

ADS: Algorithmen und Datenstrukturen 2

Teil 8

Prof. Peter F. Stadler & Sebastian Will

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

28. Mai 2014

[Letzte Aktualisierung: 18/06/2014, 18:08]

Beobachtungen:

- 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

Dynamische Programmierung

Ansatz: Betrachte Editier-Distanzen von Präfixen.

- D_{ij} sei die Editierdistanz für die Präfixe (a_1, \dots, a_i) und (b_1, \dots, 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 Präfixe.

Rekursion:

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

Triviale Lösungen für leere Präfixe:

$$D_{0,0} = s(-, -) = 0; \quad D_{0,j} = \sum_{j'=1}^j s(-, b_{j'}); \quad D_{i,0} = \sum_{i'=1}^i s(a_{i'}, -)$$

	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

- 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: $O(n \times m)$
- = typisches Beispiel für Dynamische Programmierung (DP)

Dynamische Programmierung (DP)

- 1 Lösen eines Problem durch Zurückführen auf einfachere Teilprobleme, die gelöst und zur Gesamtlösung zusammengefügt werden.
- 2 In DP sind diese Teilprobleme i.A. *“überlappend”*, d.h. sie können ihrerseits gemeinsame Teilprobleme haben.
- 3 Um mehrfache Berechnung von Teilproblemlösungen einzusparen, werden die Teilergebnisse abgespeichert. (\Rightarrow Effizienz)

Anmerkungen:

- Zusammensetzung aus Teillösungen in (1) funktioniert nur für bei geeigneten Problemen und geeigneter Zerlegung, nicht allgemein!
- Divide-and-Conquer ist ein weiteres algorithmisches Prinzip, auf das (1) zutrifft. Im Gegensatz zu DP, wird bei D&C i.A. in *nicht-überlappende* Teilprobleme zerlegt, die *unabhängig* gelöst werden.

Prinzip Dynamische Programmierung (für Optimierung)

- 1 Charakterisiere den Lösungsraum.
- 2 Definiere rekursiv, wie eine optimale Lösung aus kleineren optimalen Lösungen 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.

Wann funktioniert DP?

- 1 Es muss nicht immer möglich sein, die Lösungen kleinerer Probleme so zu kombinieren, dass sich die Lösung eines größeren Problems ergibt.
- 2 Die Anzahl der zu lösenden Probleme kann unverträglich groß sein.
- 3 Es ist noch nicht gelungen, genau anzugeben, welche Probleme mit Hilfe der dynamischen Programmierung in effizienter Weise gelöst werden können. Es gibt viele “schwierige” Probleme, für die sie nicht anwendbar zu sein scheint, aber auch viele “leichte” Probleme, für die sie weniger effizient ist als Standardalgorithmen.

Systematische Theorie wurde in den letzten Jahren vor allem von Robert Giegerich in Bielefeld entwickelt (ADP = Algebraische DP):

- Zerlegung des Problems ... Grammatik
- Zerlegung der Kostenfunktion ... Algebra
- Optimierung ... Auswahlregel

- **klassisches Beispiel für DP:** minimiere Rechenaufwand für die Multiplikation mehrerer Matrizen unterschiedlicher Dimension, z.B.

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

- Matrixmultiplikation ist *assoziativ*, es gibt also verschiedene Berechnungswege.
- **Zur Erinnerung:** Für Matrixmultiplikation AB muss die Anzahl der Spalten von A mit der Anzahl der Zeilen von B übereinstimmen!

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 .

Im Beispiel:

$$M_1 = AB \dots 4 \times 2 \times 3 = 24$$

$$M_2 = M_1 C \dots 4 \times 3 \times 1 = 12$$

$$M_3 = M_2 D \dots 4 \times 1 \times 2 = 8$$

... insgesamt 84

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

Es gibt noch viele weitere Möglichkeiten:

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
- Wann ist die Anzahl der Multiplikationen am kleinsten?
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.

- Allgemeiner Fall: n Matrizen sind miteinander zu multiplizieren: $M_1 M_2 M_3 \dots M_n$ 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.
- Die 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 $M_k M_{k+1} \dots M_{l-1} M_l$ zu berechnen?

- 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.

- 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.

$$(M_k M_{k+1} \dots M_{l-1} M_l) = (M_k M_{k+1} \dots M_{i-1})(M_i M_{i+1} \dots M_{l-1} M_l)$$

- Sei also C_{ij} der minimale Aufwand für die Berechnung des Teilproduktes $M_i M_{i+1} \dots M_{j-1} M_j$.
- $C_{ii} = 0$, 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
- Produkt dieser beiden Faktoren benötigt daher $r_k r_i r_{l+1}$ Operationen.
- Die beste Zerlegung von $(M_k M_{k+1} \dots M_{l-1} M_l)$ ist daher 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} .

Matrix Multiplikation

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

Mit Hilfe der dynamischen Programmierung kann das Problem der Multiplikation mehrerer Matrizen in einer zu n^3 proportionalen Zeit und mit einem zu n^2 proportionalen Speicheraufwand gelöst werden.

Traveling Salesman Problem (TSP)

- **Gegeben:** n Städte, paarweise Distanzen:
 d_{ij} Entfernung von Stadt i nach Stadt j .
- **Gesucht:** Rundreise mit minimaler Länge durch alle Städte, also Permutation $\pi : (1, \dots, n) \rightarrow (1, \dots, n)$, für die

$$c(\pi) = \left[\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} \right] + d_{\pi(n)\pi(1)}$$

minimal.

- NP-hart (d.h. exponentiell, ausser $P=NP$)
- Naiver Algorithmus: Alle $(n - 1)!$ Reihenfolgen betrachten.

Traveling Salesman Problem (TSP)

Dynamische Programmierung für TSP

- Sei $g(i, S)$ Länge des kürzesten Weges von Stadt i über jede Stadt in der Menge S (jeweils genau *ein* Besuch) nach Stadt 1.
- Lösung des TSP also: $g(1, \{2, \dots, n\})$.
Anmerkung: Stadt 1 kann beliebig gewählt werden, da Rundreise gesucht wird.
- Es gilt:

$$g(i, S) = \begin{cases} d_{i1} & \text{falls } S = \emptyset \\ \min_{j \in S} [d_{ij} + g(j, S \setminus \{j\})] & \text{sonst} \end{cases}$$

Traveling Salesman Problem (TSP)

```
for ( i = 2 ; i <= n; i++) g[i,{}] = d[i,1]
for ( k = 1; k <= n-2 ; k++ )
  for (S, |S| = k, 1 not in S )
    for (i in {2,...,n}-S )
      Berechne g[i,S] gemäß Formel
Berechne g[1,{ 2,...,n }] gemäß Formel
```

Komplexität: Tabellengröße \times Aufwand je Tabelleneintrag

Größe: $< n2^n$ (Anzahl der i 's mal Anzahl betrachtete Teilmengen)

Tabelleneintrag: Suche nach Minimum unter j aus S : $O(n)$

Insgesamt: $O(n^2 \times 2^n)$, deutlich besser als $(n-1)!$.

Warum Dynamische Programmierung nützlich ist

- An einer völlig hypothetischen Universität ist es üblich, dass in Klausuren viel mehr Aufgaben gestellt werden als in der vorgegebenen Zeit t^* gelöst werden können.
- Ein ebenfalls völlig hypothetischer Student möchte nun seinen Punktestand optimieren.
- Welche Aufgaben sollten bearbeitet werden?
- Für jede Aufgabe $i \in \{1, \dots, n\}$ kennen wir die Bearbeitungszeit t_i und die dafür erreichbare Punktezahl p_i . Die Aufgabe besteht also darin, eine Teilmenge $S \subseteq \{1, \dots, n\}$ zu finden, so dass

$$\sum_{i \in S} p_i \rightarrow \max \quad \text{während} \quad \sum_{i \in S} t_i \leq t^*$$

- Üblicherweise lässt man stattdessen einen Dieb mit Rucksack, der ein Fassungsvermögen t^* , Gegenstände mit Wert p_i und Volumen t_i aus einem Tresor klauen ... weswegen das Problem Rucksackproblem und nicht Klausurproblem heisst.

Beim Rucksackproblem nehme ausserdem an: jeder Gegenstand kann **mehrfach** eingepackt werden.

Definiere

- $\text{kosten}[j]$... Volumen von j
- $\text{wert}[j]$... Wert von j

- Sei W_i der maximale Wert von Gegenständen mit Volumen i
- Wenn bei Volumen i der letzte eingepackte Gegenstand j ist, so ist der beste Wert $\text{wert}[j] + W_{i-\text{kosten}[j]}$
- Also:

$$W_i = \max_j (\text{wert}[j] + W_{i-\text{kosten}[j]})$$

- Effiziente Berechnung, indem die Operationen in einer zweckmäßigen Reihenfolge ausgeführt werden.
- Variablen
 - i Kapazität ($1 \leq i \leq M$)
 - j Gegenstand ($1 \leq j \leq N$)
- Felder
 - `best_wert[i]` ... bester Wert bei Kosten i ($\hat{=}$ W_i)
 - `best_obj[i]` ... ausgewähltes j für besten Wert bei Kosten i
- ```
for (j = 1 ; j <= N ; j ++) {
 for (i = 1 ; i <= M ; i++) {
 if (i >= kosten[j]) {
 if (best_wert[i]
 < best_wert[i - kosten[j]] + wert[j]) {
 best_wert[i] = best_wert[i - kosten[j]] + wert[j];
 best_obj[i] = j ;
 }
 }
 }
}
```

# Rucksackproblem

|             |   |   |    |    |    |
|-------------|---|---|----|----|----|
| Bezeichnung | A | B | C  | D  | E  |
| Wert        | 4 | 5 | 10 | 11 | 13 |
| Kosten      | 3 | 4 | 7  | 8  | 9  |

## Lösung des Rucksack-Beispiels:

|              |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
|--------------|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| j = 1;       | i | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| best_wert[i] |   | 0 | 0 | 4 | 4 | 4 | 8 | 8  | 8  | 12 | 12 | 12 | 16 | 16 | 16 | 20 | 20 | 20 |
| best_obj[i]  |   | - | - | A | A | A | A | A  | A  | A  | A  | A  | A  | A  | A  | A  | A  | A  |
| j = 2        |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
| best_wert[i] |   | 0 | 0 | 4 | 5 | 5 | 8 | 9  | 10 | 12 | 13 | 14 | 16 | 17 | 18 | 20 | 21 | 22 |
| best_obj[i]  |   | - | - | A | B | B | A | B  | B  | A  | B  | B  | A  | B  | B  | A  | B  | B  |
| j = 3        |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
| best_wert[i] |   | 0 | 0 | 4 | 5 | 5 | 8 | 10 | 10 | 12 | 14 | 15 | 16 | 18 | 20 | 20 | 22 | 24 |
| best_obj[i]  |   | - | - | A | B | B | A | C  | B  | A  | C  | C  | A  | C  | C  | A  | C  | C  |
| j = 4        |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
| best_wert[i] |   | 0 | 0 | 4 | 5 | 5 | 8 | 10 | 11 | 12 | 14 | 15 | 16 | 18 | 20 | 21 | 22 | 24 |
| best_obj[i]  |   | - | - | A | B | B | A | C  | D  | A  | C  | C  | A  | C  | C  | D  | C  | C  |
| j = 5        |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |
| best_wert[i] |   | 0 | 0 | 4 | 5 | 5 | 8 | 10 | 11 | 13 | 14 | 15 | 17 | 18 | 20 | 21 | 23 | 24 |
| best_obj[i]  |   | - | - | A | B | B | A | C  | D  | E  | C  | C  | E  | C  | C  | D  | E  | C  |

# Rucksackproblem: Beispiel

- Das erste Zeilenpaar zeigt den maximalen Wert (den Inhalt der Felder `best_wert` und `best_obj`), wenn nur Elemente A benutzt werden.
- Das zweite Zeilenpaar zeigt den maximalen Wert, wenn nur Elemente A und B verwendet werden, usw.
- Der höchste Wert, der mit einem Rucksack der Größe 17 erreicht werden kann, ist 24.
- Im Verlaufe der Berechnung dieses Ergebnisses hat man auch viele Teilprobleme gelöst, z. B. ist der größte Wert, der mit einem Rucksack der Größe 16 erreicht werden kann, 22, wenn nur Elemente A, B und C verwendet werden.
- Der tatsächliche Inhalt des optimalen Rucksacks kann mit Hilfe des Feldes `best_obj` berechnet werden. Per Definition ist `best_obj[M]` in ihm enthalten, und der restliche Inhalt ist der gleiche wie im optimalen Rucksack der Größe  $M - \text{kosten}[\text{best\_obj}[M]]$  usw.

# Rucksackproblem: Eigenschaften

- Rucksack-Probleme mit DP:  $O(N \cdot M)$  Zeit
- $\Rightarrow$  Rucksack-Problem leicht, wenn  $M$  klein ist; sonst kann die Laufzeit aber sehr groß werden.
- Problem: die Effizienz dieses Verfahrens hängt davon ab, wie viele *verschiedene Werte* die Kosten annehmen können.  
Z.B. kritisch, wenn Kosten keine kleinen ganzen Zahlen sind.  
(Man kann das Verfahren auf reell-wertige Kosten erweitern, bekommt aber dann schnell dieses Problem.)
- Grundsätzlich gilt bei diesem Algorithmus, dass optimale Entscheidungen nicht geändert werden müssen, nachdem sie einmal getroffen wurden.  
( $\hat{=}$  *Bellmansches Optimalitätsprinzip*)
- Zur Erinnerung: DP funktioniert, immer wenn (dem Algorithmus eine Problemzerlegung zugrunde gelegt wird, bei der) dieses Prinzip gilt.