ADS: Algorithmen und Datenstrukturen 2

8. Teil

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

25. Mai 2010

Editier-Distanz

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

Editier-Distanz

Ansatz: Betrachte Editier-Distanzen von Präfixen.

- D_{ij} sei die Editierdistanz für die Präfixe $(a_1, ..., a_i)$ 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 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:

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

Editierdistanz

	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: $O(n \times m)$

Prinzip Dynamische Programmierung

1. Charakterisiere den Lösungsraum und die Struktur der intendierten optimalen Lösung
2. Definiere rekursiv, wie sich eine optimale Lösung (und der ihr zugeordnete Wert) aus kleineren optimalen Lösungen (und deren Werte) zusammensetzt.
3. Konzipiere den Algorithmus in einer bottom-up Weise so, dass für $n = 1, 2, 3, \ldots$ tabellarisch optimale Teillösungen (und deren zugeordnete Werte) gefunden werden. Beim Finden einer bestimmten optimalen Teillösung der Größe $k > 1$ ist auf alle optimalen Teillösungen der Größe $< k$ zurückzugreifen.

Voraussetzung: Bellmannsches Optimalitätsprinzip

Die optimale Lösung für ein Problem der Größe n setzt sich aus optimalen Teillösungen kleinerer Größe zusammen.

Prinzip Dynamische Programmierung

Wann funktioniert dynamische Programmierung?

- Erstens muss es nicht immer möglich sein, die Lösungen kleinerer Probleme so zu kombinieren, dass sich die Lösung eines größeren Problems ergibt.
- Zweitens kann die Anzahl der zu lösenden Probleme unvertretbar groß sein.
- 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:

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

Matrix-Produkte

- Ein klassischer Anwendungsfall der dynamischen Programmierung ist das Problem der Minimierung des Rechenaufwands, der für die Multiplikation einer Reihe von Matrizen unterschiedlicher Dimension erforderlich ist.

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

- Ziel: Multiplikation der 6 Matrizen. Die Anzahl der Spalten in einer Matrix muss stets mit der Anzahl der Zeilen in der folgenden Matrix übereinstimmen, damit die Multiplikation ausführbar ist.

Matrix-Produkte

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

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

Es gibt noch viele weitere Möglichkeiten:

Matrix-Produkte

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.

Matrix-Produkte

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

Matrix-Produkte

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

- 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.
- $M_k M_{k+1} \dots M_{i-1}$ is eine $r_k \times r_i$ Matrix
  $M_i M_{i+1} \dots M_{l-1} M_l$ is eine $r_i \times r_{l+1}$ Matrix
- Produkt dieser beiden Faktoren benötig 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 [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

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, Entfernungsmatrix $M = (m_{ij})$,
  m_{ij} Entfernung von Stadt i nach Stadt j.
- **Gesucht:** Rundreise über alle Städte mit minimaler Länge,
  also Permutation $\pi : (1, ..., n) \rightarrow (1, ..., n)$.
  $c(\pi) = m_{\pi(n)\pi(1)} + \sum_{i=1}^{n} n - 1 m_{\pi(i)\pi(i+1)}$
- NP-vollständig (Rechenzeit mit großer Sicherheit exponentiell)
- 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 $g(1, 2, ..., n)$. (Stadt 1 kann beliebig gewählt werden, da Rundreise gesucht wird.)
- Es gilt:

$$g(i, S) = \begin{cases} m_{i1} & \textit{falls } S = \varnothing \\ \min_{j \in S} m_{ij} + g(j, S \setminus \{j\}) & \textit{sonst} \end{cases}$$

Traveling Salesman Problem (TSP)

```
for ( i = 2 ; i = n; i++) g[i,{}] = m[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 der vorgebenen Zeit t^* gelöst werden können.
- Ein ebenfalls völlig hypothetischer Student möchte nun seinen Punktestand optimieren.
- Welche Aufgaben sollten bearbeitet werden?
- Für jede Aufgabe $i \in \{1, \ldots, n\}$ kennen wir die Bearbeitungszeit t_i und die dafür erreichbare Punktezahl p_i. Die Aufgabe besteht also darin, eine Teilmenge $S \subseteq \{1, \ldots, n\}$ zu finden, so dass

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

- 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 und vor der Polizei davonlaufen ... weswegen das Problem Rucksackproblem und nicht Klausurproblem heisst.

Rucksackproblem

- Sei P_t die maximale Punktzahl, die bis zum Zeitpunkt t erarbeit werden kann.
- Wenn die letzte bearbeitete Aufgabe j ist, so ist das beste Ergebnis, das erzielt werden kann $p_j + P_{t-t_j}$, also der Punktegewinn für Aufgabe j plus der best-mögliche Punktestand P_{t-t_j}, der in der Zeit $t - t_j$ erreicht werden kann.
- Also: $P_t = \max_j p_j + P_{t-t_j}$

Rucksackproblem

- Effiziente Berechnung, indem die Operationen in einer zweckmäßigen Reihenfolge ausgeführt werden:
- cost ... Punktezahl; size ... Zeitaufwand;
  val[j] ... Wert (Punkte) der Aufgabe j

```
for ( j = 1 ; j <= N ; j ++ ) {
   for ( i = 1 ; i <= M ; i++ ) {
      if ( i >= size[j] ) {
         if ( cost[i] < cost[i - size[j]] + val[j] ) {
            cost[i] = cost[i - size[j]] + val[j] ;
            best[i] = j ;
         }
      }
   }
}
```

Rucksackproblem

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

Lösung des Rucksack-Beispiels:

```
       k 1 2 3 4 5 6  7  8  9 10 11 12 13 14 15 16 17
j = 1
cost [k] 0 0 4 4 4 8  8  8 12 12 12 16 16 16 20 20 20
best [k] - - A A A A  A  A  A  A  A  A  A  A  A  A  A
j = 2
cost [k] 0 0 4 5 5 8  9 10 12 13 14 16 17 18 20 21 22
best [k] - - A B B A  B  B  A  B  B  A  B  B  A  B  B
j = 3
cost [k] 0 0 4 5 5 8 10 10 12 14 15 16 18 20 20 22 24
best [k] - - A B B A  C  B  A  C  C  A  C  C  A  C  C
j = 4
cost [k] 0 0 4 5 5 8 10 11 12 14 15 16 18 20 21 22 24
best [k] - - A B B A  C  D  A  C  C  A  C  C  D  C  C
j = 5
cost [k] 0 0 4 5 5 8 10 11 13 14 15 17 18 20 21 23 24
best [k] - - A B B A  C  D  E  C  C  E  C  C  D  E  C
```

Rucksack Problem: Beispiel

- Das erste Zeilenpaar zeigt den maximalen Wert (den Inhalt der Felder cost und best), 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 berechnet werden. Per Definition ist best[M] in ihm enthalten, und der restliche Inhalt ist der gleiche wie im optimalen Rucksack der Größe M - size [best [M]] usw.

Rucksack Problem: Eigenschaften

- Für die Lösung des Rucksack-Problems mit Hilfe der dynamischen Programmierung wird eine zu $N \cdot M$ proportionale Zeit benötigt.
- Somit kann das Rucksack-Problem leicht gelöst werden, wenn M nicht groß ist; für große Fassungsvermögen kann die Laufzeit jedoch unvertretbar groß werden.
- Eine grundlegende Schwierigkeit ist es, dass das Verfahren nicht anwendbar ist, wenn M sehr viele verschiedene Werte annehmen kann, z.B., wenn Größe oder Wert der Objekte beliebige z. B. reelle Zahlen anstatt (kleiner) ganzer Zahlen sind.
- Wenn jedoch die Fassungsvermögen sowie die Größen und Werte der Gegenstände ganze Zahlen sind, so gilt das grundlegende Prinzip, dass optimale Entscheidungen nicht geändert werden müssen, nachdem sie einmal getroffen wurden.
- Jedesmal, wenn dieses allgemeine Prinzip zur Anwendung gebracht werden kann, ist die dynamische Programmierung anwendbar.