11 Statistische Syntaxmodelle

11 Statistische Syntaxmodelle

11.1 Erweiterungen von CFG-Grammatiken

11.1.1 Grammatikentwicklung

Grammatik-Entwicklung

- **Ziel automatischer Syntaxanalyse**:
  → Entwicklung von Grammatik mit hoher **Abdeckung/*coverage***
  → beschreibungsadäquates **Model der syntaktischen Struktur eines sehr großen Ausschnitts** einer natürlichen Sprache

- **Unifikationsgrammatiken:**
  → modellieren **Agreement, Rektion und Subkategorisierung** über Merkmalconstraints
  → Erkennung genau der **wohlgeformten Sätze**
  → **beschreibungsadäquate Strukturzuweisung**

Grammatiksysteme

- **von Experten erstellte Grammatik-Systeme**, die den Anspruch haben, einen großen Ausschnitt der Syntax einer natürlichen Sprache abzubilden:
  - Head-Driven Phrase Structure Grammar (**HPSG**):
    $\rightarrow$ **LinGO Matrix Framework**
    $\rightarrow$ **delph-in.net** (deutsche Grammatik)
  - Lexical Functional Grammar (**LFG**): **Pargram** Projekt
  - Lexicalized **Tree Adjoining Grammar**: **XTAG** Projekt

Zunahme Ambiguität mit Abdeckung

- **hohe Abdeckung** (viele Regeln, großes Lexikon mit ambigen Einträgen) **und Input langer (komplexer) Sätze** führen zu:
  → **hoher Aufwand beim Parsing**
  → **große Anzahl an Ableitungen/Analysen (Ambiguität)**

- z. B. durch Ambiguität im Lexikon:

  [NP Time] [VP flies] like an arrow.

  [VP Time] [NP flies] like an arrow.

  [NP Time flies] [VP like] an arrow.

11.1.2 Disambiguierung durch statistische Modelle

- **Erweiterung** von CFGs **um probabilistische Parameter**
  $\rightarrow$ **gewichtete Grammatik**: Produktionsregeln erhalten Bewertung
  $\rightarrow$ erlaubt **Ranking der Ableitungen** eines strukturell ambigen Satzes aufgrund von **Trainingsdaten aus Korpus**

- **Disambiguierung über empirisches Modell**
  → **statt Disambiguierung über** explizite **semantische Informationen** im Anschluss an syntaktisches Parsing durch semantisches Parsing:
  → Auswahl Ableitung aufgrund von **statistischen Informationen aus Korpusdaten zu Kollokationen von Wörtern und syntaktischen Kategorien**
  → **beste syntaktische Analyse** eines Satzes = **die im Sprachgebrauch häufigste**
  → **graduelle Modellierung von Grammatikalität**

- **Probabilistische CFG (= PCFG)** erlaubt in Kombination mit **dynamischem Parsing** das effiziente Auffinden der besten (= wahrscheinlichsten) Ableitung
- **ohne Gewichtung**: dynamische Programmierung (CYK, Earley) kann zwar Parsing-Aufwand bei großem Suchraum (großer Grammatik) reduzieren, aber **keine Auswahl** treffen aus den gefundenen Ableitungen
- **statistische Informationen** können auch **im Parsing von Unfikationsgrammatiken** (wie LFG, HPSG) zur Disambiguierung verwendet werden

- **nächste Sitzung:** statt bloßer Erweiterung einer gegebenen CFG um statistische Informationen aus Treebanks: **Extraktion von Grammatiken aus Treebanks**
  $\rightarrow$ in solchen ***induzierten* Grammatiken** können auch **lexikalische Informationen und Informationen zum strukturellen Kontext** berücksichtigt werden, die der weiteren **Disambiguierung** dienen

11.1.3 Normalisierung und *Parent-Annotation* von CFG-Grammatiken

Normalisierung

- **Chomsky-Normalform** (CNF)

  $\rightarrow$ Einschränkung der Form von CFG-Regeln:

  $\Rightarrow$ **RHS: 2 Nichtterminale oder 1 Terminal:** $A \rightarrow B\ C$**,** $A \rightarrow a$

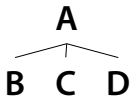
  $\rightarrow$ **Binärbäume** (bis Präterminalknoten, dort: unäre Bäume)

  $\rightarrow$ **jede CFG kann in CNF umgewandelt** werden:

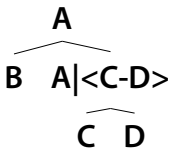
  $A \rightarrow B\ C\ D \Rightarrow A \rightarrow B\ X, X \rightarrow C\ D$ (***Right-Factored***)

  $A \rightarrow B\ C\ D \Rightarrow A \rightarrow X\ D, X \rightarrow B\ C$ (***Left-Factored***)

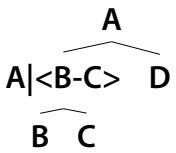
Original:



Right-Factored:



Left-Factored:



- **Anwendung Chomsky-Normalform:**
  - notwendig für **CYK-Chart-Parsing**
  - zur **Reduktion von extrahierten Grammatikregeln** aus flach annotiertem Korpus:
    * $VP \rightarrow V\ PP$
      $VP \rightarrow V\ PP\ PP$
      $VP \rightarrow V\ PP\ PP\ PP$ usw.
    * mit ***Chomsky-adjunction*** ($A \rightarrow A\ B$):
      $VP \rightarrow V\ PP$
      $VP \rightarrow VP\ PP$

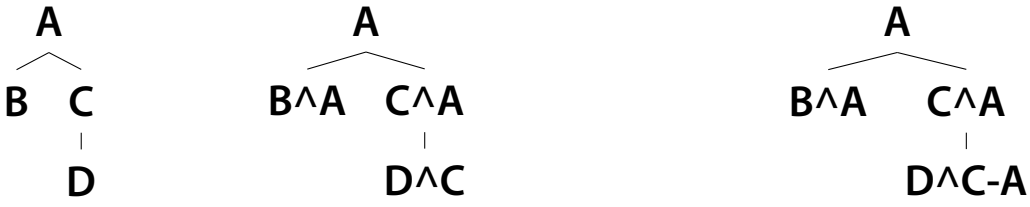
- ***Parent Annotation***

  → Kategorie des **Mutterknoten in Kategoriensymbol** aufnehmen

  → Modellierung von **Kontext**; s. nächste Sitzung: *history-based PCFGs*

  → ergibt anderes PCFG-Modell: **mehr Nichtterminale, andere Gewichtung**

Original: **Parent Annotation:** **+Grandparent Annot.:**



11.2 Probabilistische kontextfreie Grammatiken

Literatur:

- **MS**: Manning, Christopher D. & Schütze, Hinrich (1999): *Foundations of Statistical Natural Language Processing*.
- NLTK-Teilkapitel 8.6 ('*Grammar Development*') und 8.5.2 ('*Scaling up*'): `http://www.nltk.org/book/ch08.html`
- Teilkapitel 2.12 ('*Grammar Induction*') des Zusatzkapitels zu Kapitel 8: `http://www.nltk.org/book/ch08-extras.html`
- Die Teilkapitel 2.9-2.11 des Zusatzkapitels zu Kapitel 8 behandeln probabilistische Chart Parsing-Algorithmen: `http://www.nltk.org/book/ch08-extras.html`

Disambiguierung durch statischen Erweiterung (PCFG)

- Erweiterung von CFG-Grammatiken durch statistische Parameter zur **Disambiguierung**
- **strukturelle Disambiguierung** durch *parse selection* (Herausfiltern der wahrscheinlichsten Ableitung)
- **Wahrscheinlichkeiten der Regeln** müssen anhand von Korpusdaten gelernt werden (Parameter-Abschätzung)
- Algorithmen dynamischer Programmierung (**Viterbi-Algorithmus**) zur **effizienten Auffindung der wahrscheinlichsten Ableitung**

11.2.1 Probabilistische kontextfreie Grammatik (PCFG)

- **PCFG** = kontextfreie Grammatik, deren Regeln mit **Wahrscheinlichkeiten** gewichtet sind:
  $VP \rightarrow VP\ PP\ \ 0.6$
  $NP \rightarrow NP\ PP\ \ 0.2$
  $NP \rightarrow N\ \ 0.8$

- **Wahrscheinlichkeiten** aller Regeln **für die Expansion eines bestimmten Nonterminals** addieren sich zu **1**

- **Ableitung/Baum** ist **Menge an Regeln/Expansionen**
  $\rightarrow$ Teilbäume mit Tiefe 1

- **Wahrscheinlichkeit einer Ableitung T** $(Tree)$ als Multiplikation der Wahrscheinlichkeiten ihrer Regeln:
  $P(T) = \prod_{i=1}^{n} P(R_i) = \prod_{i=1}^{n} P(RHS_i|LHS_i)$
  $\rightarrow$ Iteration über die n Knoten im Baum: **Produkt der Wahrscheinlichkeit der Expansion des LHS-Knotens von** R_i **zu RHS-Smbolfolge von** R_i
  $\rightarrow$ **Annahme Unabhängigkeit der Regel-Auswahl**

- zur **Disambiguierung** muss die **wahrscheinlichste Ableitung T*** zu einem Satz S gefunden werden:
  $\rightarrow\ T^* = arg\ max\ P(T|S) = arg\ max \frac{P(T,S)}{P(S)} = arg\ max\ \frac{P(T)}{P(S)}$
  $(P(T,S) = P(T)P(S|T), P(S|T) = 1;$
  $jeder\ Baum\ leitet\ genau\ einen\ Satz\ ab)$
  $\rightarrow T^* = arg\ max\ P(T)$
  $(da\ P(S)\ konstant\ fuer\ ein\ S, also\ irrelevant\ fuer\ Auswahl\ Ableitung\ zu\ gegebenem\ Satz)$

- **Satzwahrscheinlichkeit**: Summe der Wahrscheinlichkeiten aller möglichen Ableitungen eines Satzes:
  $P(S) = \sum P(T,S) = \sum P(T)$

Beispiel-PCFG PP-Attachment-Ambiguität

$S \rightarrow NP\ VP\ \ 1$

$PP \rightarrow P\ NP\ \ 1$

$NP \rightarrow DET\ N\ \ 0.5$

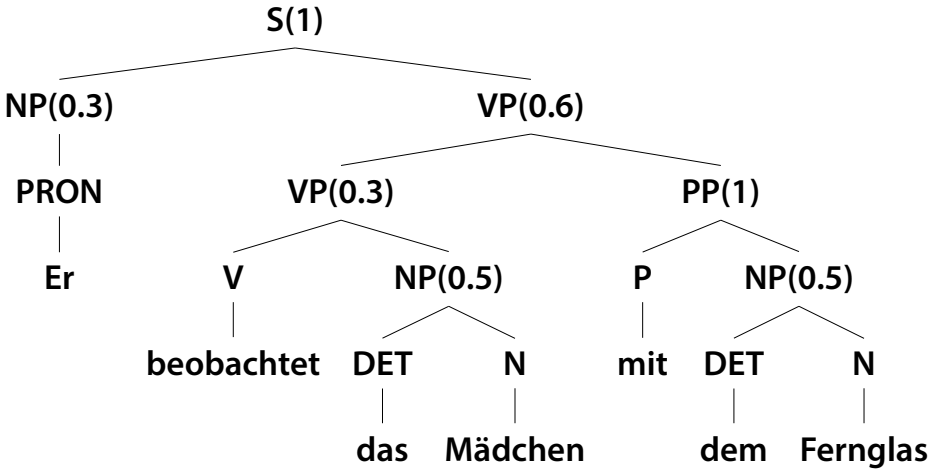
$NP \rightarrow PRON\ \ 0.3$

$NP \rightarrow NP\ PP\ 0.2$

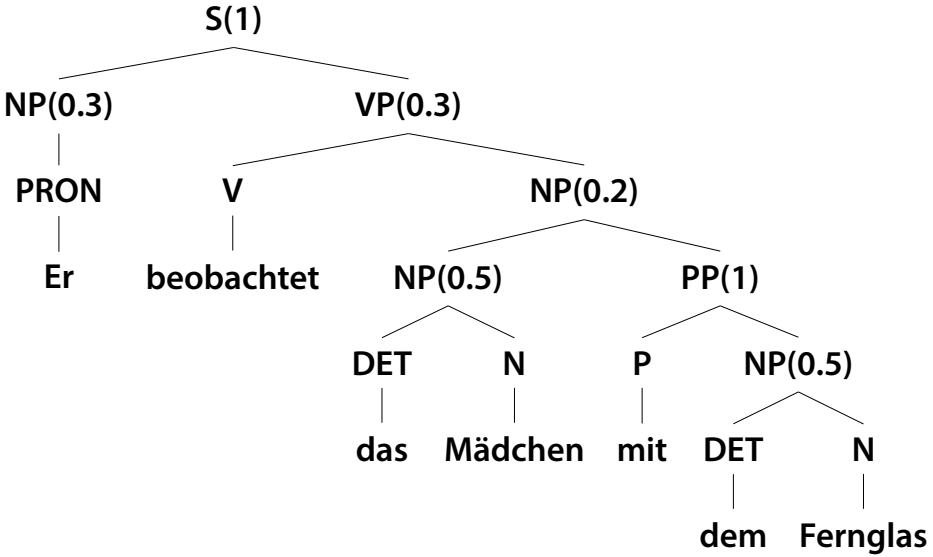
$VP \rightarrow V\ \ 0.1$

$VP \rightarrow V\ NP\ \ 0.3$

$VP \rightarrow VP\ PP\ \ 0.6$



$$P(T_1) = 1 * 0.3 * \mathbf{0.6} * 0.3 * 0.5 * 1 * 0.5 = \mathbf{0.0135}$$



$P(T_2) = 1 * 0.3 * 0.3 * \mathbf{0.2} * 0.5 * 1 * 0.5 = \mathbf{0.0045}$

$\Rightarrow$ **Auswahl adverbialer Lesart** : $P(T_1) > P(T_2)$

Grund: $P(VP, PP|VP) > P(NP, PP|NP)$

Auflistung 1: *NLTK: Probabilistische kontextfreie Grammatik (PCFG)*

```
grammar1 = nltk.PCFG.fromstring("""
    S → NP VP [1.0]
    PP → P NP [1.0]
    NP → Det N [0.8] | Det N PP [0.1] | 'I' [0.1]
    VP → V NP [0.8] | VP PP [0.2]
    Det → 'an' [0.7] | 'my' [0.3]
    N → 'elephant' [0.5] | 'pajamas' [0.5]
    V → 'shot' [1.0]
    P → 'in' [1.0]
    """)
```

```
15 parser = nltk.ViterbiParser(grammar1)
16
17 for tree in parser.parse(sent):
18     print(tree)
19 #(S
20 #   (NP I)
21 #   (VP
22 #     (VP (V shot) (NP (Det an) (N elephant)))
23 #     (PP (P in) (NP (Det my) (N pajamas)))))
     (p=0.0005376)
24
25 #(VP-attachment als wahrscheinlichste
     Ableitung)
26
27
28
```

```
29 grammar2 = nltk.PCFG.fromstring("""
30     S → NP VP [1.0]
31     PP → P NP [1.0]
32     NP → Det N [0.7] | Det N PP [0.2] | 'I'
         [0.1]
33     VP → V NP [0.8] | VP PP [0.2]
34     Det → 'an' [0.7] | 'my' [0.3]
35     N → 'elephant' [0.5] | 'pajamas' [0.5]
36     V → 'shot' [1.0]
37     P → 'in' [1.0]
38     """)
39
40
41
42
43
```

```
44 parser = nltk.ViterbiParser(grammar2)
45 for tree in parser.parse(sent):
46     print(tree)
47 #(S
48 #   (NP I)
49 #   (VP
50 #     (V shot)
51 #     (NP
52 #       (Det an)
53 #       (N elephant)
54 #       (PP (P in) (NP (Det my) (N
     pajamas)))))) (p=0.000588)
55
56 #(NP-attachment als wahrscheinlichste
     Ableitung)
```

11.2.2 Abschätzung der Regelwahrscheinlichkeiten

- Zwei Methoden für Abschätzung:
  - ***supervised*** = Bestimmung der relativen Häufigkeiten der Expansionen eines Nichtterminals in geparstem (syntaktisch annotiertem) Korpus (**Maximum Likelihood Estimation**)
  - ***unsupervised*** = wiederholtes Parsen von Korpus mit der gegebenen kontextfreien Grammatik und sukzessive Verbesserung eines statistischen Modells (**Inside-Outside-Algorithmus**)

Maximum Likelihood Estimation

- Abschätzung der Regelwahrscheinlichkeit als **relative Häufigkeit der Expansion des LHS-Nonterminals zu RHS-Symbolfolge in Treebank** (syntaktisch annotiertem Korpus)
- $P(\alpha \rightarrow \beta|\alpha) = \frac{count(\alpha \rightarrow \beta)}{\sum_\gamma count(\alpha \rightarrow \gamma)} = \frac{count(\alpha \rightarrow \beta)}{count(\alpha)}$
- **Expansionswahrscheinlichkeit:**
  $P(RHS|LHS) = P(Expansion|Nonterminal)$
  $\rightarrow$ Idee: gute probabilistische Grammatik **maximiert die Wahrscheinlichkeit der Trainingsdaten**

- **Beispiel**: Wahrscheinlichkeit für Expansion *VP* $\rightarrow$ *V NP PP*:

$$P(V, NP, PP|VP) = \frac{count(VP \rightarrow V\ NP\ PP)}{count(VP \rightarrow \backslash *)}$$

- $count(VP \rightarrow V\ NP\ PP) = 10$
  $count(VP \rightarrow V\ NP) = 50$
  $count(VP \rightarrow V) = 40$
  $\Rightarrow MLE(VP \rightarrow V\ NP\ PP \mid VP) = 1/10$

Inside-Outside-Algorithmus

- Abschätzung der Regelwahrscheinlichkeiten auch **ohne syntaktisch annotiertes Trainingskorpus**, d. h. ***unsupervised*** möglich mit **Inside-Outside-Algorithmus**
- Variante von **EM-Algorithmus** (*Expectation-Maximation*)
  $\rightarrow$ **iterativen Abschätzung der Regelwahrscheinlichkeiten** (als Parameter des statistischen Modells)
  $\rightarrow$ **Übertragung des Forward-Backward-Algorithmus** (zur Abschätzung von Parametern bei HMMs) auf PCFGs

11.2.3 *Probabilistisches Parsing (Zusatz)

- **Suche der wahrscheinlichsten Ableitung**:
  $arg\,max\ P(T|S) = arg\,max\ P(T)$
- **Suche aller Ableitungen** und Berechnung ihrer Wahrscheinlichkeiten wird **bei großen Grammatiken sehr aufwendig**
- besser: **probabilistische Varianten von Chart-Parsing-Algorithmen** wie CYK- oder Earley-Algorithmus
- Verwendung statistischer Informationen in **dynamischer Programmierung** zum effizienten Auffinden der wahrscheinlichsten (Teil)bäume

- **PCFG-Version des Viterbi-Algorithmus** (analog zu HMM): **Finden der wahrscheinlichsten verborgenen Zustandsfolge** ($Ableitung\ T$), die die beobachtete Sequenz emitiert ($Satz\ S$)
  $\rightarrow$ Bestimmung des **wahrscheinlichsten Baumes** durch Zurückgreifen **auf berechnete Teilbäume**
  $\rightarrow$ die **Wahrscheinlichkeit größerer** Teilbäume ergibt sich **aus den Wahrscheinlichkeiten der kleineren**, da aufgrund der Kontextfreiheit die Wahrscheinlichkeit eines Teilbaums unabhängig von seiner Position ist
  $\rightarrow$ **nur die Teilbäume mit höchster Wahrscheinlichkeit** werden behalten und zur Berechnung verwendet

- **Performanz-Optimierung** des Parsings durch Verwendung statistischer Informationen
  $\rightarrow$ statt allen möglichen **nur die wahrscheinlichsten Teilergebnisse verwenden**

- `nltk.ViterbiParser`
  → **Bottom-up-PCFG-Parser**
  → berechnet inkrementell (beginnend mit Spanne Länge 1) die **wahrscheinlichsten (Teil)bäume** durch Ausfüllen einer '***Most Likely Constituents Table***'

- für **gegebene Spanne** und **Knoten-Wert** (LHS einer Regel):
  → Suche nach **Folgen von Tabellen-Einträgen**, die gemeinsam die **Spanne abdecken**
  → Überprüfung, ob **Tabellen-Einträge die RHS-Werte der Regel als Knotenwerte** haben (LHS der Tabellen-Einträge)

Span	Node	Tree	Prob
$[0:1]$	NP	(NP I)	0.15
$[6:7]$	NP	(NN telescope)	0.5
$[5:7]$	NP	(NP the telescope)	0.2
$[4:7]$	PP	(PP with (NP the telescope))	0.122
$[0:4]$	S	(S (NP I) (VP saw (NP the man)))	0.01365
$[0:7]$	S	(S (NP I) (VP saw (NP (NP the man) (PP with (NP the telescope)))))	0.0004163250

Abbildung 1: *Most Likely Constituents Table (Ausschnitt)*

- Tabelle enthält **nur die wahrscheinlichste Ableitung für eine Spanne und Knoten-Wert**: z. B. wird nur die *NP-attachment*-Variante für Spanne $[1:7]$ und Knoten-Wert VP aufgenommen:

  $[1:7]$ VP (VP saw (NP (NP the man) (PP with (NP the telescope))))

  $[1:7]$ VP (VP saw (NP (NP the man)) (PP with (NP the telescope)))

Auflistung 2: *NLTK: PCFG-Parsing mit Viterbi-Parser*

```
1  #http://www.nltk.org/_modules/nltk/parse/viterbi.html
2  #http://www.nltk.org/book/ch08-extras.html
3
4  grammar = nltk.PCFG.fromstring('''
5    NP  → NNS [0.5] | JJ NNS [0.3] | NP CC NP
       [0.2]
6    NNS → "cats" [0.1] | "dogs" [0.2] | "mice"
       [0.3] | NNS CC NNS [0.4]
7    JJ  → "big" [0.4] | "small" [0.6]
8    CC  → "and" [0.9] | "or" [0.1]
9    ''')
10
11 sent = 'big cats and dogs'.split()
12
13 viterbi_parser = nltk.ViterbiParser(grammar)
```

```
14 for tree in viterbi_parser.parse(sent):
15     print(tree)
16 #(NP (JJ big) (NNS (NNS cats) (CC and) (NNS
     dogs))) (p=0.000864)
17
18 viterbi_parser.trace(3)
19 for tree in viterbi_parser.parse(sent):
20     print(tree)
21
22
23 #Inserting tokens into the most likely
     constituents table...
24 #   Insert: |=...| big
25 #   Insert: |.=..| cats
26 #   Insert: |..=.| and
27 #   Insert: |...=| dogs
```

```
28 #Finding the most likely constituents spanning
     1 text elements...
29 #   Insert: |=...| JJ → 'big' [0.4]
                    0.4000000000
30 #   Insert: |.=..| NNS → 'cats' [0.1]
                   0.1000000000
31 #   Insert: |.=..| NP → NNS [0.5]
                     0.0500000000
32 #   Insert: |..=.| CC → 'and' [0.9]
                    0.9000000000
33 #   Insert: |...=| NNS → 'dogs' [0.2]
                   0.2000000000
34 #   Insert: |...=| NP → NNS [0.5]
                     0.1000000000
35
36
```

```
37 #Finding the most likely constituents spanning
     2 text elements...
38 #   Insert: |==..| NP → JJ NNS [0.3]
                    0.0120000000
39 #Finding the most likely constituents spanning
     3 text elements...
40 #   Insert: |.===| NP → NP CC NP [0.2]
                  0.0009000000
41 #   Insert: |.===| NNS → NNS CC NNS [0.4]
                 0.0072000000
42 #   Insert: |.===| NP → NNS [0.5]
                       0.0036000000
43 #  Discard: |.===| NP → NP CC NP [0.2]
                  0.0009000000
44 #  Discard: |.===| NP → NP CC NP [0.2]
                  0.0009000000
```

```
45 #Finding the most likely constituents spanning
     4 text elements...
46 #   Insert: |====| NP → JJ NNS [0.3]
                    0.0008640000
47 #  Discard: |====| NP → NP CC NP [0.2]
                   0.0002160000
48 #  Discard: |====| NP → NP CC NP [0.2]
                   0.0002160000
49 #(NP (JJ big) (NNS (NNS cats) (CC and) (NNS
     dogs))) (p=0.000864)
```

- `nltk.parse.pchart` = **Klasse von Bottom-up-PCFG-Chart-Parsern**
- **Chart-Parsing mit zusätzlicher Datenstruktur *edge queue***, deren **Sortierung** die Reihenfolge der Abarbeitung der Zustände festlegt
  $\rightarrow$ ***edge*** in Chart-Parsing nach Kay = **Zustand** bei Earley/CYK
- im Gegensatz zu Viterbi-Parser wird **nicht nur die wahrscheinlichste Ableitung** gefunden, sondern die **n-besten Ableitungen**
- Verwendung von **statistischen Daten zur Sortierung**

- **Strategien zur Sortierung des *edge queues***:

  - **Lowest Cost First** = `nltk.InsideChartParser`
    $\rightarrow$ **Sortierung nach Wahrscheinlichkeit** der Ableitungen
    $\rightarrow$ findet immer die **optimale** Lösung (wahrscheinlichste Ableitung)
    $\rightarrow$ **Problem: kürzere Teilergebnisse haben üblicherweise eine höhere Wahrscheinlichkeit** (P = Produkt der Regelwahrscheinlichkeiten) und werden so zuerst abgearbeitet; **vollständige Ableitung wird erst spät produziert**

- **Best-First Search** = `nltk.LongestChartParser`
  → **Sortierung nach Länge** (für vollständige Ableitung: längste Spanne gesucht)
  → i. A. **schneller** als *Lowest Cost First*
  → garantiert nicht, dass optimale Ableitung gefunden wird

- **Beam Search (Pruning)** = `nltk.InsideChartParser(grammar, beam_size=20)`
  → *Lowest-Cost-First*, aber **nur die n-besten partiellen Ergebnisse behalten** (= Pruning)
  → **schneller** als *Lowest-Cost-First* ohne Pruning
  → garantiert nicht, dass optimale Ableitung gefunden wird
  → garantiert nicht, dass überhaupt eine Ableitung gefunden wird (wenn notwendige *edges* fehlen)

Auflistung 3: *NLTK: PCFG-Parsing mit ChartParser*

```
1  #http://www.nltk.org/_modules/nltk/parse/pchart.html
2  #http://www.nltk.org/book/ch08-extras.html
3
4  inside_parser = nltk.InsideChartParser(grammar)
5  longest_parser =
     nltk.LongestChartParser(grammar)
6  beam_parser = nltk.InsideChartParser(grammar,
     beam_size=20)
7
8  for tree in inside_parser.parse(sent):
9      print(tree)
10 #(NP (JJ big) (NNS (NNS cats) (CC and) (NNS
     dogs))) (p=0.000864)
11 #(NP (NP (JJ big) (NNS cats)) (CC and) (NP
     (NNS dogs))) (p=0.000216)
```

11.3 Dependenzbasierte Modelle

11.3.1 Dependenzgrammatiken

- in Computerlinguistik sind traditionell **Konstituenten-basierte Formalismen dominant** (Chomsky-Tradition Generativer Grammatik)
  - siehe `Stanford PCFG Parser`
- **Dependenzbasierte Syntaxmodelle** werden immer wichtiger
  - siehe u.a. `spaCy`

- Syntaxmodelle von binären Abhängigkeitsrelationen zwischen Wörtern statt Phrasenstruktur-Grammatikregeln (PSG)
- Dependency-Parsing-Modelle können aus **Dependency-Treebanks** induziert werden
  $\rightarrow$ Dependency-Treebanks können handannotiert sein oder abgeleitet aus PSG-Treebank
- Dependenzanalysen können auch **sekundär aus Analysen mit konstituentenbasierten Parsern** erzeugt werden
  $\rightarrow$ z. B. beim Stanford-Parser

11.3.2 Vorteile von Dependenzmodellen

- **Relationale Informationen direkt** vorhanden statt indirekt über Position in Strukturbaum
  $\rightarrow$ Verwendung z. B. für **Informationsextraktion und semantisches Parsing**
- **Wortgrammatik** = direkte Modellierung von Relation zwischen Wörtern
  $\rightarrow$ **keine Lexikalisierung notwendig**
- **Dependenzgrammatik als Wortgrammatik**
  $\Rightarrow$ **reduziert *sparse data*-Problem bei Parameterabschätzung**

11.3.3 *Dependency-Treebanks*

- von Experten erstellte **dependenzsyntaktisch annotierte Korpora**:
  - **relationsannotierte Tokenlisten** = Knoten + Relationen
  - verschiedene Formate: dot-Format, CoNLL-Format
- Einsatz zu **Training und Evaluation** von Dependenz-Parsing-Systemen

Auflistung 4: *dot-Format*

```
digraph G{
1 [label="1 (Hund)"]
2 [label="2 (jagt)"]
3 [label="3 (Katze)"]

2 → 1 [label="nsubj"]
2 → 3 [label="obj"]
}
```

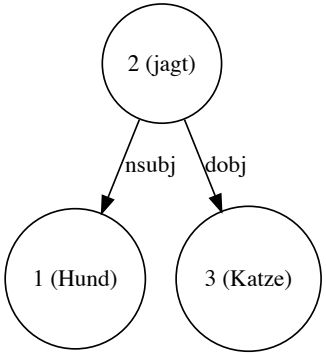


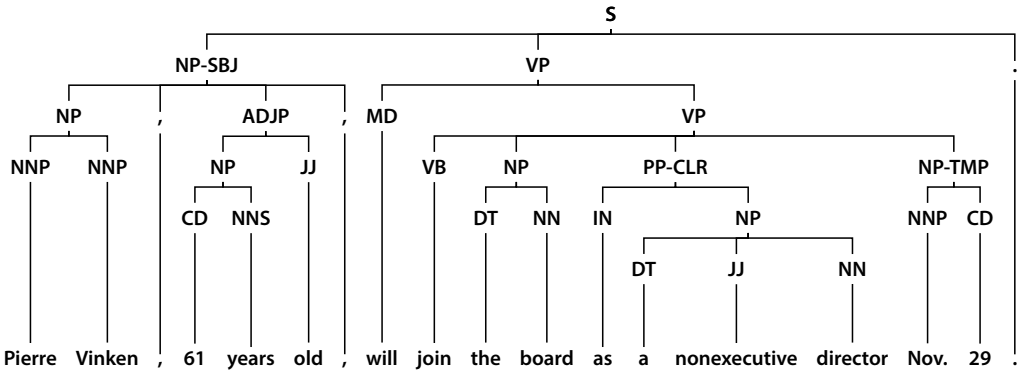
Abbildung 2: *Visualsierung mit graphviz*

Gewinnung *Dependency-Treebanks* aus PSG-Treebanks:

- **Transformation** von **kopfannotierten Konstituenten-Bäumen** in einen Dependenzgraph (s. Sitzung 5):
  1. **Finden aller *head-dependent*-Relationen** über *head-finding-rules*
  2. **Labeln der Relationen** über handgeschrieben Regeln
     $\rightarrow$ Bestimmung Relationstyp **über Strukturposition**:
     `NP mit Mutterknoten S ist subj`
     $\rightarrow$ bei Penn-Treebank: Verwendung **funktionaler Informationen in den Nichtterminalen**: `NP-SBJ`

Funktionale Kategorien in Penn Treebank:

- **Grammatische Relationen/funktionale Angaben** in den phrasalen Kategorien, z. B.: `NP-SBJ`
  → `PP-CLR`: 'closely related', z. B. für präpositionales Objekt
  → `NP-PUT`: adverbiales Komplement von *put*
  → `NP-ADV`: für Kasusadverbial

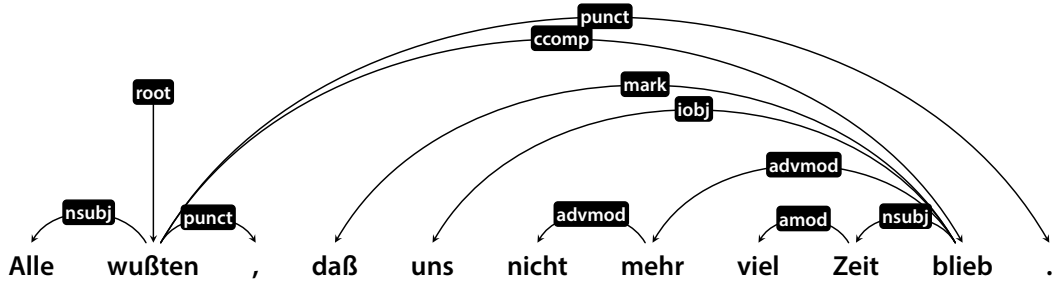


CoNLL-Dependency-Treebanks:

- CoNLL: Shared Tasks zu **Dependency Parsing**: mit annotierten Treebanks für Evaluation der Systeme
- **UD-Treebanks (>30 Sprachen)** im CoNLL-U-Format:
  → `http://universaldependencies.org/format.html`
- **TIGER Dependency Bank** (in Dependency-Format konvertiertes TIGER-Korpus, deutsch) verwendet in CoNLL und UD-Treebanks, konvertiert in Stanford bzw. Universal Dependencies

1	Alle	alle	PRON	PIS	Case=Nom..	2	nsubj	_	_
2	wußten	wissen	VERB	VVFIN	Number=Plur..	0	root	_	SpaceAfter=No
3	,	,	PUNCT	$,	_	2	punct	_	_
4	daß	daß	SCONJ	KOUS	_	10	mark	_	_
5	uns	wir	PRON	PPER	Case=Dat..	10	iobj	_	_
6	nicht	nicht	PART	PTKNEG	Polarity=Neg	7	advmod	_	_
7	mehr	mehr	ADV	ADV	_	10	advmod	_	_
8	viel	viel	ADJ	PIAT	Case=Nom..	9	amod	_	_
9	Zeit	Zeit	NOUN	NN	Case=Nom..	10	nsubj	_	_
10	blieb	bleiben	VERB	VVFIN	Number=Sing..	2	ccomp	_	SpaceAfter=No
11	.	.	PUNCT	$.	_	2	punct	_	_

Tabelle 1: *Satz im CoNLL-Format (deutsches UD-Korpus)*



11.3.4 Statistische Dependency-Parsing-Modelle

1. **Übergangsbasiertes Dependenz-Parsing:**
   → **Stack-basierter Shift-Reduce-Parser**
   → **Auswahl des Übergangs** von einem Zustand (***Konfiguration*** *von Stack, Buffer und erkannten Relationen*) zum nächsten **über Klassifikator**
   → **Klassifikator: bildet Konfigurationen auf Übergänge ab**
   → **trainiert anhand von Dependency-Treebank**

2. **Graphbasiertes Dependenz-Parsing:**

   → **Auswahl von am besten bewerteten Baum** im Graph aller möglichen Relationen zwischen den Wörtern eines Satzes

   → Lernen der **Gewichte der Relationen** anhand von **Dependency-Treebank**

   → Vorteil: **Parsing nicht-projektiver Strukturen** möglich (diskontinuierliche Strukturen)

   → Vorteil: **globale Bewertung der Dependenzstruktur von Sätzen** statt lokaler Entscheidungen

Übergangsbasierte Parsing-Systeme:

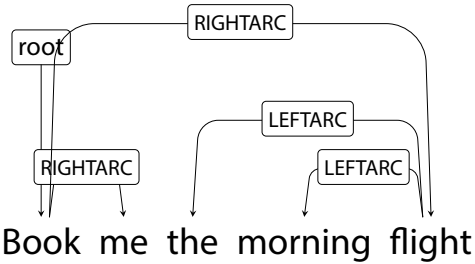
- **Malt-Parser** (Nivre et al.): *transition-based Dependency Parser*
- **Stanford-Dependency-Parser** (Manning et al.):
  - neben der **Transformation von** PCFG-geparsten **Konstituentenbäumen in Dependenzgraphen** (`englishPCFG.ser.gz`):
  - **Transition-based Dependency-Parsing-Modell:**
    $\rightarrow$ `englishFactored.ser.gz`: verwendet PCFG-Parser und Dependenz-Parser und vergleicht Ergebnisse
- **spaCy**: *transition-based* Dependency-Parsing; Modelle gelernt mit neuronalen Netzen (`https://spacy.io/models/#architecture`)

11.3.5 Beispiel: Übergangsbasiertes Dependency-Parsing

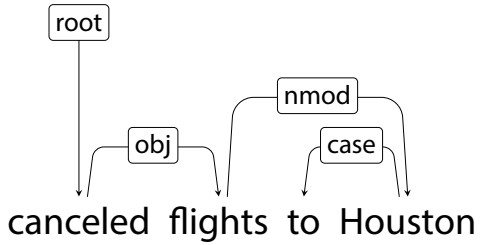
- `SHIFT`-**Operation**: Wörter in **Wortliste** (Buffer) auf Stack
  $\rightarrow$ Stack wird mit `root`-Knoten **initialisiert**
  $\rightarrow$ Abschluss, wenn Wortliste leer und nur noch `root` auf Stack

- `REDUCE`-**Operation**:
  $\rightarrow$ statt Ersatz durch Nonterminal (CFG):
  $\Rightarrow$ **Hinzufügen von Relation zwischen den beiden obersten Elementen auf dem Stack**
  $\Rightarrow$ **Löschen des Dependents vom Stack**

- **2 mögliche `REDUCE`-Operationen** (je nach Position Kopf):
  - **LEFTARC** (Kopf rechts): the $\leftarrow$ flights
  - **RIGHTARC** (Kopf links): book $\rightarrow$ me
- **Einschränkung bei *RIGHTARC***: nur, wenn der Dependent der möglichen Relation nicht Kopf einer der Relationen aus der Menge offener Relationen ist
  $\rightarrow$ Einschränkung verhindert, dass **Wort zu früh vom Stack** genommen wird
  $\rightarrow$ dagegen `LEFTARC`: immer möglich (d.h. nur projektive Strukturen, siehe unten)



Step	Stack	Word List (Buffer)	Transition	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book $\rightarrow$ me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning $\leftarrow$ flight)
7	[root, book, the, flight]	[]	LEFTARC	(the $\leftarrow$ flight)
8	[root, book, flight]	[]	RIGHTARC	(book $\rightarrow$ flight)
9	[root, book]	[]	RIGHTARC	(root $\rightarrow$ book)
10	[root]	[]	Done	



	Stack	Word List (Buffer)	Transition	
	[root,canceled,flights]	[to, Houston]	**SHIFT oder RIGHTARC ?**	
mögliche Übergänge:				Relation Added
SHIFT	[root,canceled,flights,to]	[Houston]		-
RIGHTARC	[root,canceled]	[to, Houston]		(canceled → flights)

- **richtiger Übergang:** `SHIFT`
  → bei `RIGHTARC` wird *flights* zu früh vom Stack entfernt; Relation (*flights* → *Houston*) wäre dann nicht mehr möglich

11.3.6 Evaluation von Dependenz-Parsing-Systemen

- **Überprüfung an Testmenge** (Teilmenge Dependency-Treebank)
- ***unlabeled attachment accuracy***: **korrekte Zuweisung Dependent zu Kopf**
- ***labeled attachment accuracy***: **korrekte Zuweisung** und **korrekte Relation** zwischen Dependent und Kopf

11.3.7 Regelbasierte Dependenzgrammatiken

- **von Experten erstellte** Dependenzgrammatiken
- wichtige Dependengrammatik-Formalismen: **Bedeutung-Text-Modell** (I.A. Meľčuk), **Word Grammar**, **Link Grammar**, **Constraint-Grammar**

- Modellierung über (lexikalisierte) **kontextfreie Grammatiken**
  $\rightarrow$ nur projektive Strukturen möglich
- Modellierung über **Constraint-basierte Dependenzgrammatiken**
  $\rightarrow$ **Angabe von Wohlgeformtheitsbedingungen**
  $\rightarrow$ **Entfernung von Constraint-verletzenden Graphen** im Parsing
- **Constraint-Parsing**: Verarbeitung von **nicht-projektiven Strukturen**

Auflistung 5: *Dependenzgrammatik*

```
grammar = nltk.DependencyGrammar.fromstring("""
    'shot' → 'I' | 'elephant' | 'in'
    'elephant' → 'an' | 'in'
    'in' → 'pajamas'
    'pajamas' → 'my'
    """)

parser =
  nltk.ProjectiveDependencyParser(grammar)

for tree in parser.parse(sent):
    print(tree)
    tree.draw()
#(shot I (elephant an (in (pajamas my))))
#(shot I (elephant an) (in (pajamas my)))
```

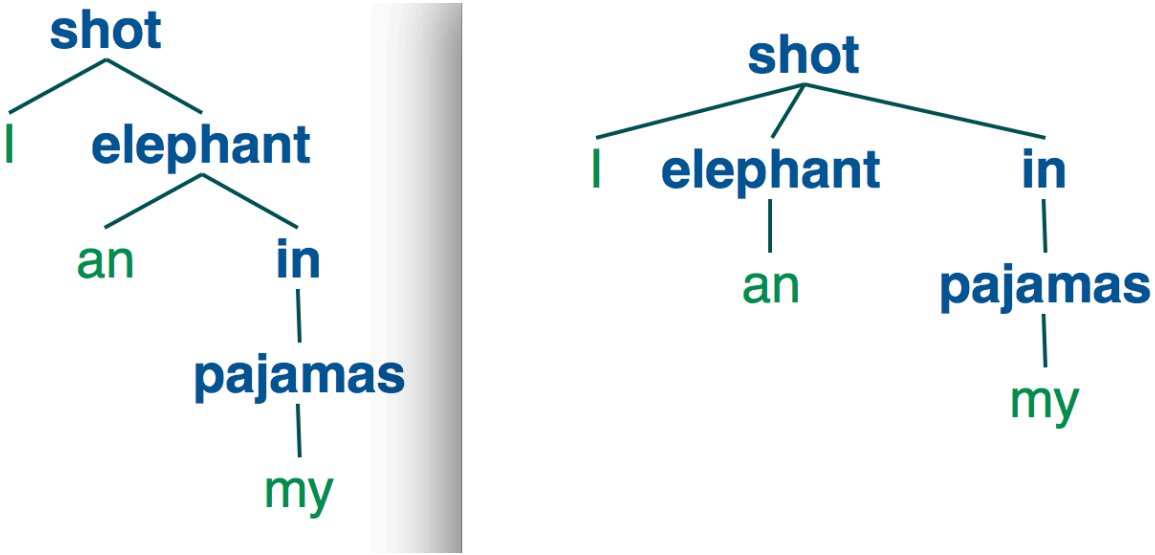
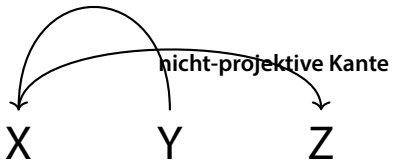
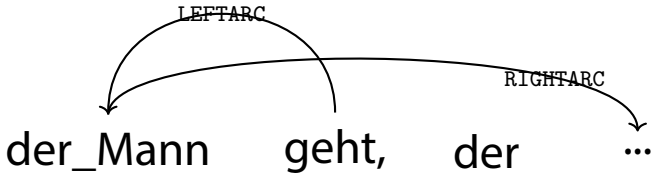


Abbildung 3: *Syntaxbäume Dependenzanalyse (Stemmas)*

Nichtprojektivität

- **projektive Struktur:** alle Kanten sind **projektiv, d. h. es gibt einen Pfad vom Kopf der Relation zu jedem Wort zwischen Kopf und Dependent**
- **nicht-projektive Struktur: Überschneidung von Kanten**
  $\rightarrow$ z. B.: ***Dependent eines Wortes folgt nach dessen Kopf***





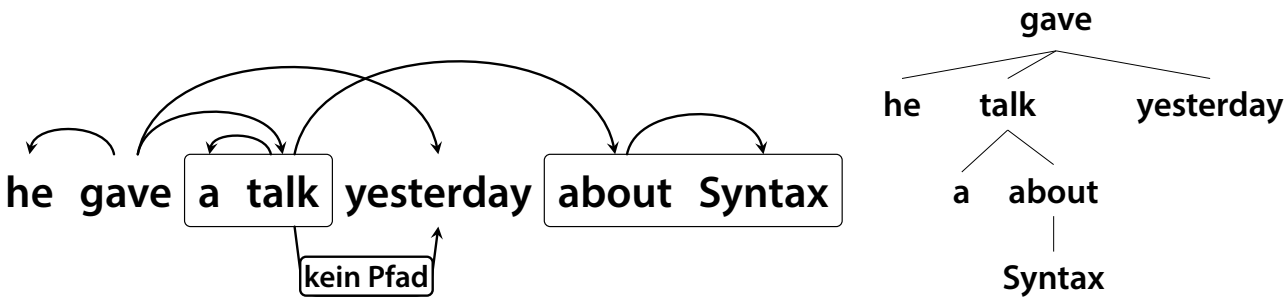


Abbildung 4: *Dependenzanalyse diskontinuierlicher = nicht-projektiver Struktur (mit und ohne Berücksichtigung linearer Ordnung)*

- **linguistisch: nicht-projektive Strukturen** entstehen durch **diskontinuierliche Elemente**
  $\rightarrow$ **freie Wortstellung und *long distance dependencies***

- Dependenzgrammatiken sind **besser** als Konstituentengrammatiken **geeignet**, **diskontinuierliche Strukturen abzubilden**
  → Modellierung **relationaler Struktur**, nicht der linearen Anordnung
  → Dependenzstruktur **abstrahiert von der linearen Anordnung**
  → **bei Verarbeitung** (Parsing) können **nicht-projektive Strukturen aber problematisch** sein

- bei **Ableitung Dependenzgrammatik** von PSG-Treebanks durch *head-finding-rules* ergeben sich **automatisch projektive** Strukturen