14 - Dynamische Programmierung (1)

Prof. Dr. S. Albers

Überblick

- Allgemeine Vorgehensweise, Unterschiede zur rekursiven Lösung
- Ein einfaches Beispiel: Die Berechnung der Fibonacci Zahlen

Technik der dynamischen Programmierung

Rekursiver Ansatz: Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt.

Phänomen: Mehrfachberechnungen von Lösungen

Methode: Speichern einmal berechneter Lösungen in einer Tabelle für spätere Zugriffe.

Beispiel: Fibonacci-Zahlen

$f(0) = 0$
$f(1) = 1$
$f(n) = f(n-1) + f(n-2)$, falls $n \geq 2$

Bem: Es gilt:

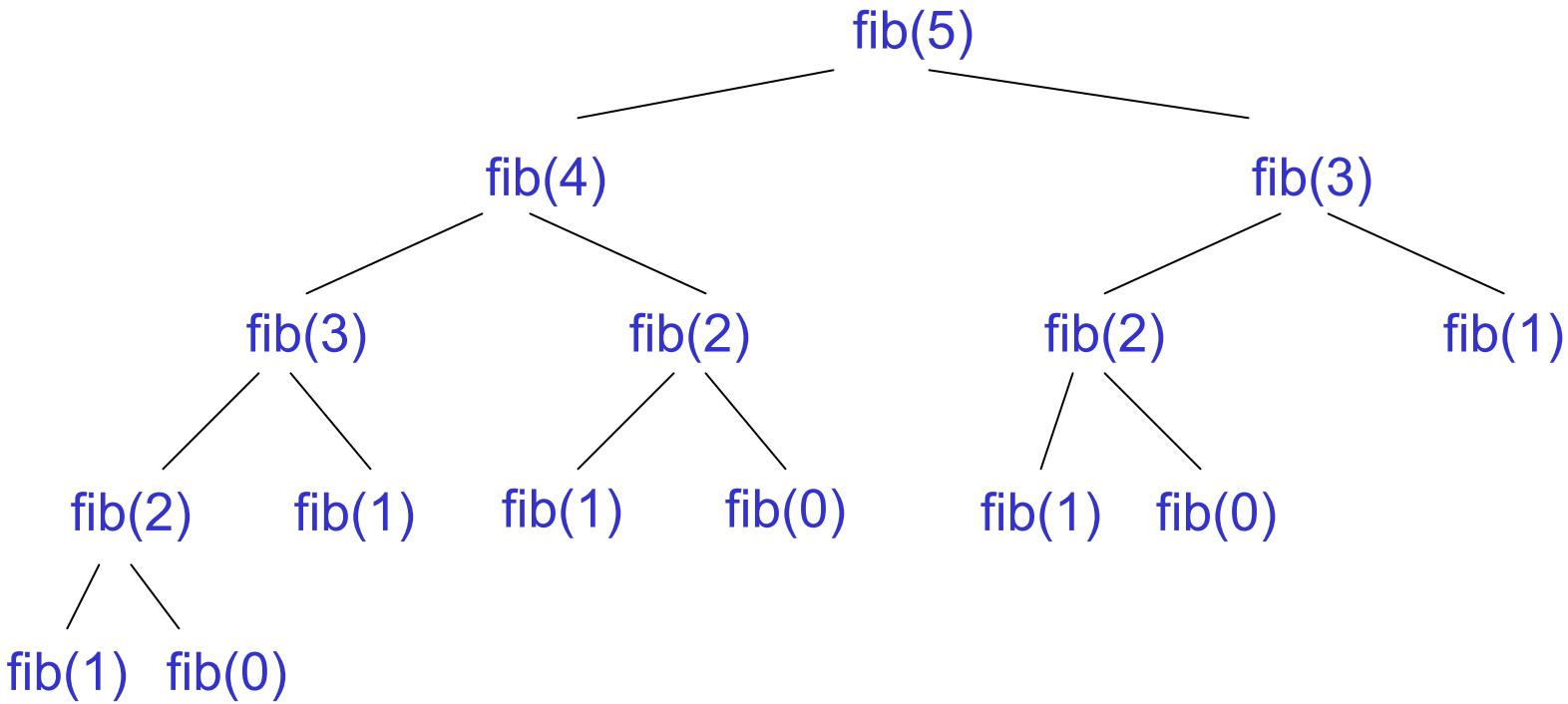
$$f(n) = \left[\frac{1}{\sqrt{5}} (1.618\ldots)^n \right]$$

Direkte Implementierung:

procedure *fib* (*n* : *integer*) : *integer*
if (*n* == 0) or (*n* ==1)
then return *n*
else return *fib*(*n* – 1) + *fib*(*n* – 2)

Beispiel: Fibonacci-Zahlen

Aufrufbaum:



Mehrfachberechnung!

$$T(n) \approx \left[\left(1 + \frac{1}{\sqrt{5}}\right) \left(\frac{\sqrt{5}+1}{2}\right)^n - 1 \right] \approx \left[1.447 \times 1.618^n - 1\right]$$

Dynamisches Programmieren

Vorgehen:

1. Rekursive Beschreibung des Problems P

2. Bestimmung einer Menge T, die alle Teilprobleme von P enthält, auf die bei der Lösung von P – auch in tieferen Rekursionsstufen – zurückgegriffen wird.

2. Bestimmung einer Reihenfolge $T_0, ..., T_k$ der Probleme in T, so dass bei der Lösung von T_i nur auf Probleme T_j mit $j < i$ zurückgegriffen wird.

4. Sukzessive Berechnung und Speicherung von Lösungen für $T_0,...,T_k$.

Beispiel Fibonacci-Zahlen

1. Rekursive Definition der Fibonacci-Zahlen nach gegebener Gleichung.

2. $T = \{ f(0),..., f(n\text{-}1)\}$

3. $T_i = f(i), \quad i = 0,...,n-1$

4. Berechnung von *fib*(*i*) benötigt von den früheren Problemen nur die zwei letzten Teillösungen *fib*(*i* – 1) und *fib*(*i* – 2) für i $\geq$ 2.

Beispiel Fibonacci-Zahlen

Berechnung mittels dynamischer Programmierung, Variante 1:

procedure *fib*(*n* : *integer*) : *integer*

1 $f_0 := 0; f_1 := 1$

2 **for** $k := 2$ **to** n **do**

3 $\quad\quad f_k := f_{k-1} + f_{k-2}$

4 **return** f_n

Beispiel Fibonacci-Zahlen

Berechnung mittels dynamischer Programmierung, Variante 2:

procedure *fib* (*n* : *integer*) : *integer*

1 $f_{vorletzte} := 0; f_{letzte} := 1$

2 **for** $k := 2$ **to** n **do**

3 $f_{aktuell} := f_{letzte} + f_{vorletzte}$

4 $f_{vorletzte} := f_{letzte}$

5 $f_{letzte} := f_{aktuell}$

6 **if** $n \leq 1$ **then return** n else r**eturn** $f_{aktuell}$;

Lineare Laufzeit, konstanter Platzbedarf!

Memoisierte Version der Berechnung der Fibonacci-Zahlen

Berechne jeden Wert genau einmal, speichere ihn in einem Array $F[1...n]$:

```
procedure fib (n : integer) : integer
1   F[0] := 0;   F[1] := 1;
2   for i :=2 to n do
3         F[i] := ∞;
4   return lookupfib(n)
```

Die Prozedur *lookupfib* ist dabei wie folgt definiert:

```
procedure lookupfib(k : integer) : integer
1   if F[k] < ∞
2       then return F[k]
3       else F[k] := lookupfib(k – 1) + lookupfib(k – 2);
4                 return F[k]
```

14 - Dynamische Programmierung (2)

Matrixkettenprodukt

Prof. Dr. S. Albers

Das Optimalitätsprinzip

Typische Anwendung für dynamisches Programmieren:
Optimierungsprobleme

Eine optimale Lösung für das Ausgangsproblem setzt sich aus *optimalen* Lösungen für kleinere Probleme zusammen.

Kettenprodukt von Matrizen

Gegeben: Folge (Kette) $\langle A_1, A_2, ..., A_n \rangle$ von Matrizen

Gesucht: Produkt $A_1 \cdot A_2 \cdot .... \cdot A_n$

Problem: Organisiere die Multiplikation so, dass möglichst wenig skalare Multiplikationen ausgeführt werden.

Definition: Ein Matrizenprodukt heißt *vollständig geklammert*, wenn es entweder eine einzelne Matrix oder das geklammerte Produkt zweier vollständig geklammerter Matrizenprodukte ist.

Beispiel für vollständig geklammerte Matrizenprodukte der Kette $\langle A_1, A_2, \ldots, A_n \rangle$

Alle vollständig geklammerten Matrizenprodukte der Kette $\langle A_1, A_2, A_3, A_4 \rangle$ sind:

$(A_1(A_2(A_3A_4)))$

$(A_1((A_2\,A_3)\,A_4))$

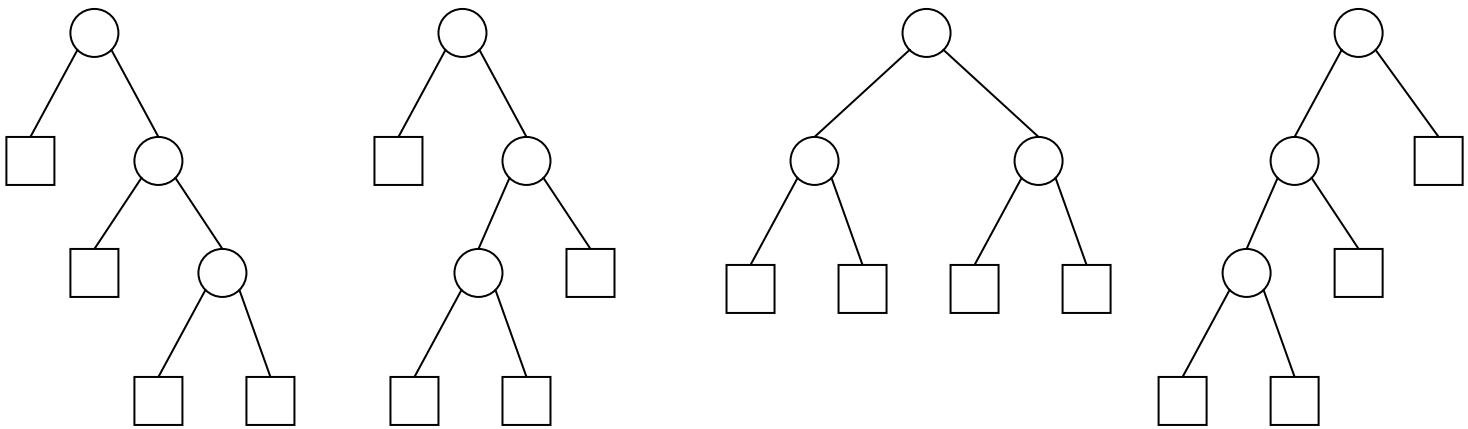
$((A_1\,A_2)(A_3\,A_4))$

$((A_1(A_2\,A_3))\,A_4)$

$(((A_1A_2)\,A_3)\,A_4)$

Anzahl der verschiedenen Klammerungen

Klammerungen entsprechen strukturell verschiedenen Bäumen.



Anzahl der verschiedenen Klammerungen

P(n) sei die Anzahl der verschiedenen Klammerungen von $A_1 \ldots A_k A_{k+1} \ldots A_n$

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \quad \text{für } n \geq 2$$

$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + \mathrm{O}\left(\frac{4^n}{\sqrt{n^5}}\right)$$

$$P(n+1) = C_n \quad n - \text{te Catalansche Zahl}$$

Bem: Finden der optimalen Klammerung durch Ausprobieren sinnlos.

Multiplikation zweier Matrizen

$$A = (a_{ij})_{p\times q}, B = (b_{ij})_{q\times r}, A \cdot B = C = (c_{ij})_{p\times r},$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}.$$

Algorithmus *Matrix-Mult*
Input: Eine ($p \times q$) Matrix A und eine ($q \times r$) Matrix B
Output: Die ($p \times$ r) Matrix C = A · B

```
1  for i := 1 to p do
2      for j :=1 to r do
3          C[i, j] := 0
4          for k := 1 to q do
5              C[i, j] := C[i, j] + A[i, k] ·B[k,j]
```

Anzahl Multiplikationen und Additionen: $p \cdot q \cdot r$

Bem: für zwei $n \times n$ – Matrizen werden hier n^3 Multiplikationen benötigt.
Es geht auch mit $O(n^{2.376})$ Multiplikationen.

Matrizenkettenprodukt Beispiel

Berechnung des Produkts von A_1, A_2, A_3 mit

A_1 :10 $\times$ 100 Matrix
A_2 :100 $\times$ 5 Matrix
A_3 : 5 $\times$ 50 Matrix

a) Klammerung $((A_1\, A_2)\, A_3)$ erfordert

$A' = (A_1\, A_2)$:

$A'\, A_3$:

Summe:

Matrizenkettenprodukt Beispiel

A_1 :10 $\times$ 100 Matrix
A_2 :100 $\times$ 5 Matrix
A_3 : 5 $\times$ 50 Matrix

a) Klammerung $(A_1\,(A_2\,A_3))$ erfordert

$A'' = (A_2\,A_3)$:
$A_1\,A''$:

Summe:

Struktur der optimalen Klammerung

$(A_{i...j}) = ((A_{i...k})\,(A_{k+1....j}))$ $i \le k < j$

Jede optimale Lösung des Matrixkettenprodukt Problems enthält optimale Lösungen von Teilproblemen.

Rekursive Bestimmung des Wertes einer optimalen Lösung:

m[i,j] sei minimale Anzahl von Operationen zur Berechung des Teilproduktes $A_{i...j}$:

m[i,j] = 0 falls i = j

m[i,j] = $\min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}$,sonst

s[i,k] = optimaler Splitwert für ein k, für das das Minimum angenommen wird

Rekursive Matrixkettenprodukt

Algorithmus *rek-mat-ket*(*p, i, j*)
Input: Eingabefolge $p = \langle p_0, p_1, \ldots, p_n \rangle$, $p_{i-1} \times p_i$ Dimensionen der Matrix A_i
Invariante: rek-mat-ket(*p, i, j*) liefert *m*[*i, j*]

```
1  if i = j then return 0
2  m[i, j] := ∞
3  for k := i to j – 1 do
4        m[i, j] := min(m[i,j], p_{i-1} p_k p_j +
                                  rek-mat-ket(p, i, k) +
                                  rek-mat-ket(p, k+1, j))
5  return m[i, j]
```

Aufruf: rek-mat-ket(*p*,1, *n*)

Rekursives Matrixkettenprodukt, Laufzeit

Sei $T(n)$ die Anzahl der Schritte zur Berechnung von rek-mat-ket(p,1,n).

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1 \right)$$

$$\geq n + 2 \sum_{i=1}^{n-1} T(i)$$

$$\Rightarrow T(n) \geq 3^{n-1} \quad \text{(vollst. Induktion)}$$

Exponentielle Laufzeit!

Matrixkettenprodukt mit dynamischer Programmierung

Algorithmus *dyn-mat-ket*

Input: Eingabefolge $p = \langle p_0, p_1, \ldots, p_n \rangle$ $p_{i-1} \times p_i$ Dimension der Matrix A_i

Output: $m[1,n]$

```
1  n:= length(p)
2  for i:= 1 to n do m[i, i] := 0
3  for l:=2 to n do
       /* l = Länge des Teilproblems */
4      for i := 1 to n – l + 1 do
           /* i ist der linke Index */
5          j := i + l - 1
           /* j ist der rechte Index*/
6          m[i, j] := ∞
7          for k := i to j - 1 do
8              m[i, j] := min(m[i, j], p_{i-1} p_k p_j + m[i, k] + m[k + 1, j])
9  return m[1, n]
```

Berechnungsbeispiel

A_1 30×35

A_2 35×15

A_3 15×5

A_4 5×10

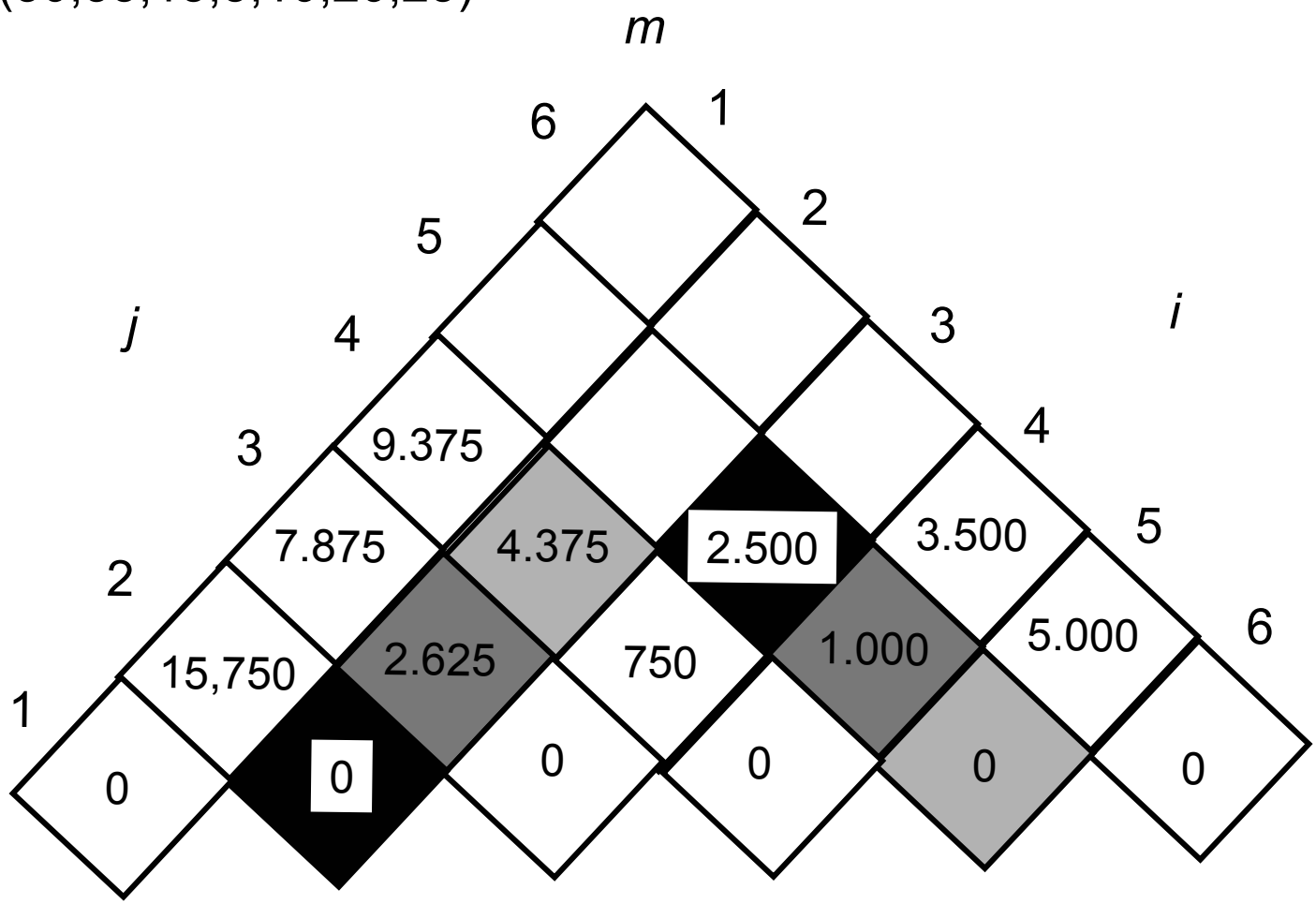
A_5 10×20

A_6 20×25

P = (30,35,15,5,10,20,25)

Berechnungsbeispiel

P = (30,35,15,5,10,20,25)



Berechnungsbeispiel

$$m[2,5] = \min_{2 \le k < 5} \left(m[2,k] + m[k+1,5] + p_1 p_k p_5 \right)$$

$$\min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 \end{cases}$$

$$\min \begin{cases} 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

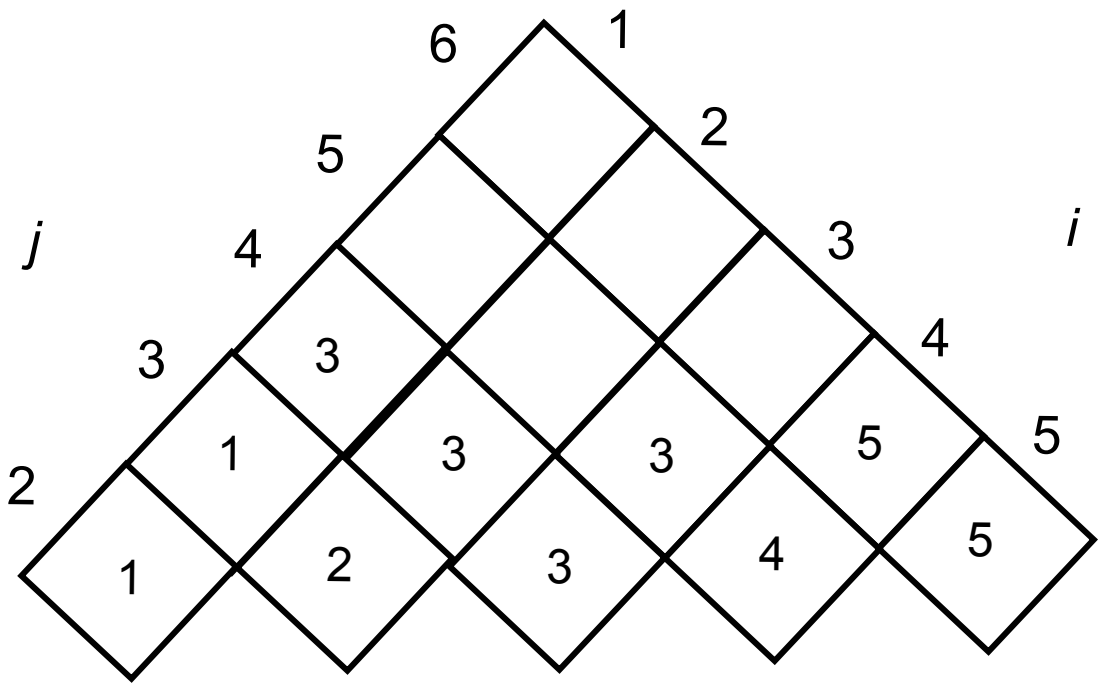
$$= 7125$$

Matrixkettenprodukt und optimale Splitwerte mit dynamischer Programmierung

Algorithmus *dyn-mat-ket(p)*
Input: Eingabefolge $p = \langle p_0, p_1, \ldots, p_n \rangle$ $p_{i-1} \times p_i$ Dimension der Matrix A_i
Output: $m[1,n]$ und eine Matrix $s[i,j]$ von opt. Splitwerten

```
1  n := length(p)
2  for i := 1 to n do m[i, i] := 0
3  for l := 2 to n do
4      for i := 1 to n – l +1 do
5          j := i + l – 1
6          m[i, j] := ∞
7      for k := i to j – 1 do
8          q := m[i, j]
9          m[i, j] := min(m[i, j], p_{i-1} p_k p_j +
                                   m[i, k] + m[k + 1, j])
10         if m[i, j] < q then s[i, j] := k
11 return (m[1, n], s)
```

Berechnungsbeispiel für Splitwerte



Berechnung der optimalen Klammerung

Algorithmus *Opt-Klam*

Input: Die Sequenz *A* der Matrizen, die Matrix *s* der optimalen Splitwerte, zwei Indizes *i* und *j*

Output: Eine optimale Klammerung von $A_{i...j}$

```
1   if i < j
2      then X := Opt-Klam(A, s, i, s[i, j])
3           Y := Opt-Klam(A, s, s[i, j] + 1, j)
4           return (X·Y)
5      else return A_i
```

Aufruf: *Opt–Klam*(*A*, *s*, 1, *n*)

Matrixkettenprodukt mit dynamischer Programmierung (Top-down Ansatz)

Notizblock-Methode zur Beschleunigung einer rekursiven Problemlösung:

Ein Teilproblem wird nur beim *ersten Auftreten* gelöst, die Lösung wird in einer Tabelle gespeichert und bei jedem späteren Auftreten desselben Teilproblems wird die Lösung (ohne erneute Rechnung!) in der Lösungstabelle nachgesehen.

Memoisiertes Matrixkettenprodukt (Notizblock-Methode)

Algorithmus *mem-mat-ket*(*p, i, j*)

Invariante: mem-mat-ket(*p, i, j*) liefert $m[i, j]$ und $m[i, j]$ hat den korrekten Wert, falls $m[i, j] < \infty$

```
1  if i = j then return 0
2  if m[i, j] < ∞ then return m[i, j]
3  for k := i to j – 1 do
4        m[i, j] := min(m[i, j], p_{i-1} p_k p_j +
                        mem-mat-ket(p, i, k) +
                        mem-mat-ket(p, k + 1, j))
5  return m[i, j]
```

Memoisiertes Matrixkettenprodukt (Notizblock-Methode)

Aufruf:

```
1  n:= length(p) – 1
2  for i := 1 to n do
3      for j := 1 to n do
4          m[i, j] := ∞
5  mem-mat-ket(p,1,n)
```

Zur Berechnung aller Einträge $m[i, j]$ mit Hilfe von mem-mat-ket genügen insgesamt $O(n^3)$ Schritte.

$O(n^2)$ Einträge
jeder Eintrag $m[i, j]$ wird einmal eingetragen
jeder Eintrag $m[i, j]$ wird zur Berechnung eines Eintrages $m[i´,j´]$ betrachtet, falls $i´ = i$ und $j´ > j$ oder $j´ = j$ und $i´ < i$
→ $\leq 2n$ Einträge benötigen $m[i, j]$

Bemerkung zum Matrixkettenprodukt

1. Es gibt eine Algorithmus mit linearer Laufzeit $O(n)$, der eine Klammerung findet mit Multiplikationsaufwand $\leq 1.155\ M_{opt}$
2. Es gibt einen Algorithmus mit Laufzeit $O(n \log n)$, der optimale Klammerung findet.

16 - Dynamische Programmierung (3)

Konstruktion optimaler Suchbäume

Prof. Dr. S. Albers

Technik der dynamischen Programmierung

Rekursiver Ansatz: Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt.

Phänomen: Mehrfachberechnungen von Lösungen

Methode: Speichern einmal berechneter Lösungen in einer Tabelle für spätere Zugriffe.

Das Optimalitätsprinzip

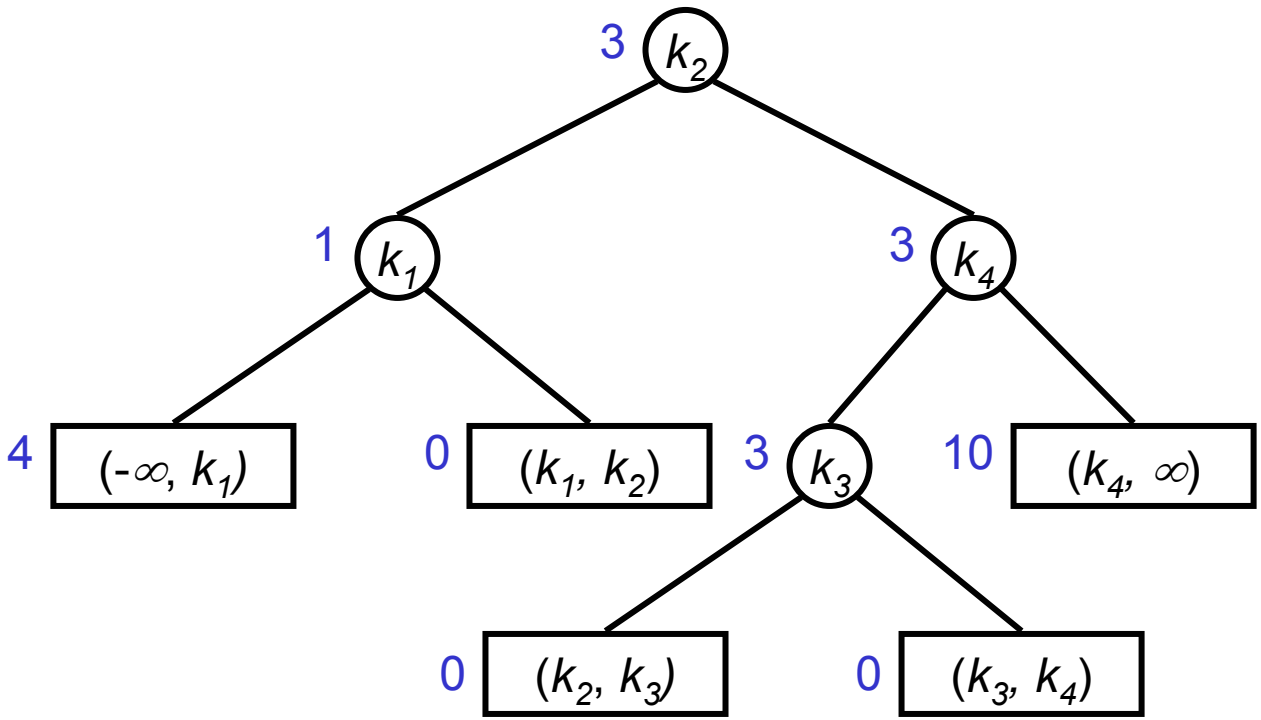
Typische Anwendung für dynamisches Programmieren: *Optimierungsprobleme*

Eine optimale Lösung für das Ausgangsproblem setzt sich aus *optimalen* Lösungen für kleinere Probleme zusammen.

Konstruktion optimaler Suchbäume

$(-\infty,k_1)$ k_1 (k_1,k_2) k_2 (k_2,k_3) k_3 (k_3,k_4) k_4 (k_4, ∞)

4 1 0 3 0 3 0 3 10



Gewichtete Pfadlänge:

$3 \cdot 1 + 2 \cdot (1 + 3) + 3 \cdot 3 + 2 \cdot (4 + 10)$

Konstruktion optimaler Suchbäume

Gegeben: Schlüsselmenge *S*

$$S = \{k_1,....k_n\} \quad -\infty = k_0 < k_1 \leq ... \leq k_n < k_{n+1} = \infty$$

a_i: (absolute) Häufigkeit, mit der nach k_i gesucht wird

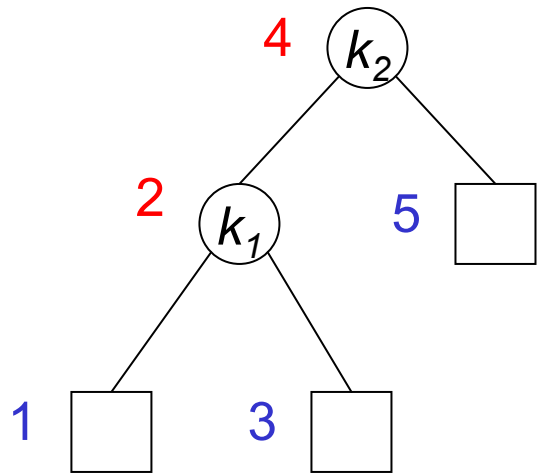
b_j: (absolute) Häufigkeit , mit der nach $x \in (k_j, k_{j+1})$ gesucht wird

Gewichtete Pfadlänge *P*(T) eines Suchbaumes T für S:

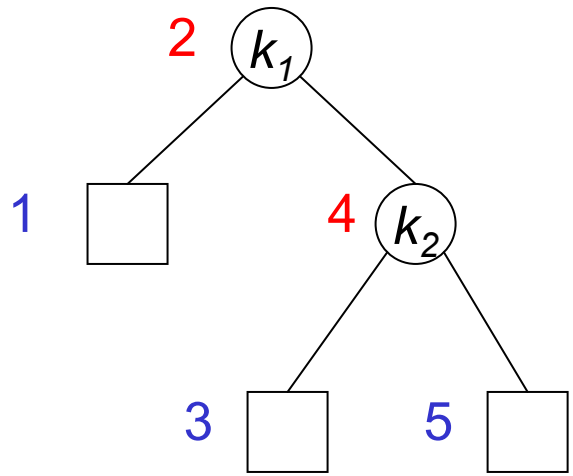
$$P(T) = \sum_{i=1}^{n} (Tiefe(k_i) + 1)a_i + \sum_{j=0}^{n} Tiefe((k_j, k_{j+1}))b_j$$

Gesucht: Ein Suchbaum für *S* mit minimaler Pfadlänge *P*

Konstruktion optimaler Suchbäume

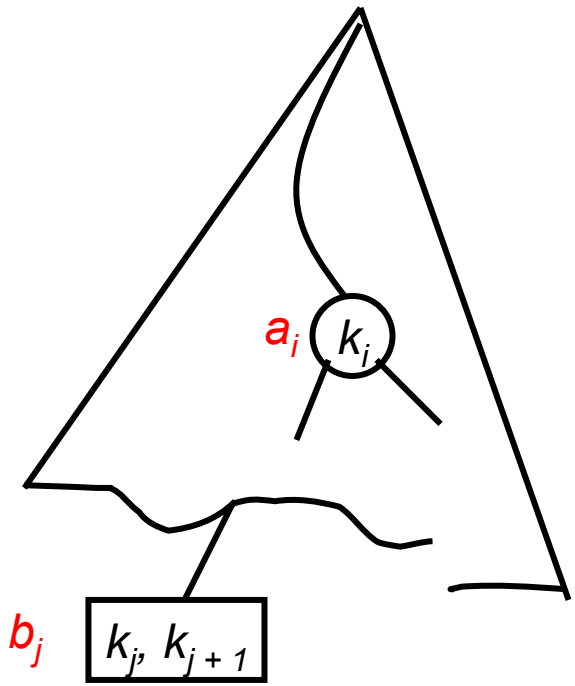


$P(T_1) = 21$



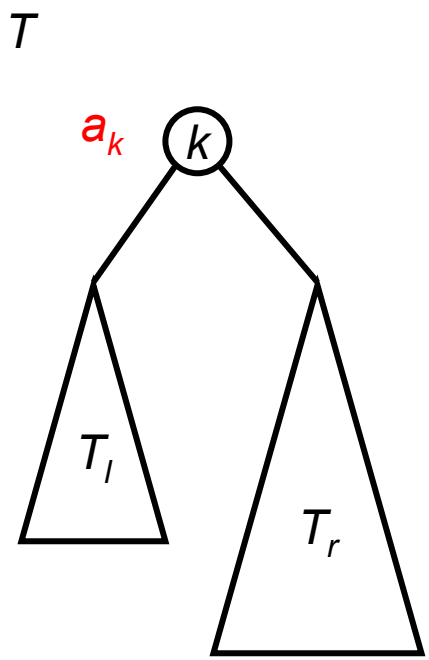
$P(T_2) = 37$

Konstruktion optimaler Suchbäume



Ein Suchbaum mit minimal möglicher gewichteter Pfadlänge ist ein optimaler Suchbaum

Konstruktion optimaler Suchbäume



$P(T) = P(T_l) + G(T_l) + P(T_r) + G(T_r) + a_{Wurzel}$

$= P(T_l) + P(T_r) + G(T)$ *mit*

$G(T)$:= Gesamtgewicht der Knoten von T

Ist T Baum mit minimaler gewichteter Pfadlänge, so auch T_l und T_r

Konstruktion optimaler Suchbäume

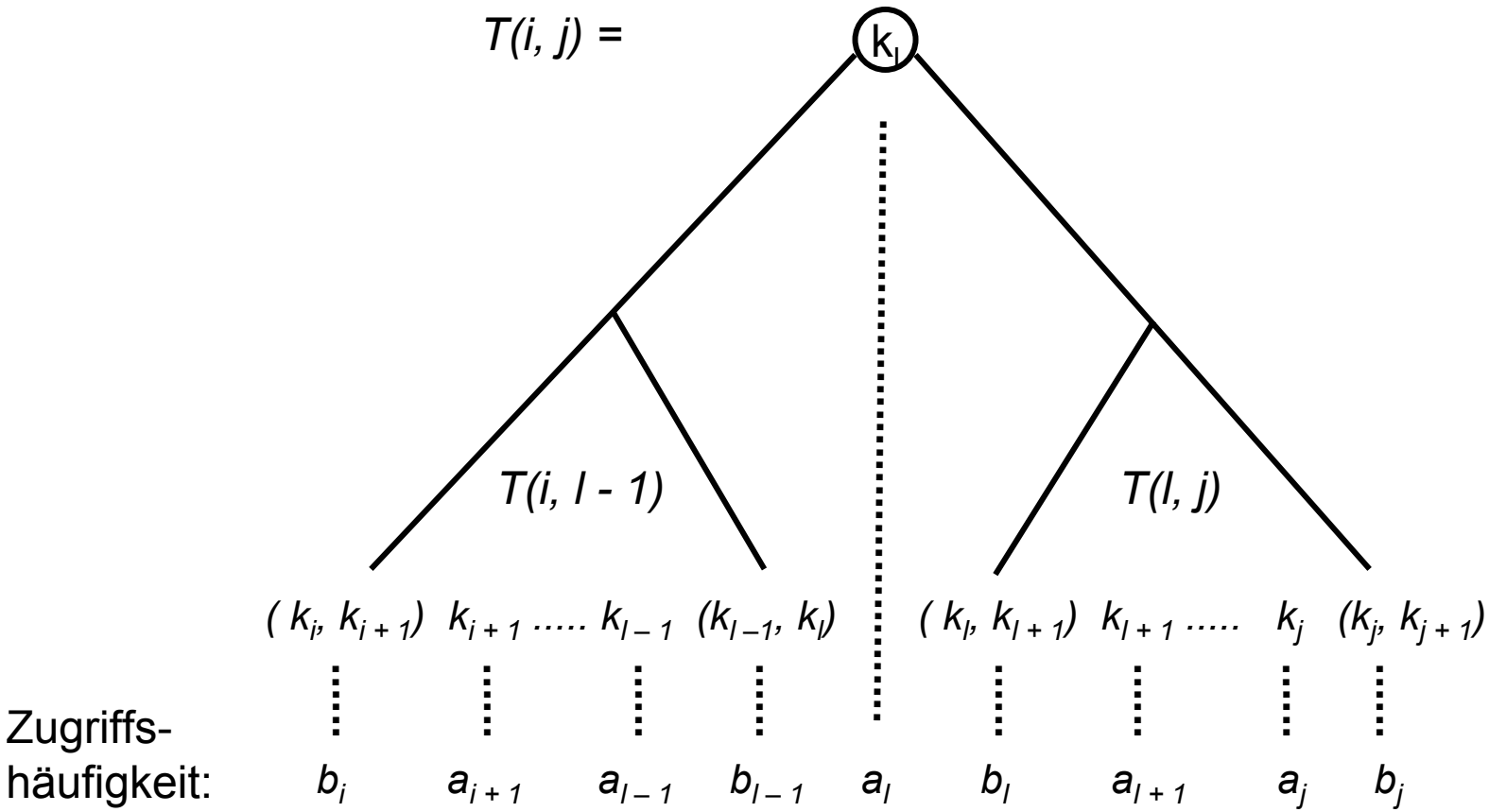
Sei

T(i, j) optimaler Suchbaum für $(k_i, k_{i+1})\ k_{i+1} \ldots\ k_j\ (k_j, k_{j+1})$,

W(i, j) das Gewicht von T(i, j), also $W(i, j) = b_i + a_{i+1} + \ldots + a_j + b_j$,

P(i, j) die gewichtete Pfadlänge von T(i, j).

Konstruktion optimaler Suchbäume



Konstruktion optimaler Suchbäume

$W(i, i) = b_i$, für $0 \leq i \leq n$

$W(i, j) = W(i, j-1) + a_j + b_j$, für $0 \leq i < j \leq n$

$P(i, i) = 0$, für $0 \leq i \leq n$

$P(i, j) = W(i, j) + \min_{i < l \leq j} \{ P(i, l-1) + P(l, j) \}$, für $0 \leq i < j \leq n$

r(i, j) = diejenige Zahl, für die das Minimum angenommen wird

Konstruktion optimaler Suchbäume

Anfangsfälle

Fall 1: $h = j - i = 0$

$T(i, i) = (k_i, k_{i+1})$

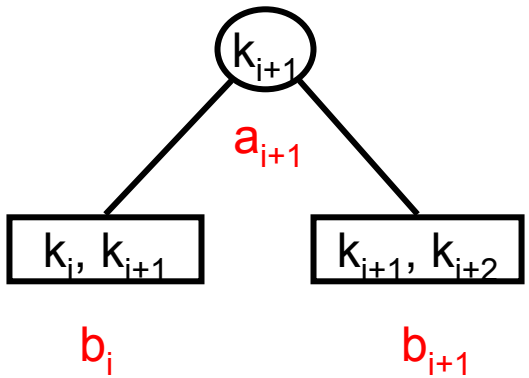
$W(i, i) = b_i$

$P(i, i) = 0$, $r(i, i)$ undefiniert

Konstruktion optimaler Suchbäume

Fall 2: h = j – i = 1

T(i, i+1)



$W(i, i+1) = b_i + a_{i+1} + b_{i+1} = W(i, i) + a_{i+1} + W(i+1, i+1)$

$P(i, i+1) = W(i, i+1)$

$r(i, i+1) = i + 1$

Berechnung der optimalen Pfadlänge mit dynamischer Programmierung

Fall 3: h = j - i > 1

for h = 2 **to** n **do**

  for i = 0 **to** (n – h) **do**

    { j = i + h;

      finde das (größte) l, i < l ≤ j, für das P(i, l – 1) + P(l, j) minimal wird;

      P(i, j) = P(i, l –1) + P(l, j) + W(i, j);

      r(i, j) = l;

    }

Konstruktion optimaler Suchbäume

Definiere:

$$\left.\begin{aligned} P(i,j) &:= \text{opt. Pfadlänge für} \\ G(i,j) &:= \text{Summe von} \end{aligned}\right\} b_i a_{i+1} b_{i+1} \ldots a_j b_j$$

Dann ist

$$G(i,j) = \begin{cases} b_i & \text{falls } i = j \\ G(i,j-1) + a_j + G(j,j) & \text{sonst} \end{cases}$$

$$P(i,j) = \begin{cases} 0 & \text{falls } i = j \\ G(i,j) + \min\limits_{i<l\le j}\{P(i,l-1) + P(l,i)\} & \text{sonst} \end{cases}$$

→ Die gesuchte Lösung $P(0,n)$ kann in Zeit $O(n^3)$ und Platz $O(n^2)$ berechnet werden.

Konstruktion optimaler Suchbäume

Satz

Ein optimaler Suchbaum für n Schlüssel und $n + 1$ Intervalle mit gegebenen Zugriffshäufigkeiten kann in Zeit $O(n^3)$ konstruiert werden.

17 - Dynamische Programmierung (4)

Editierdistanz
Approximative Zeichenkettensuche
Sequence Alignment

Prof. Dr. S. Albers

Dynamisches Programmieren

- Algorithmenentwurfstechnik, oft bei Optimierungsproblemen angewandt
- Allgemein einsetzbar bei rekursiven Problemlöseverfahren, wenn Teillösungen mehrfach benötigt werden
- Lösungsansatz: Tabellieren von Teilergebnissen
- Vorteil: Laufzeitverbesserungen, oft polynomiell statt exponentiell

Zwei verschiedene Ansätze

Bottom-up:

+ kontrollierte effiziente Tabellenverwaltung, spart Zeit
+ spezielle optimierte Berechnungsreihenfolge, spart Platz
- weitgehende Umcodierung des Originalprogramms erforderlich
- möglicherweise Berechnung nicht benötigter Werte

Top-down: (Memoisierung, Notizblockmethode)

+ Originalprogramm wird nur gering oder nicht verändert
+ Nur tatsächlich benötigte Werte werden berechnet
- separate Tabellenverwaltung benötigt Zeit
- Tabellengröße oft nicht optimal

Probleme der Ähnlichkeiten von Zeichenketten

Editier-Distanz

Berechne für zwei gegebene Zeichenfolgen *A* und *B* möglichst effizient die Editier-Distanz *D(A,B)* und eine minimale Folge von Editieroperationen, die *A* in B überführt.

i	n	f	-	-	-	o	r	m	a	t	i	k	-
i	n	t	e	r	p	o	l	-	a	t	i	o	n

Probleme der Ähnlichkeiten von Zeichenketten

Approximative Zeichenkettensuche

Finde für einen gegebenen Text *T*, ein Muster *P* und eine Distanz *d* alle Teilketten *P´* in *T* mit $D(P,P') \leq d$

Sequence Alignment

Finde optimale Alignments zwischen DNA-Sequencen

G A G C A - C T T G G A T T C T C G G
- - - C A C G T G G - - - - - - - - -

Editier-Distanz

Gegeben: Zwei Zeichenketten A = $a_1 a_2 .... a_m$ und $B = b_1 b_2 ... b_n$

Gesucht: Minimale Kosten $D(A,B)$ für eine Folge von Editieroperationen, um A in B zu überführen.

Editieroperationen:

1. Ersetzen eines Zeichens von *A* durch ein Zeichen von *B*
2. Löschen eines Zeichens von *A*
3. Einfügen eines Zeichens von *B*

Editier-Distanz

Einheitskostenmodell:

$$c(a,b) = \begin{cases} 1 & \text{falls } a \neq b \\ 0 & \text{falls } a = b \end{cases}$$

$a = \varepsilon,\ b = \varepsilon$ möglich

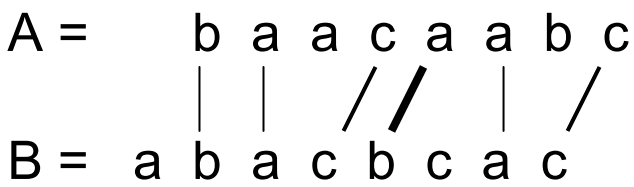
Dreiecksungleichung soll für *c* im allgemeinen gelten:

$$c(a,c) \leq c(a,b) + c(b,c)$$

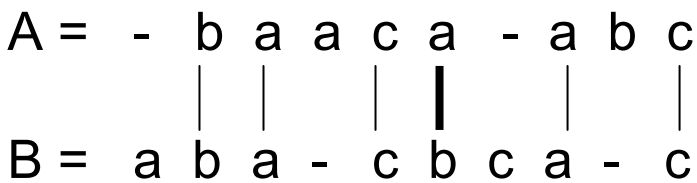
→ Ein Buchstabe wird höchstens einmal verändert

Editier-Distanz

Spur als Repräsentation von Editiersequenzen



oder mit Indents



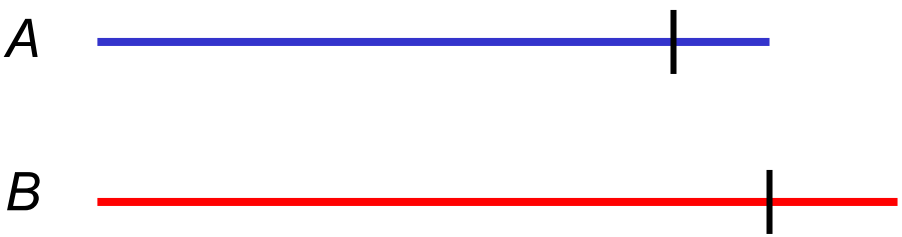
Editier-Distanz (Kosten) : 5

Aufteilung einer optimalen Spur ergibt zwei optimale Teilspuren
→ Dynamische Programmierung anwendbar

Berechnung der Editier-Distanz

Sei $A_i = a_1 \ldots a_i$ und $B_j = b_1 \ldots b_j$

$$D_{i,j} = D(A_i, B_j)$$



Berechnung der Editier-Distanz

Drei Möglichkeiten eine Spur zu beenden:

1. a_m wird durch b_n ersetzt:
   $D_{m,n} = D_{m-1,n-1} + c(a_m, b_n)$

2. a_m wird gelöscht: $D_{m,n} = D_{m-1,n} + 1$

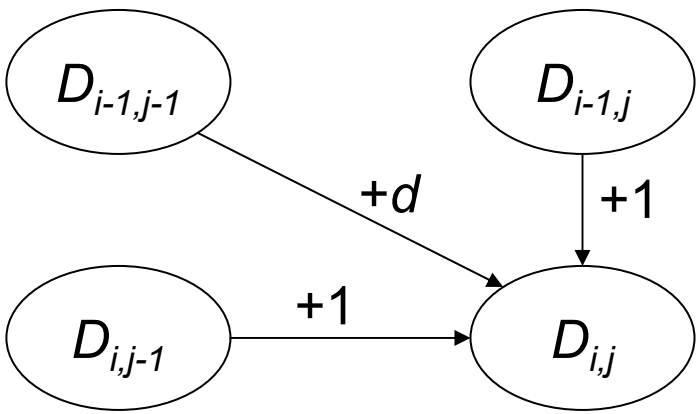
3. b_n wird eingefügt: $D_{m,n} = D_{m,n-1} + 1$

Berechnung der Editier-Distanz

Rekursionsgleichung, falls $m,n \geq 1$:

$$D_{m,n} = \min \begin{Bmatrix} D_{m-1,n-1} & + & c(a_m, b_n), \\ D_{m-1,n} & + & 1, \\ D_{m,n-1} & + & 1 \end{Bmatrix}$$

→ Berechnung aller $D_{i,j}$ erforderlich, $0 \leq i \leq m$, $0 \leq j \leq n$.



Rekursiongleichung für die Editier-Distanz

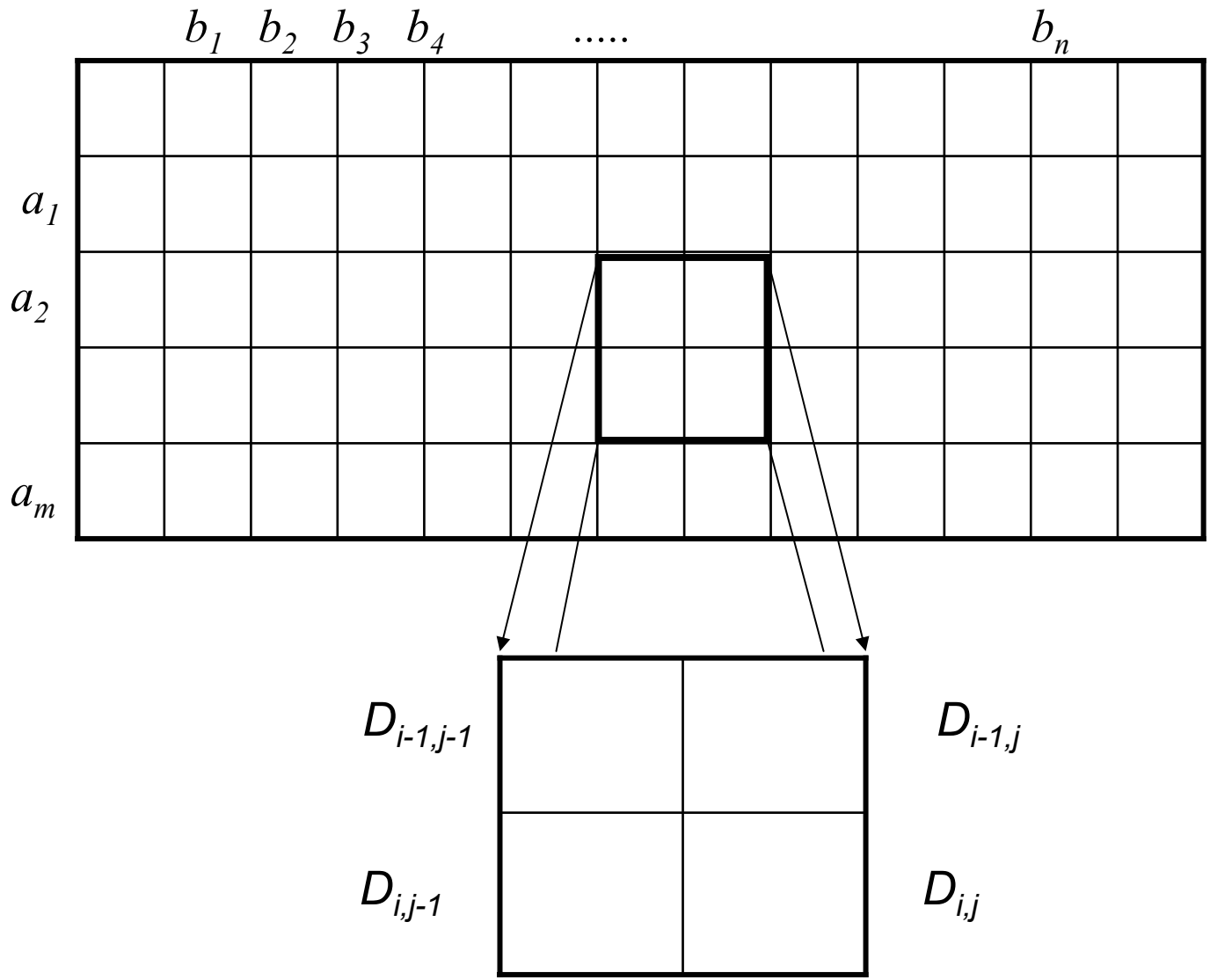
Anfangsbedingungen:

$$D_{0,0} = D(\varepsilon, \varepsilon) = 0$$
$$D_{0,j} = D(\varepsilon, B_j) = j$$
$$D_{i,0} = D(A_i, \varepsilon) = i$$

Rekursionsgleichung:

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} + c(a_i, b_j) \\ D_{i-1,j} + 1, \\ D_{i,j-1} + 1 \end{cases}$$

Berechnungsreihenfolge für die Editier-Distanz



Algorithmus für die Editier-Distanz

Algorithmus Editierdistanz

Input: Zwei Zeichenketten $A = a_1 .... a_m$ und $B = b_1 ... b_n$

Output: Matrix $D = (D_{ij})$

```
1 D[0,0] := 0
2 for i := 1 to m do D[i,0] = i
3 for j := 1 to n do D[0,j] = j
4 for i := 1 to m do
5    for j := 1 to n do
6          D[i,j] := min( D[i - 1,j] + 1,
7                         D[i,j - 1] + 1,
8                         D[i –1, j – 1] + c(a_i,b_j))
```

Beispiel für die Editier-Distanz

		a	b	a	c
	0	1	2	3	4
b	1				
a	2				
a	3				
c	4				

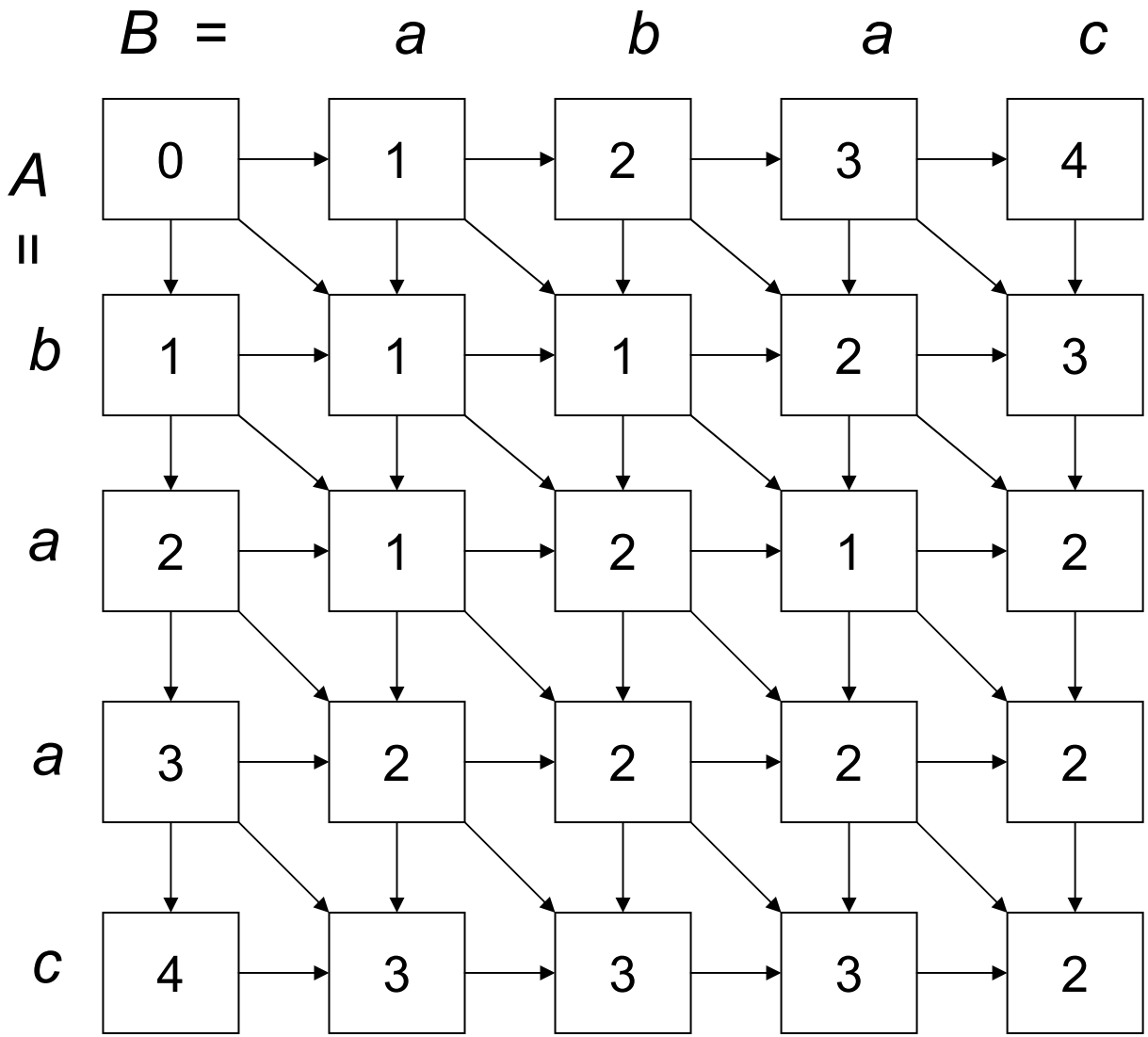
Berechnung der Editieroperation

Algorithmus Editieroperationen (*i,j*)
Input: Berechnet Matrix *D*

```
1   if i = 0 and j = 0 then return
2   if i ≠ 0 and D[i,j] = D[i – 1 , j] + 1
3     then „lösche a[i]“
4             Editieroperationen (i – 1, j)
5   else if j ≠ 0 and D[i,j] = D[i, j – 1] + 1
6      then „füge b[j] ein“
7             Editieroperationen (i,j – 1)
8   else
      /* D[i,j] = D[i – 1, j – 1 ] + c(a[i], b[j])  */
9           „ersetze a[i] durch b[j] “
10          Editieroperationen (i – 1, j – 1)
```

Aufruf: Editieroperationen(*m,n*)

Spurgraph der Editieroperationen



Subgraph der Editieroperationen

Spurgraph: Übersicht über alle möglichen Spuren zur Transformation von A in B, gerichtete Kanten von Knoten (i, j) zu $(i + 1, j)$, $(i, j + 1)$ und $(i + 1, j + 1)$.
Gewichtung der Kanten entsprechen den Editierkosten.

Kosten nehmen entlang eines optimalen Weges monoton zu.

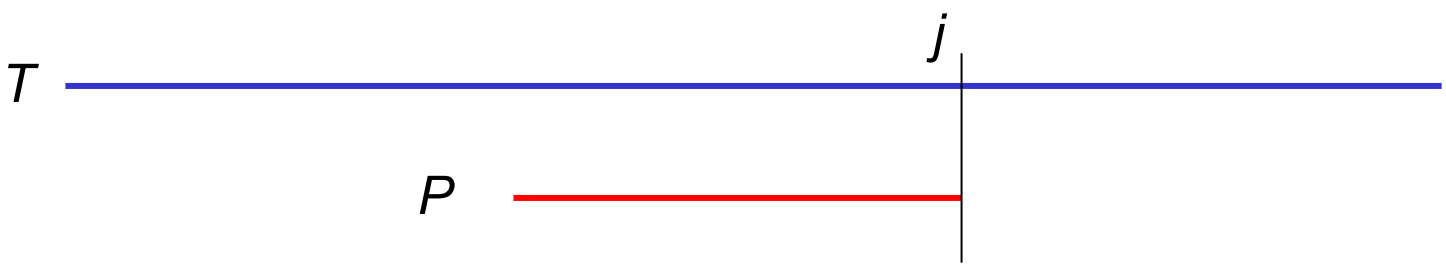
Jeder Weg mit monoton wachsenden Kosten von der linken oberen Ecke zu rechten unteren Ecke entspricht einer optimalen Spur.

Approximative Zeichenkettensuche

Gegeben: Zeichenketten $P = p_1 p_2 \ldots p_m$ (Muster) und
$T = t_1 t_2 \ldots t_n$ (Text)

Gesucht: Ein Intervall $[j', j]$, $1 \leq j' \leq j \leq n$, so dass das Teilwort
$T_{j',j} = t_{j'} \ldots t_j$ das dem Muster P
ähnlichste Teilwort von T ist, d.h. für alle anderen
Intervalle $[k', k]$, $1 \leq k' \leq k \leq n$, gilt:

$$D(P, T_{j',j}) \leq D(P, T_{k',k})$$



Approximative Zeichenkettensuche

Naives Verfahren:

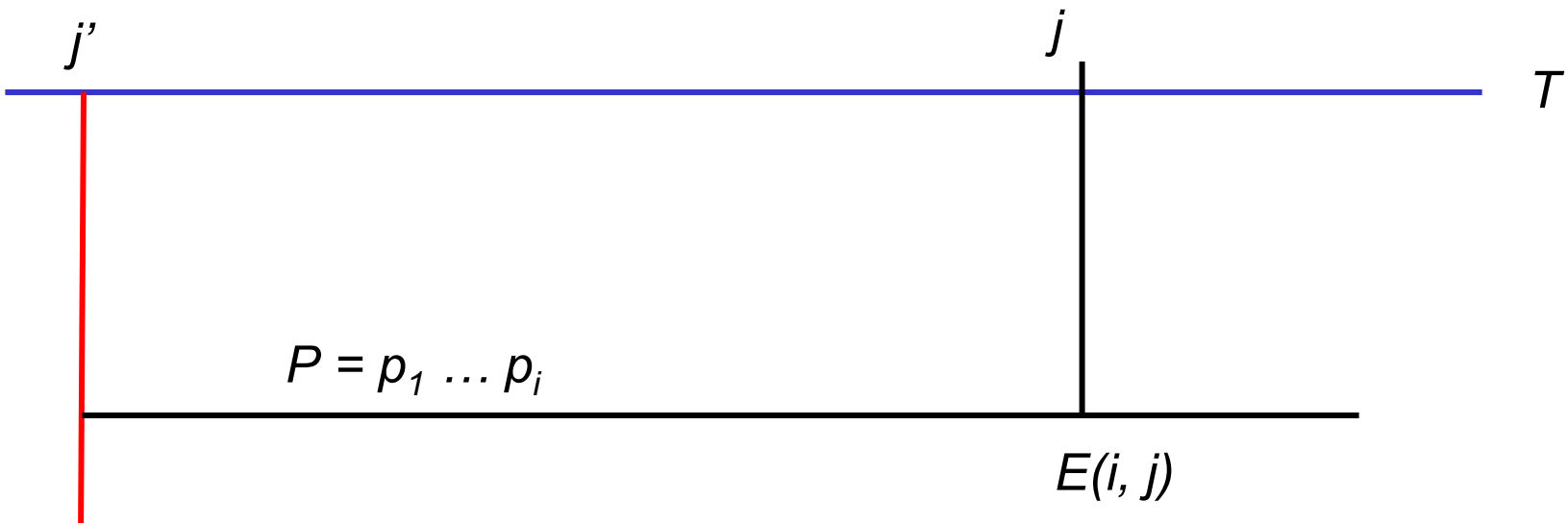
for all $1 \leq j' \leq j \leq n$ **do**

Berechne $D(P, T_{j',j})$

wähle Minimum

Approximative Zeichenkettensuche

Betrachte verwandtes Problem:



Für jede Textstelle *j* und jede Musterstelle *i* berechne die Editierdistanz des zu P_i ähnlichsten, bei *j* endenden Teilstücks $T_{j',j}$ von *T*.

Approximative Zeichenkettensuche

Methode:

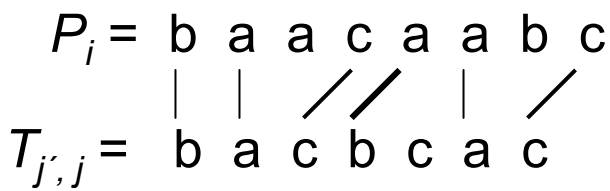
for all $1 \leq j \leq n$ ***do***

Berechne j', so dass $D(P, T_{j',j})$ minimal ist

Für $1 \leq i \leq m$ und $0 \leq j \leq n$ *sei:*

$$E_{i,j} = \min_{1 \leq j' \leq' j+1} D(P_i, T_{j',j})$$

Optimale Spur:



Approximative Zeichenkettensuche

Rekursionsgleichung:

$$E_{i,j} = \min\begin{Bmatrix} E_{i-1,j-1} + c(p_i, t_j), \\ E_{i-1,j} + 1, \\ E_{i,j-1} + 1 \end{Bmatrix}$$

Bemerkung:

j' kann für $E_{i-1,\,j-1}$, $E_{i-1,j}$ und $E_{i,\,j-1}$ ganz verschieden sein.

Teilspur einer optimalen Spur ist eine optimale Teilspur.

Approximative Zeichenkettensuche

Anfangsbedingungen:

$$E_{0,0} = E(\varepsilon, \varepsilon) = 0$$
$$E_{i,0} = E(P_i, \varepsilon) = i$$

aber

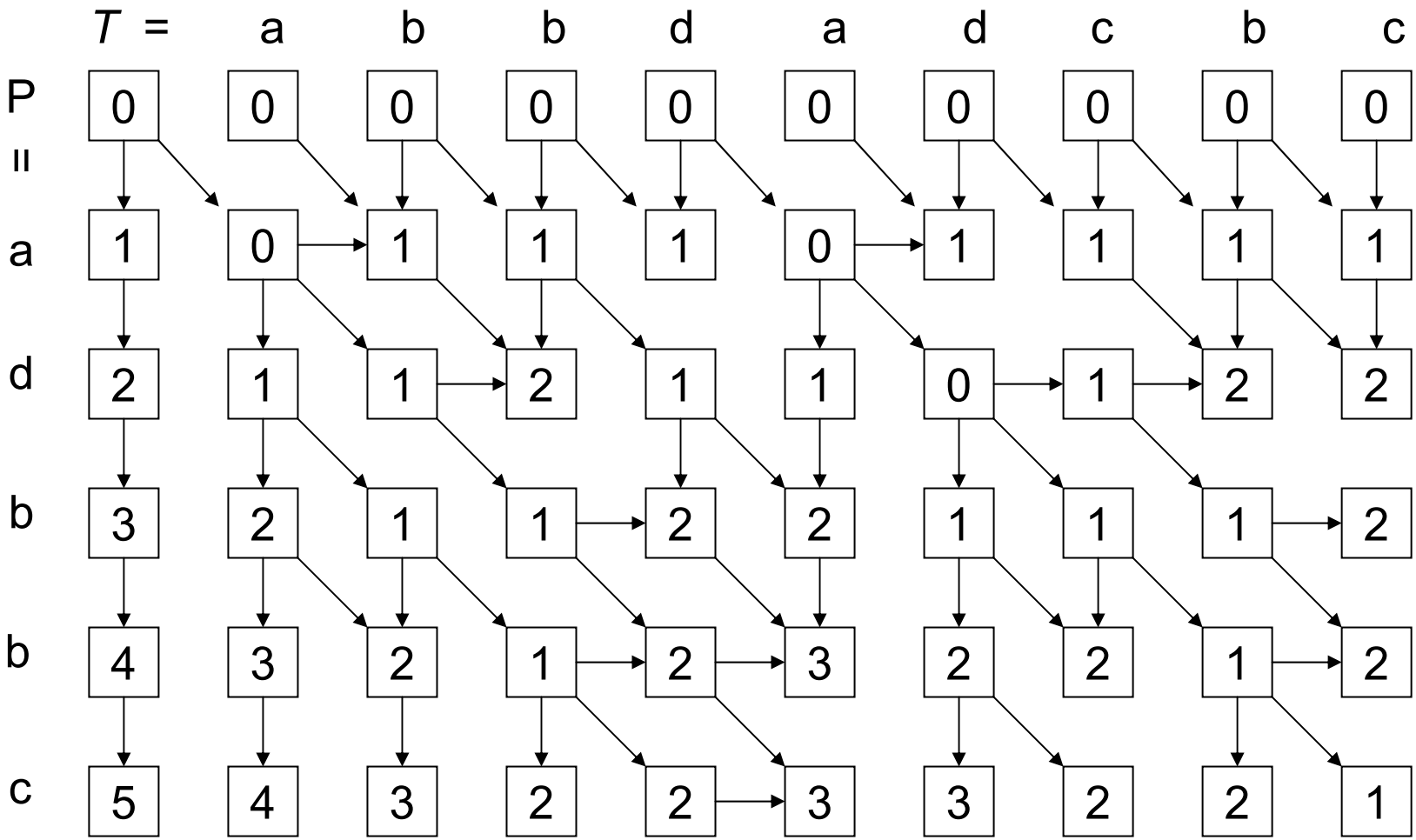
$$E_{0,j} = E(\varepsilon, T_j) = 0$$

Beobachtung:

Die optimale Editiersequenz von P nach $T_{j',j}$ beginnt nicht mit einer Einfügung von $t_{j'}$.

Approximative Zeichenkettensuche

Abhängigkeitsgraph



Approximative Zeichenkettensuche

Satz

Gibt es im Abhängigkeitsgraphen einen Weg von $E_{0, j'-1}$ nach $E_{i, j}$, so ist $T_{j', j}$ ein zu P_i ähnlichstes, bei j endendes Teilstück von T mit

$$D(P_i, T_{j',j}) = E_{i, j}$$

Ähnlichkeit von Zeichenketten

Sequence Aligment:

Finde für zwei DNA – Sequenzen Einfügestellen von Leerzeichen, so dass die Sequenzen möglichst ähnlich sind

G A - C G G A T T A G
G A T C G G A A T A G

Ähnlichkeit von Zeichenketten

Ähnlichkeitsmaß für Zeichenpaare:

Bsp.	Situation	allgemein
+ 1	für Match	} *s*(a,b)
- 1	für Mismatch	
- 2	FürLeerzeichen	- c

Ähnlichkeitsmaß für Sequencen:

$$S(A,B) = \sum_{alleZeichenpaare} \text{Ähnlichkeit des Zeichenpaars}$$

Ziel: Finde Alignment mit optimaler Ähnlichkeit

Ähnlichkeit von Zeichenketten

Ähnlichkeiten zweier Zeichenketten *S(A,B)*

Operationen:

1. Ersetzen eines Zeichens *a* durch ein Zeichen *b* :
   Gewinn: *s*(*a,b*)

2. Löschen eines Zeichens von *A*, Einfügen eines Zeichens von *B*
   Verlust: – *c*

Aufgabe:
Finde eine Folge von Operationen zur Unwandlung von *A* in *B*,
so dass die Summe der Gewinne maximiert wird.

Ähnlichkeit von Zeichenketten

$S_{i,j} = S(A_i, B_j)$, $0 \leq i \leq m$, $0 \leq j \leq n$

Rekursionsgleichung:

$$S_{m,n} = \max\ (S_{m-1,n-1} + s(a_m, b_n),\ S_{m-1,n} - c,\ S_{m,n-1} - c)$$

Anfangsbedingung:

$$S_{0,0} = S(\varepsilon, \varepsilon) = 0$$
$$S_{0,j} = S(\varepsilon, B_j) = -jc$$
$$S_{i,0} = S(A_i, \varepsilon) = -ic$$

Ähnlichste Teilzeichenketten

Gegeben: Zwei Zeichenketten $A = a_1 .... a_m$ und $B = b_1 ... b_n$

Gesucht: Zwei Intervalle $[i', i] \subseteq [1, m]$ und $[j', j] \subseteq [1, n]$,
so dass:

$$S(A_{i',i} , B_{j',j}) \geq S(A_{k',k} , B_{l',l}),$$

für alle $[k',k] \subseteq [1, m]$ und $[l',l] \subseteq [1, n]$.

Naives Verfahren:

for all $[i', i] \subseteq [1, m]$ **and** $[j', j] \subseteq [1, n]$ **do**
Berechne $S(A_{i',i} , B_{j',j})$

Ähnlichste Teilzeichenketten

Methode:

for all $1 \le i \le m$, $1 \le j \le n$ **do**

Berechne i' und j', so dass S($A_{i',i}$, $B_{j',j}$) maximal ist

Für $0 \le i \le m$ und $0 \le j \le n$ sei:

$$H_{i,j} = \max_{\substack{1 \le i' \le i+1, \\ 1 \le j' \le j+1}} S(A_{i',i}, B_{j',j})$$

Optimale Spur:

$A_{i',i}$ = b a a c a - a b c

| | | | | | |

$B_{j',j}$ = b a - c b c a - c

Ähnlichste Teilzeichenketten

Rekursionsgleichung:

$$H_{i,j} = \max\begin{Bmatrix} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} - c, \\ H_{i,j-1} - c, \\ 0 \end{Bmatrix}$$

Anfangsbedingung:

$$H_{0,0} = H(\varepsilon, \varepsilon) = 0$$
$$H_{i,0} = H(A_i, \varepsilon) = 0$$
$$H_{0,j} = H(\varepsilon, B_j) = 0$$