UNIVERSITÄT
LEIPZIG

Algorithmen und Datenstrukturen II

Vorlesung *sep7*

Leipzig, 14.05.2024
Peter F. Stadler & Thomas Gatter & Ronny Lorenz

DYNAMISCHE PROGRAMMIERUNG

Wozu Dynamische Programmierung?

- Dynamische Programmierung ist eine generische *Methode* in der mathematischen Optimierung.
- Generell ist die Aufgabe aus einer Menge von möglichen Lösungen eine optimale Lösung zu bestimmen.
- Im Prinzip können wir alle möglichen Lösungen enumerieren und die Beste auswählen. Allerdings sind die Lösungsräume häufig groß. Zum Beispiel exponentiell in Bezug auf die Eingabe.
- Im Folgenden werden wir sehen, das Dynamische Programmierung es uns erlaubt die optimale Lösung deutlich schneller als durch Enumeration zu finden.

Wikipedia *(Link)*

Prinzip Dynamische Programmierung (für Optimierung)

1. Charakterisiere den Lösungsraum.
2. Definiere rekursiv, wie eine optimale Lösung aus kleineren optimalen Lösungen (für Teilprobleme) zusammengesetzt wird.
3. Lege eine Berechnungsreihenfolge fest, so dass die Lösungen von kleineren Teilproblemen berechnet werden, bevor diese für die Lösung eines grösseren benötigt werden.

Voraussetzung für (2): *Bellmannsches Optimalitätsprinzip*
Die optimale Lösung eines (Teil-)Problems (der Grösse *n*) setzt sich aus optimalen Lösungen kleinerer Teilprobleme zusammen.

Activity Selection mit Gewichten

- Gegeben sei eine Menge A von $n = |A|$ *Aktivitäten* k, z.B. Lehrveranstaltungen mit vorgegebenen Anfangs- und Endzeiten a_k, e_k, und einem *Wert* v_k.
- Eine Menge B von Aktivitäten ist kompatibel wenn zwei Aktivitäten in B zeitlich nicht überlappen.
- Gesucht ist die Menge von kompatiblen Aktivitäten mit maximaler Kardinalität (maximalem Gesamtwert)

$$f(B) = \sum_{k \in B} v_k$$

Das Greedy-lösbare Activity Selection entspricht dem Spezialfall $v_k = 1$ für alle k.

Activity Selection: Sortierung

Zur Vereinfachung nehmen wir an, dass die Aktivitäten bereits nach den Endzeitpunkten sortiert sind, also

$$e_1 \leq e_2 \leq \cdots \leq e_n$$

Wenn nicht, sortieren wir sie einfach.

Definition

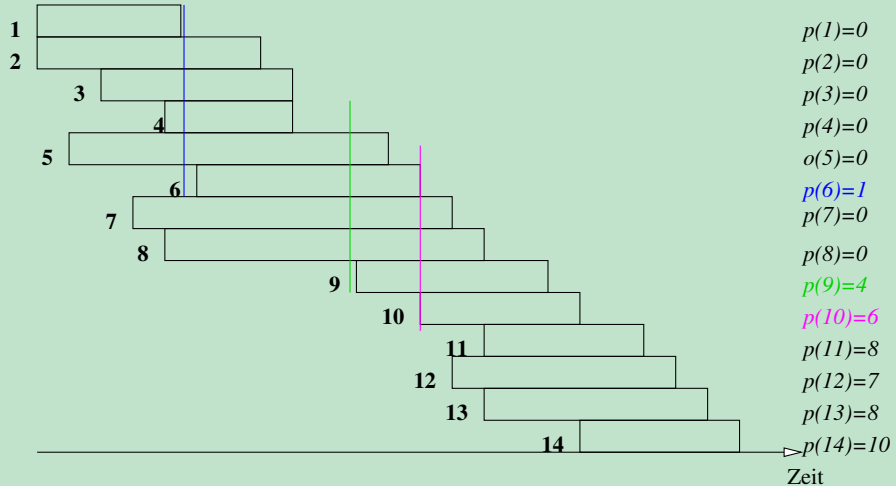
$p(k)$ ist der grösste Index einer Aktivität, die mit Aktivität k kompatibel ist. Also

$$p(k) := \max\{i | e_i \leq a_k\}$$

Beispiel

Aktivität	Startzeit	Endzeit	Gewicht v_k
1	1:00 Uhr	5:00 Uhr	2
2	1:00 Uhr	7:00 Uhr	1
3	3:00 Uhr	8:00 Uhr	3
4	4:30 Uhr	8:00 Uhr	4
5	2:00 Uhr	11:00 Uhr	2
6	6:00 Uhr	12:00 Uhr	1
7	4:00 Uhr	13:00 Uhr	2
8	4:30 Uhr	14:00 Uhr	6
9	10:00 Uhr	16:00 Uhr	3
10	12:00 Uhr	17:00 Uhr	4
11	14:00 Uhr	19:00 Uhr	1
12	13:00 Uhr	20:00 Uhr	2
13	14:00 Uhr	21:00 Uhr	3
14	17:00 Uhr	22:00 Uhr	3

Beispiel


1
2
3
4
5
6
7
8
9
10
11
12
13
14
p(1)=0
p(2)=0
p(3)=0
p(4)=0
o(5)=0
p(6)=1
p(7)=0
p(8)=0
p(9)=4
p(10)=6
p(11)=8
p(12)=7
p(13)=8
p(14)=10
Zeit

Beispiel

Aktivität	Startzeit	Endzeit	Gewicht v_k	$p(k)$
1	1:00 Uhr	5:00 Uhr	2	0
2	1:00 Uhr	7:00 Uhr	1	0
3	3:00 Uhr	8:00 Uhr	3	0
4	4:30 Uhr	8:00 Uhr	4	0
5	2:00 Uhr	11:00 Uhr	2	0
6	6:00 Uhr	12:00 Uhr	1	1
7	4:00 Uhr	13:00 Uhr	2	0
8	4:30 Uhr	14:00 Uhr	6	0
9	10:00 Uhr	16:00 Uhr	3	4
10	12:00 Uhr	17:00 Uhr	4	6
11	14:00 Uhr	19:00 Uhr	1	8
12	13:00 Uhr	20:00 Uhr	2	7
13	14:00 Uhr	21:00 Uhr	3	8
14	17:00 Uhr	22:00 Uhr	3	10

Activity Selection: Teillösungen I

Notation

$V[k] :=$ Wert der optimalen Lösung für das Teilproblem, das nur aus den ersten k Aktivitäten besteht.

1.Fall Die optimale Teillösung B enthält die Aktivität k.

$\rightarrow$ Dann müssen alle anderen Aktivitäten in B vor a_k geendet haben, d.h., die letzte möglicherweise inkludierte Teillösung war $p(k)$.
Die bestmögliche Lösung der Aktivitäten bis zu $p(k)$ hat – *nach Definition* – den Wert $V[p(k)]$. Die optimale Lösung mit letzter Aktivität k hat daher den Wert:
$V[k] = v_k + V[p(k)]$

2.Fall Die optimale Teillösung B enthält die Aktivität k *nicht*.

$\rightarrow$ Dann ist die beste Lösung diejenige die bis zu Aktivität $k - 1$ gefunden wurde.
$V[k] = V[k - 1]$

Activity Selection: Teillösungen II

Die insgesamt bei k beste Lösung ist daher die bessere der beiden Alternativen. Daher erfüllen die Werte $V[k]$ die **Rekursionsgleichung**.

$$V[k] = \max(v_k + V[p(k)], V[k-1])$$

Startbedingung: Die leere Menge von Aktivitäten ist mit allen anderen kompatibel und hat den Wert $V[0] = 0$.

Activity Selection: Teillösungen III

- **Vorprozessierung:** Sortiere A nach Endzeiten
- **Forwärts-Schritt:** Befülle das Array $A[k]$ schrittweise von $k = 0, 1, \ldots, n$ entsprechend der Rekursionsgleichung

$$V[0] = 0$$
$$V[k] = \max(v_k + V[p(k)], V[k-1])$$

  Der Wert der optimalen Lösung findet sich dann in $V[n]$.
- **Rückwärts-Schritt** zur Bestimmung der besten Lösung (*Backtracing*): beginne mit $k = n$ und einer leeren Lösungsmenge B; frage welche der beiden Alternativen erfüllt ist, und navigiere so durch das Array V.

Beispiel

Aktivität	Startzeit	Endzeit	Gewicht v_k	$p(k)$	$V(k)$	
1	1:00 Uhr	5:00 Uhr	2	0	2	(Fall 1)
2	1:00 Uhr	7:00 Uhr	1	0	2	(Fall 2)
3	3:00 Uhr	8:00 Uhr	3	0	3	(Fall 1)
4	4:30 Uhr	8:00 Uhr	4	0	4	(Fall 1)
5	2:00 Uhr	11:00 Uhr	2	0	4	(Fall 2)
6	6:00 Uhr	12:00 Uhr	1	1	4	(Fall 2)
7	4:00 Uhr	13:00 Uhr	2	0	4	(Fall 2)
8	4:30 Uhr	14:00 Uhr	6	0	6	(Fall 1)
9	10:00 Uhr	16:00 Uhr	3	4	7	(Fall 1)
10	12:00 Uhr	17:00 Uhr	4	6	8	(Fall 1)
11	14:00 Uhr	19:00 Uhr	1	8	8	(Fall 2)
12	13:00 Uhr	20:00 Uhr	2	7	8	(Fall 2)
13	14:00 Uhr	21:00 Uhr	3	8	9	(Fall 1)
14	17:00 Uhr	22:00 Uhr	3	10	11	(Fall 1)

Activity Selection: Backtracing I

– Beginne mit $k = n$ und einer leeren Lösungsmenge $B = \emptyset$.
Frage welche der beiden Alternativen erfüllt ist:

$$V[k] = v_k + V[p(k)] \qquad \text{oder} \qquad V[k] = V[k-1]$$

---

1. Wenn $V[k] = v_k + V[p(k)]$, dann ist k Teil der optimalen Lösung. Also setze

$$B \leftarrow B \cup \{k\}$$

Die Teillösung davor endet spätestens mit Aktivität $p(k)$, also setze

$$k \leftarrow p(k)$$

Activity Selection: Backtracing II

2. Wenn $V[k] = V[k-1]$, dann ist k nicht Teil der optimalen Lösung.
   Also bleibt B unverändert und wir setzen

$$k \leftarrow k - 1$$

– Das Backtracing terminiert sobald $k = 0$ erreicht wird. Dann steht in B **eine** kompatible Menge von Aktivitäten mit optimalem Wert $V[n]$.

Was tun wenn $V[k] = v_k + V[p(k)] = V[k-1]$ ist?
Wie erkenne ich im Backtracing ob die optimale Lösung eindeutig ist?

Beispiel

Aktivität	Startzeit	Endzeit	Gewicht v_k	$p(k)$	$V(k)$	
1	1:00 Uhr	5:00 Uhr	2	0	2	(Fall 1)
2	1:00 Uhr	7:00 Uhr	1	0	2	(Fall 2)
3	3:00 Uhr	8:00 Uhr	3	0	3	(Fall 1)
4	4:30 Uhr	8:00 Uhr	4	0	4	(Fall 1)
5	2:00 Uhr	11:00 Uhr	2	0	4	(Fall 2)
6	6:00 Uhr	12:00 Uhr	1	1	4	(Fall 2)
7	4:00 Uhr	13:00 Uhr	2	0	4	(Fall 2)
8	4:30 Uhr	14:00 Uhr	6	0	6	(Fall 1)
9	10:00 Uhr	16:00 Uhr	3	4	7	(Fall 1)
10	12:00 Uhr	17:00 Uhr	4	6	8	(Fall 1)
11	14:00 Uhr	19:00 Uhr	1	8	8	(Fall 2)
12	13:00 Uhr	20:00 Uhr	2	7	8	(Fall 2)
13	14:00 Uhr	21:00 Uhr	3	8	9	(Fall 1)
14	17:00 Uhr	22:00 Uhr	3	10	11	(Fall 1)

Die Menge von Aktivitäten 4, 10 und 14 ist optimal.

Das Alignment- bzw. String-Editing-Problem I

Wir beginnen mit der *Hamming-Distanz*:

Hier sind nur Substitutionen in der gleichen Spalte erlaubt.
Eine gleiche Ersetzung (z.B. A $\rightarrow$ A) gibt 0.
Eine ungleiche (z.B. A $\rightarrow$ B) gibt 1.
Ausserdem müssen die beiden Strings gleich lang sein.

Das Alignment- bzw. String-Editing-Problem II

Beispiel

A B C D E F
B C D E F G

Macht hier die Hamming-Distanz wirklich “Sinn”?
Die Kosten sind 6, aber eine kleine Verschiebung

A B C D E F -
- B C D E F G

liefert ein viel besseres Ergebnis: $1 + 0 + \cdots + 0 + 1 = +2$

Alignments von Strings I

Problem

Vergleiche 2 Strings *x* und *y* über dem selben Alphabet $\mathcal{A}$ mit drei erlaubten Operationen.

Editier-Operationen: Insertion, Deletion, und Substitution von Zeichen

Was ist die minimale Anzahl an Editier-Operationen, mit der *x* in *y* umgewandelt werden kann? (“Levenshtein-Distanz”)

Alignments von Strings II

Beispiel

L I P S I A $\rightarrow$ ***L E I P Z I G***

L	-	I	P	S	I	A
L	E	I	P	Z	I	G

ins(`E:-`), subst(`Z:S`), subst(`A:G`)

Allgemeine Eigenschaften von Editier-Distanzen I

Gegeben sei eine Menge X von *Objekten* und eine Menge Ω von *(elementaren) Operationen* $\omega : X \to X$.

Definition

Ein **Editier-Pfad** von x nach y ist eine Folge $x = x_0, x_1, x_2, \ldots, x_\ell = y$ sodass $x_{i+1} = \omega(x_i)$ für eine geeignete Operation $\omega \in \Omega$.

Allgemeine Eigenschaften von Editier-Distanzen II

Definition

Die **Editier-Distanz** $d(x, y)$ ist die minimale Zahl von Editier-Operationen, die nötig ist, um x in y umzuwandeln.

Es gilt:

- Die Konkatenation von Editierpfaden ist wieder ein Editier-Pfad.
- Wenn z auf einem *kürzesten* Editierpfad von x nach y liegt, dann gilt $d(x, y) = d(x, z) + d(z, y)$

Editier-Distanz

Beobachtungen zum String Editing:

- Jede optimale Folge von Editier-Operationen ändert jedes Zeichen höchstens einmal
- Jede Zerlegung einer optimalen Anordnung führt zur optimalen Anordnung der entsprechenden Teilsequenzen
- Konstruktion der optimalen Gesamtlösung durch rekursive Kombination von Lösungen für Teilprobleme

Dieses Problem kann durch ***Dynamische Programmierung*** gelöst werden.

Editier-Distanz I

Ansatz: Betrachte Editier-Distanzen von Zeichenketten.

- D_{ij} sei die Editierdistanz für die Zeichenketten $(a_1, ..., a_i)$ und $(b_1, ..., b_j)$; $0 \leq i \leq n$; $0 \leq j \leq m$.

Wie sieht ein zu D_{ij} gehöriges Alignment aus?

- *Letzte Position:* (a_i, b_j) oder $(a_i, -)$ oder $(-, b_j)$
- *Davor:* optimale Alignments der entsprechenden Zeichenketten.

Editier-Distanz II

Rekursion:

$$D_{ij} = \min \begin{cases} D_{i-1,j-1} + \begin{cases} 0 & \text{falls } a_i = b_j \\ 1 & \text{falls } a_i \neq b_j \end{cases} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

Triviale Lösungen für leere Präfixe:
$D_{0,0} = s(-,-) = 0; \quad D_{0,j} = j; \quad D_{i,0} = i$

Editier-Distanz - Algorithmus

Gegeben 2 Wörter (Zeichenketten)
$w_1 = a_1...a_n$ und $w_2 = b_1...b_m$

Algorithmus

1. erstelle Tabelle $D[0...n][0...m]$
2. $D[i][0] = i, 0 \leq i \leq n$
3. $D[0][j] = j, 0 \leq j \leq n$
4. **for** i=1 **to** n **do**
   for j=1 **to** m **do**
   $c \leftarrow$ **if** $a_i = b_j$ **then** 0 **else** 1
   $D[i][j] = min\{D[i-1][j-1] + c,$
   $D[i][j-1] + 1,$
   $D[i-1][j] + 1\}$

Editierdistanz

- Jeder Weg von links oben nach rechts unten entspricht einer Folge von Edit-Operationen, die *a* in *b* transformiert
- Möglicherweise mehrere Pfade mit minimalen Kosten
- Komplexität: $O(n \times m)$
- ist ein typisches Beispiel für Dynamische Programmierung (DP)

	j	0	1	2	3
i		-	R	A	D
0	-	0	1	2	3
1	A	1	1	1	2
2	U	2	2	2	2
3	T	3	3	3	3
4	O	4	4	4	4

Backtracing im Editierproblem I

Woher kommt die optimale Lösung $D_{4,3}$?

$$D_{4,3} = 4? = \begin{cases} D_{3,2} + s(O,D) = 3 + 1 = 4 \\ D_{4,2} + 1 = 4 + 1 = 5 \\ D_{3,3} + 1 = 3 + 1 = 4 \end{cases}$$

Wir wählen die erste passenden Gleichheit, und ignorieren die Alternative. Deren Existenz zeigt uns, dass es mehrere gleich-gute Lösungen gibt.

Backtracing im Editierproblem II

Woher kommt $D_{3,2} = 3$? ... $D_{3,2} = D_{2,1} + s(T, A) = 2 + 1$
Woher kommt $D_{2,1} = 2$? ... $D_{2,1} = D_{1,0} + s(U, R) = 1 + 1$
Woher kommt $D_{1,0} = 1$? ... $D_{1,0} = D_{0,0} + 1 = 0 + 1$

Das (langweilige) optimale Alignment ist:

A	U	T	O
-	R	A	D

Dies erhält man, in dem die Edit Schritte, für die man Gleichheit erhalten hat, von hinten nach vorne gelesen werden.
Der *optimale Eintrag* $D_{4,3}$ gehört zum Ende beider Strings.

Matrix-Produkte I

– **klassisches Beispiel für DP**: minimiere Rechenaufwand für die Multiplikation mehrerer Matrizen unterschiedlicher Dimension.

Beispiel

Hier ein Beispiel mit 6 Matrizen $A = a_ij, \ldots, F = f_ij$ welches wir in den nächsten Folien häufiger sehen werden

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11}b_{12}b_{13} \\ b_{21}b_{22}b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} (d_{11}d_{12}) \begin{pmatrix} e_{11}e_{12} \\ e_{21}e_{22} \end{pmatrix} \begin{pmatrix} f_{11}f_{12}f_{13} \\ f_{21}f_{22}f_{23} \end{pmatrix}$$

Matrixmultiplikation ist *assoziativ*, es gibt also verschiedene Berechnungswege.

Matrix-Produkte II

Zur Erinnerung: Für Matrixmultiplikation $A\,B$ muss die Anzahl der Spalten von A mit der Anzahl der Zeilen von B übereinstimmen, sie ist also nicht *kommutativ*!

Aufwand um eine $(p \times q)$ *mit einer* $(q \times r)$ *Matrix zu multiplizieren:*
$p \times r$ Einträge, deren jeder q skalare Multiplikationen und Additionen erfordert, also proportional zu pqr.

Matrix-Produkte III

Im Beispiel:

$$M_1 = AB \ldots 4 \times 2 \times 3 = 24$$
$$M_2 = M_1 C \ldots 4 \times 3 \times 1 = 12$$
$$M_3 = M_2 D \ldots 4 \times 1 \times 2 = 8$$
…**insgesamt 84**

Anders rum: $N_5 = EF$, $N_4 = DN_5$, $N_3 = CN_4$, …
nur 69 Skalarmultiplikationen!

Matrix-Produkte IV

→Klammern bestimmen Reihenfolge der Multiplikation:

- Reihenfolge von links nach rechts entspricht Ausdruck

  `( ( ( ( ( AB ) C ) D ) E ) F )`

- Reihenfolge von rechts nach links entspricht Ausdruck

  `( A ( B ( C ( D ( EF ) ) ) ) )`

- Jedes zulässige Setzen von Klammern führt zum richtigen Ergebnis

Matrix-Produkte V

Wann ist die Anzahl der Multiplikationen am kleinsten?

Erklärung

Wenn große Matrizen auftreten, können beträchtliche Einsparungen erzielt werden: Wenn z. B. die Matrizen B, C und F im obigen Beispiel eine Dimension von 300 statt von 3 besitzen, sind bei der Reihenfolge von links nach rechts 6024 Multiplikationen erforderlich, bei der Reihenfolge von rechts nach links dagegen ist die viel größere Zahl von 274 200 Multiplikationen auszuführen.

Matrix-Produkte VI

- **Allgemeiner Fall:** n Matrizen sind miteinander zu multiplizieren: $M_1 M_2 M_3 \cdots M_n$ wobei für jede Matrix M_i, $1 \leq i < n$, gilt: M_i hat r_i Zeilen und r_{i+1} Spalten.
- **Ziel:** Diejenige Reihenfolge der Multiplikation der Matrizen zu finden, für die die Gesamtzahl der auszuführenden Skalar-Multiplikationen minimal wird.
- **Lösung des Problems** mit Hilfe der dynamischen Programmierung besteht darin, “von unten nach oben” vorzugehen und berechnete Lösungen kleiner Teilprobleme zu speichern, um eine wiederholte Rechnung zu vermeiden.

Was ist der beste Weg das Teilprodukt $M_k M_{k+1} \ldots M_{l-1} M_l$ zu berechnen?

Entwicklung eines DP-Algorithmus: 2 und 3 Matrizen

- Berechnung der Kosten für Multiplikation benachbarter Matrizen und Speicherung in einer Tabelle.
- **Berechnung der Kosten für Multiplikation von 3 aufeinanderfolgenden Matrizen.** z. B. beste Möglichkeit, $M_1 M_2 M_3$ zu multiplizieren:
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_1 M_2$ und addiere die Kosten der Multiplikation dieses Ergebnisses mit M_3.
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_2 M_3$ und addiere die Kosten der Multiplikation dieses Ergebnisses mit M_1.
  - Vergleich der Ergebnisse und Abspeichern der besseren Variante.

Entwicklung eines DP-Algorithmus: 4 und viele Matrizen

- **Berechnung der Kosten für Multiplikation von 4 aufeinanderfolgenden Matrizen.** z. B. beste Möglichkeit, $M_1M_2M_3M_4$ zu multiplizieren:
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_1M_2M_3$ und addiere die Kosten der Multiplikation dieses Ergebnisses mit M_4.
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_2M_3M_4$ und addiere die Kosten der Multiplikation dieses Ergebnisses mit M_1.
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von M_1M_2 sowie von M_3M_4 und addiere dazu die Kosten der Multiplikation dieser beiden Ergebnisse.
  - Vergleich der Ergebnisse und Abspeicherung der besseren Variante.
- Indem man in dieser Weise fortfährt, findet man schließlich die beste Möglichkeit alle Matrizen miteinander zu multiplizieren.

Allgemeines Matrix-Produkt

Sei also C_{ij} der minimale Aufwand für die Berechnung des Teilproduktes $M_i M_{i+1} \dots M_{j-1} M_j$.

- $C_{ii} = 0 \rightarrow$Produkt besteht aus nur einem Faktor, es ist nichts zu tun.
- $M_k M_{k+1} \dots M_{i-1}$ ist eine $r_k \times r_i$ Matrix
  $M_i M_{i+1} \dots M_{l-1} M_l$ ist eine $r_i \times r_{l+1}$ Matrix

$\rightarrow$ Produkt dieser beiden Faktoren benötig $r_k r_i r_{l+1}$ Operationen.

- Die beste Zerlegung von $(M_k M_{k+1} \dots M_{l-1} M_l)$ ist diejenige, die die Summe $C_{k,i-1} + C_{i,l} + r_k r_i r_{l+1}$ minimiert.
- Daher Rekursion ($k < l$):

$$C_{kl} = \min_{i:\, k<i\leq l} (C_{k,i-1} + C_{i,l} + r_k r_i r_{l+1})$$

- Lösung des Gesamtproblems: C_{1n}.

Beispiel Matrix-Produkt

A: (4×2) B: (2×3) C: (3×1) D: (1×2) E: (2×2) F: (2×3)

	A	*B*	*C*	*D*	*E*	*F*
A	0	24 [A][B]	14 [A][BC]	22 [ABC][D]	26 [ABC][DE]	36 [ABC][DEF]
B		0	6 [B][C]	10 [BC][D]	14 [BC][DE]	22 [BC][DEF]
C			0	6 [C][D]	10 [C][DE]	19 [C][DEF]
D				0	4 [D][E]	10 [DE][F]
E					0	12 [E][F]
F						0

Eintrag Zeile B Spalte D, also C_{BD}:

$C_{BD} = \min\{C_{BB} + C_{CD} + r_B r_C r_E, C_{BC} + C_{DD} + r_B r_D r_E\} =$

$\min\{0 + 6 + 2 \cdot 3 \cdot 2, 6 + 0 + 2 \cdot 1 \cdot 2\} = \min\{6 + 12, 6 + 4\} = 10$

Beschreibung der Tabelle I

- Angabe der Gesamtkosten der Multiplikationen für jede Teilfolge in der Liste der Matrizen und der optimale “letzte” Multiplikation.
  - Z. B. besagt die Eintragung in der Zeile A und der Spalte F, dass 36 Skalar-Multiplikationen erforderlich sind, um die Matrizen A bis F zu multiplizieren. Dies wird erreicht, indem A bis C auf optimale Weise multipliziert werden, dann D bis F auf optimale Weise multipliziert werden und danach die erhaltenen Matrizen miteinander multipliziert werden.
- Für obiges Beispiel ist die ermittelte Anordnung der Klammern
  `( (A ( BC ) )( ( DE ) F )`
  wofür nur 36 Skalar-Multiplikationen benötigt werden.

Beschreibung der Tabelle II

- Für das Beispiel, in dem die Dimension von 3 auf 300 geändert wurde, ist die gleiche Anordnung der Klammern optimal, wobei hier 2412 Skalar-Multiplikationen erforderlich sind
- Mit Hilfe der dynamischen Programmierung kann das Problem der Multiplikation von n Matrizen mit Zeit in $\Theta(n^3)$ gelöst werden.

Matrix Multiplikation: Backtracing I

Woher kommt die optimale Lösung?

$$C_{AF} = 36$$

$$? = C_{AA} + C_{BF} + r_A r_B 3 = 0 + 22 + 4 \cdot 2 \cdot 3 = 46$$

$$? = C_{AB} + C_{CF} + r_A r_C 3 = 24 + 19 + 4 \cdot 3 \cdot 3 = 69$$

$$? = C_{AC} + C_{DF} + r_A r_D 3 = 14 + 10 + 4 \cdot 1 \cdot 3 = 36$$

$$? = C_{AD} + C_{EF} + r_A r_E 3 = 22 + 12 + 4 \cdot 2 \cdot 3 = 58$$

$$? = C_{AE} + C_{FF} + r_A r_F 3 = 26 + 0 + 4 \cdot 2 \cdot 3 = 50$$

Matrix Multiplikation: Backtracing II

- C_{AF} wurde aus C_{AC} und C_{DF} erzeugt

→ $(C_{AC})(C_{DF})$

Wie sind die Einträge C_{AC} und C_{DF} entstanden?

- C_{AC} aus C_{AA} (also A) und C_{BC} ... → $(A(C_{BC}))(C_{DF})$
- C_{DF} aus C_{DE} und C_{FF} (also F) ... → $(A(C_{BC}))((C_{DE})F)$
- C_{DF} aus C_{DE} und C_{FF} (also F) ... → $(A(C_{BC}))((C_{DE})F)$
- C_{BC} aus B und C ... → $(A(BC))((C_{DE})F)$
- C_{DE} aud D und E ... → $(A(BC))((DE)F)$

Matrix Multiplikation: Backtracing I

Systematisches Backtracing auf den Teillösungen:

→ Benutze einen Stack für die noch zu bearbeitenden Teillösungen.

- Bestimme die Teillösungen (Tabellen-Einträge), aus denen die optimale Lösung besteht.
- Wirf diese Teillösungen auf einen Stack
- Frage für das Top-Element des Stacks: was sind die Teillösungen aus denen die Teillösung enstanden ist, und wirf diese auf den Stack.
- Das Backtracing ist abgeschlossen wenn der Stack leer ist.

Matrix Multiplikation: Backtracing II

In unseren Fall:

- $\rightarrow C_{AF}$
- $\rightarrow (C_{AC})(C_{DF})$
- $\rightarrow (A(C_{BC}))(C_{DF})$
- $\rightarrow (A(BC))(C_{DF})$
- $\rightarrow (A(BC))((C_{DE})F)$
- $\rightarrow (A(BC))((DE)F)$

Was liegt in jedem Schritt auf dem Stack?