ADS: Algorithmen und Datenstrukturen 2

Teil 10

Prof. Dr. Gerhard Heyer

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

13. Juni 2018

[Letzte Aktualisierung: 12/06/2018, 10:28]

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.

Matrix-Produkte

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

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

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

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?

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,
  $M_1 M_2 M_3 M_4$ zu multiplizieren:
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_1 M_2 M_3$ und addiere die Kosten der Multiplikation dieses Ergebnisses mit M_4.
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_2 M_3 M_4$ und addiere die Kosten der Multiplikation dieses Ergebnisses mit M_1.
  - Entnimm der gespeicherten Tabelle die Kosten der Berechnung von $M_1 M_2$ sowie von $M_3 M_4$ 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.

Allgemeine 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 $r_k r_i r_{l+1}$ 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}.

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

Beschreibung der Tabelle

- 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.
- 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.

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 $\pi : (1, ..., n) \rightarrow (1, ..., n)$, für die

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

  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: $g(1, \{2, ..., n\})$.
  Anmerkung: Stadt 1 kann beliebig gewählt werden, da Rundreise gesucht wird.
- Es gilt:

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

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] gem"aß Formel
Berechne g[1,{ 2,...,n }] gem"aß 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 $(n-1)!$.

Dynamische Programmierung für TSP - Beispiel

Es seien n= 4 und D =

	1	2	3	4
1	-	5	13	8
2	7	-	9	14
3	12	10	-	6
4	8	4	9	-

(Wie Beispiel aus Vorlesung 9, aber auf 4 Städte verkürzt)

Dynamische Programmierung für TSP - Beispiel (2)

$$|S| = 0 : g(2,0) = 7, g(3,0) = 12, g(4,0) = 8$$

$|S| = 1 :$
$g(2,\{3\}) = d_{23} + g(3,0) = 9 + 12 = 21$
$g(2,\{4\}) = d_{24} + g(4,0) = 14 + 8 = 22$
$g(3,\{2\}) = d_{32} + g(2,0) = 10 + 7 = 17$
$g(3,\{4\}) = d_{34} + g(4,0) = 6 + 8 = 14$
$g(4,\{2\}) = d_{42} + g(2,0) = 4 + 7 = 11$
$g(4,\{3\}) = d_{43} + g(3,0) = 9 + 12 = 21$

$|S| = 2 :$
$g(2,\{3,4\}) = min\{d_{23} + g(3,4), d_{24} + g(4,3)\} = min\{9 + 14, 14 + 21\} = 23$
$g(3,\{2,4\}) = min\{d_{32} + g(2,4), d_{34} + g(4,2)\} = min\{10 + 22, 6 + 11\} = 17$
$g(4,\{2,3\}) = min\{d_{42} + g(2,3), d_{43} + g(3,2)\} = min\{4 + 21, 9 + 17\} = 25$

Dynamische Programmierung für TSP - Beispiel (3)

$|S| = 3$:
$g(1, \{2, 3, 4\}) =$
$min\{d_{12} + g(2, \{3, 4\}), d_{13} + g(3, \{2, 4\}, d_{14} + g(4, \{2, 3\}\} =$
$min\{5 + 23, 13 + 17, 8 + 25\} = 28$

Die optimale Rundreise mit 1 als Startpunkt lautet also:
$\pi = (1, 2, 3, 4)$ mit der Länge 28

Bei jeder Berechnung $g(i, S)$ kann das j gespeichert werden, das als Minimum angenommen wird.
Die Rundreise π lässt sich dann wie folgt konstruieren:

- starte mit 1
- berechne j aus $g(1, S_0)$
- berechne π rekursiv weiter

Das "Alignment"- bzw. "String-Editing"-Problem

Problem: Vergleiche 2 Strings x und y über dem selben Alphabet $\mathcal{A}$

bei gleicher Länge wär das einfach: "Hamming-Distanz" (Anzahl der unterschiedlichen Stellen); deshalb . . .

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

Frage: Was ist die minimale Anzahl an Editier-Operationen, mit der x in y umgewandelt werden kann? ("Levenshtein-Distanz")

Beispiel: `LIPSIA` $\rightarrow$ `LEIPZIG`

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

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

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 ein geeignete Operation $\omega \in \Omega$.

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

Dynamische Programmierung

Editier-Distanz

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.

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 $w_2 = b_1 ... b_m$

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**
   - **if** $a_i = b_j$ **then** c=0 **else**
   - $D(i,j) = min\{D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+1\}$

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: $O(n \times m)$

- = typisches Beispiel für Dynamische Programmierung (DP)

Parsing

Problem: Finde richtige Zerlegung einer Zeichenkette (in terminale Ketten) entsprechend einer vorgegebenen Grammatik

- Programmiersprachen (z.B. Java, HTML, ...)
- natürlichsprachliche Texte

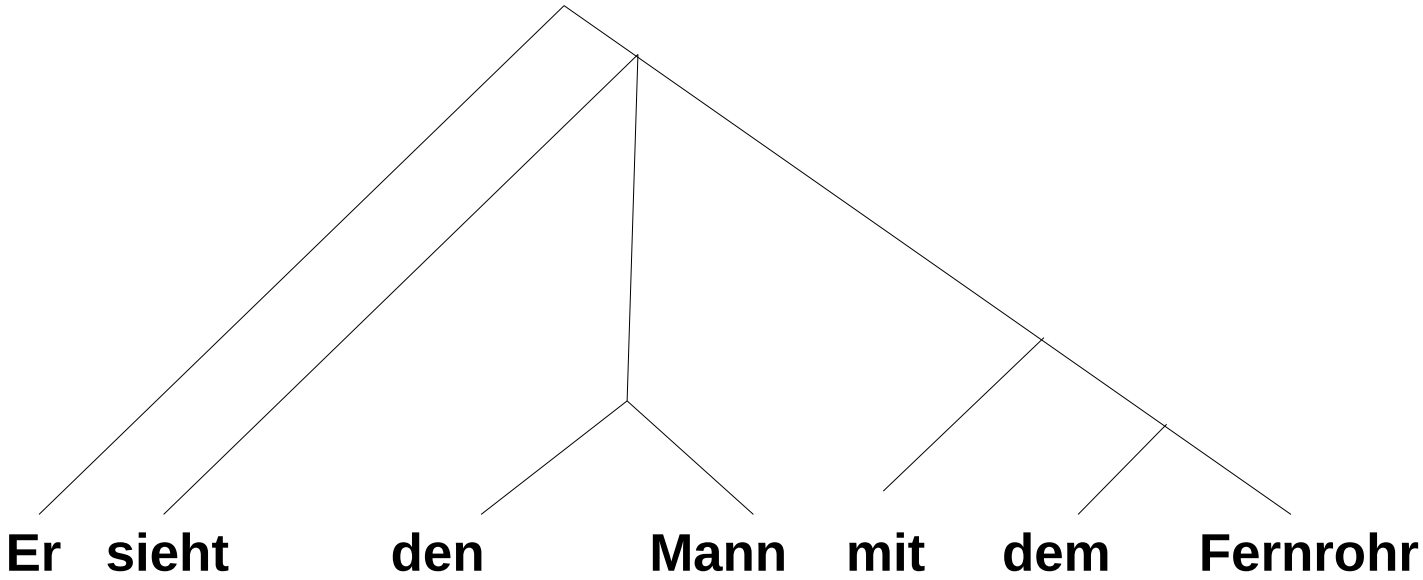
Beispiel:

Er sieht den Mann mit dem Fernrohr.

Eine (von zwei) richtigen Klammerungen:

(Er) ((sieht) ((den) (Mann)) ((mit) ((dem) (Fernrohr))))

Baumstruktur



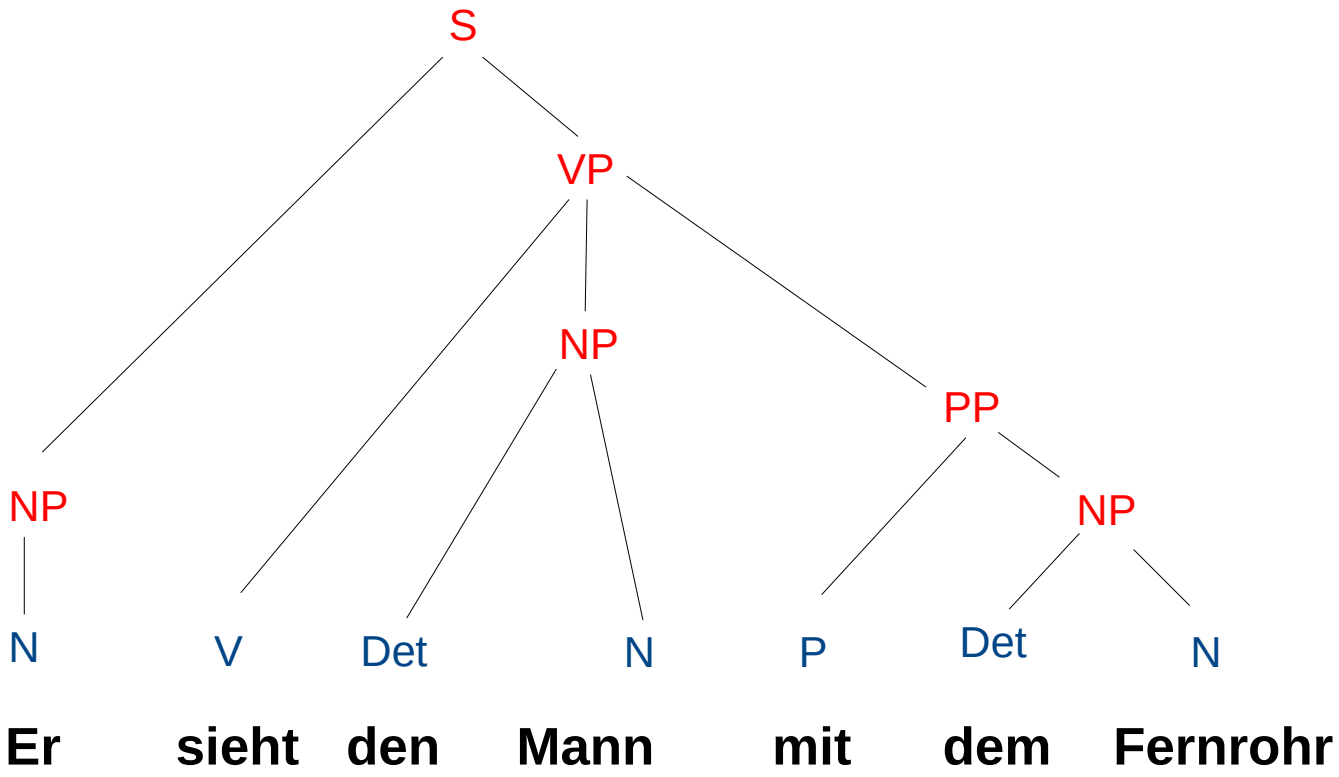
Grammatik

Grammatik hilft bei der richtigen Klammung!

Kontextfreie Chomsky Grammatik: $< s, T, N, P >$ mit

- s Startsymbol
- $T = \{Er, Mann, Fernrohr, sieht, den, dem, mit\}$
- $N = \{N, V, Det, P, NP, VP, PP\}$
- $P = \{S \rightarrow NP\ VP, NP \rightarrow (Det)\ N\ (PP), VP \rightarrow V\ NP\ (PP),\ PP \rightarrow P\ NP\}$

Satzstruktur



Parser

Parser: Automat zur Erkennung der Syntaxstruktur einer Eingabe

Parsergenerator: Automat zur Erzeugung eines Parsers aus einer Grammatik, für kontextfreie Sprachen z.B. ein *Kellerautomat*

Wichtige Parameter:

- Analyserichtung - links, rechts, middle-out
- Strategie - top-down, bottom-up, Tiefensuche, Breitensuche
- Verhalten bei Konflikten - Backtracking, Chart

Parsing eine kontextfreie Sprache

Verfahren mit Kellerautomat:

1. Verarbeitung der Eingabe top-down von links nach rechts
2. Abarbeiten (Ersetzen) eines Strings sobald dies möglich ist
3. Ansonsten auf dem Stack abspeichern und diesen Ausdruck abarbeiten sobald dies möglich ist

Problem: Natürliche Sprachen enthalten syntaktische Mehrdeutigkeiten, außerdem sog. *long distance dependencies*, deshalb für natürliche Sprache oft ineffizient, weil Teilbäume wiederholt abgearbeitet werden

Satzstruktur

Eingabe:

Er sieht den Mann mit dem Fernrohr

1 2 3 4 5 6 7

Zeile	Knoten	Regel	Aktion	Ergebnis	Stack
1		(1) S → NP VP	expandiere	NP VP	
2		(2) NP → Det N	expandiere	Det N	
3	1	**Lex (Det)**	**wähle Eintrag**	***den***	***er***
4		**(3) NP → N**	**expandiere**	**N**	
5	1	**Lex (N)**	**wähle Eintrag**	***er***	**--**
6		(4) VP → V NP	expandiere	V	
7	2	**Lex (V)**	**wähle Eintrag**	**sieht**	
8		(3) NP → Det N	expandiere	Det N	
9	3,4	**Lex (Det, N)**	**wähle Einträge**	***den, Mann***	
10		**Ø**	**nächste Regel**		***mit dem Fernrohr***
11		(5) VP → V NP PP	expandiere	Det N PP	

Parsing mit DP

Viel vermeidbares **backtracking**

Idee: Speichere Zwischenergebnisse in Tabelle

1. für kontextfreie Grammatiken in Chomsky Normalform (CNF) Algorithmus von Cocke/Younger/Kasomie (CYK)
2. Verallgemeinerung als sog. Chart Parsing (Early 1970)
3. bottom-up, links-rechts

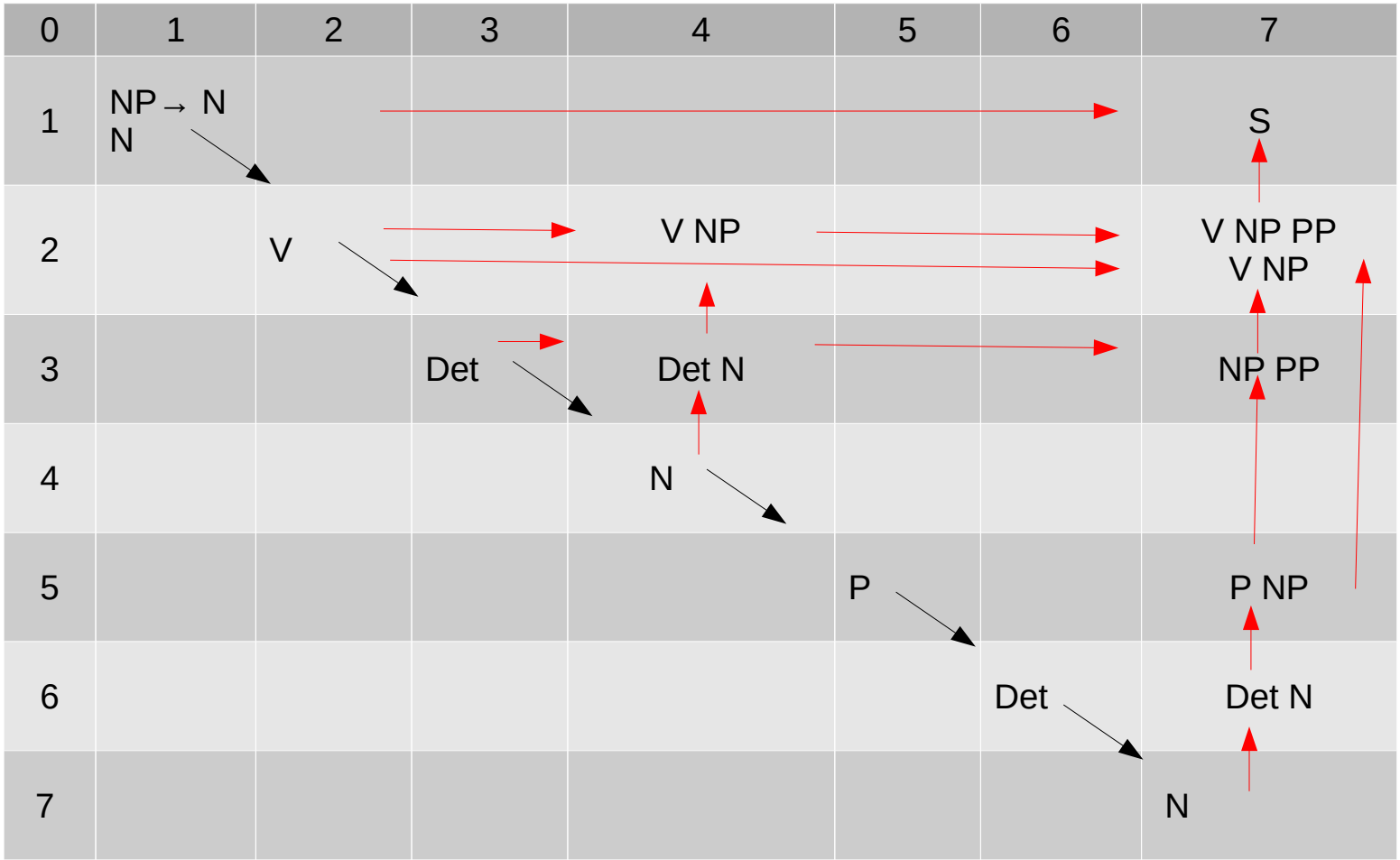
Verfahren

1. repräsentiere Wörter als Kanten (nicht als Knoten)
2. erstelle eine Tabelle der Größe n x n
3. beginne in der oberen linken Ecke
   - für jede Zelle in der Diagonalen trage eine laut Grammatik mögliche lexikalische Kategorie ein
   - für jede vervollständigte Zelle suche in derselben Spalte entsprechend der Grammatik nach möglichen Vervollständigungen
   - ist in der letzten Spalte eine Vervollständigung auf s möglich, melde Erfolg

Komplexität $O(n^3)$

Chart

Eingabe:
0 **Er** 1 **sieht** 2 **den** 3 **Mann** 4 **mit** 5 **dem** 6 **Fernrohr** 7



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 unvertretbar 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.