v]$ für $|W| = 1, 2, \ldots$

1. $|W| = 1$, dann $v = b \in W, T[\{b\}, b] = 0$
2. $|W| = 2$, dann $T[W, b] = 0$, $T[W, v] = K(v, b)$, wenn $v \neq b$
3. $|W| = 3$, dann

$$\begin{aligned} T[W, b] &= 0 \\ T[W, v] &= \min\{K(v, u) + T[W', u] \mid W' = W \setminus \{v\}, u \in W\} \\ T[W, v] &= \infty, \text{ wenn keine Kante } v \circ\!\!\longrightarrow\!\!\circ\, u \quad \text{mit } u \in W \end{aligned}$$

Das Mitführen des Weges durch $\pi[W, v] = u$, $\quad u$ vom Minimum ermöglicht eine leichte Ermittlung der Wege.

Der Eintrag $T[W, v]$ wird so gesetzt:

Prozedur Setze(W, v, b)

```
1  if v == b then
2  |  T[W, v] = 0;
3  |  π[W, v] = b;
4  else
5  |  W' = W \ {v};                          /* Hier ist |W| ≥ 2 */
6  |  T[W, v] = ∞;
7  |  foreach u ∈ Adj[v] ∩ W' do
8  |  |  l = K(u, v) + T[W', v];
9  |  |  if l < T[W, v] then
10 |  |  |  T[W, v] = l;
11 |  |  |  π[W, v] = u;
12 |  |  end
13 |  end
14 end
```

Dann insgesamt:

Prozedur KW(V, a, b)

```
   /* Gehe alle Teilmengen von V, die den Zielknoten b enthalten,
      der größe nach durch. */
1  for i = 1 to n do
2  |  foreach W ⊆ V mit b ∈ W, |W| = i do
3  |  |  foreach v ∈ W do
4  |  |  |  Setze(W, v, b)
5  |  |  end
6  |  end
7  end
```

Dann enthält $T[V, a]$ das Ergebnis. Der Weg ist

$$\begin{aligned} a_0 &= a, \\ a_1 &= \pi[V, a], \\ a_2 &= \pi[V \setminus \{a\}, a_1], \\ &\dots, \\ a_i &= \pi[V \setminus \{a_0, \dots, a_{i-2}\}, a_{i-1}], \\ &\dots \end{aligned}$$

Die Korrektheit ergibt sich mit der folgenden Invariante:

Nach dem l-ten Lauf ist $T_l[W, v]$ korrekt für alle W mit $|W| \leq l$.

Laufzeit: $O(n^2 \cdot 2^n)$, da ein Lauf von *Setze(W, v)* in $O(n)$ möglich ist.

Das hier verwendete Prinzip: Mehr rekursive Aufrufe als Möglichkeiten ⇒ Tabellieren der Aufrufe heißt:

Dynamisches Programmieren (Auffüllen einer Tabelle, 1950er)