Earley-Parsing

Übersicht

- Aktives Chart-Parsing für kontextfreie Grammatiken
- Eigenschaften von Earleys Verfahren
- Einzelne Komponenten
  - Initialisierung
  - Predictor
  - Scanner
  - Completer
- Implementation in Prolog
- Ende und Resultat der Analyse

Eigenschaften I

Von Jay Earley wurde 1970 ein Chart-Parsing-Verfahren für kontextfreie Grammatiken vorgestellt. Es…

- parst Sätze mit n Wörtern im schlechtesten Fall in einer Zeit proportional zu n^3 – Verfahren gehört zur Komplexitätsklasse $O(n^3)$
- kommt mit den häufig problematischen Regeln zurecht.
  - linksrekursive Regeln (direkt oder indirekt)
  - Tilgungsregeln (Code siehe Übungen)
- verwendet eine aktive Chart.
- benützt eine gemischte Analysestrategie
  - von links nach rechts
  - top-down und bottom-up

Eigenschaften II

Der Earley-Algorithmus

- verfolgt jeweils für eine Satzposition alle syntaktischen Alternativen gleichzeitig.
- hat am Ende des Parsings sämtliche alternativen Syntaxanalysen in der Chart.
- ist leicht in imperativen Programmiersprachen (C, Java etc.) zu implementieren.
- ist nützlich und wichtig bei praktischen Anwendungen der Computerlinguistik.
- erhielt seit 1970 zahlreiche Verbesserungsvorschläge gegenüber Earleys Original-Verfahren.

Earley-Algorithmus

Das Verfahren von Earley besteht im wesentlichen aus drei Komponenten

- **Predictor**: Schlägt passende aktive Kanten vor.
- **Scanner**: Akzeptiert Wörter der Eingabekette.
- **Completer**: Versucht aktive Kanten zu vervollständigen, d.h. passiv zu machen.

Nach der Initialisierung werden diese drei Komponenten für jeden Knoten in der Chart aufgerufen,

- jeweils solange, bis keine neuen Kanten mehr hinzukommen.

Beispiel-Grammatik

Folgende Phrasenstruktur-Grammatik mit Lexikon sei vorausgesetzt:

```
rule(s,  [np,vp]).
rule(np, [np,conj,np]).
...
```

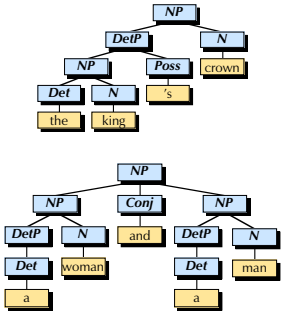
```
word(det, the).
word(det, a).
...
```

S → NP VP
NP → NP Conj NP
NP → DetP N
DetP → Det
DetP → NP Poss

Syntax

Det → a | the
N → man | woman |…
V → sleeps | chases |…
Conj → and
Poss → 's

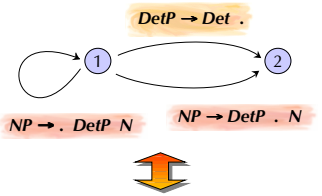
Lexikon



linguistische Motivation für links-rekursive Grammatikregeln

Die Chart

- **edge(*From, To, C, RC, SC*)**
  - Dynamische edge/5-Fakten speichern aktive und passive Kanten
    1. *From:* Startposition
    2. *To*: Endposition
    3. *C*: Kategorie
    4. *RC*: Liste der erkannten Kategorien (Aus technischen Gründen in umgekehrter Reihenfolge.)
    5. *SC*: Liste der noch zu erkennende Kategorien



NP → DetP . N
<1,2> NP → DetP . N

Zwischen Position 1 und 2 ist eine (unvollständige) NP, von der bereits die DetP gefunden wurde, der aber zur Vollständigkeit noch ein N fehlt.

```
edge(1, 1, np, [], [detp,n]).
edge(1, 2, np, [detp], [n]).
edge(1, 2, detp, [det], []).
```


Charteinträge einfügen

- **store/1: Einfügen von Kanten**
  - Identische Kanten dürfen nicht mehrfach in die Chart eingetragen werden, sonst gäbe es unerwünschte Mehrfachlösungen.
  - Eine Kante wird gespeichert, d.h. der Chart hinzugefügt, falls sie nicht schon gespeichert ist.

```
:- dynamic edge/5.

store(Edge) :-
    \+ call(Edge),
    asserta(Edge).
```

  - Falls *call(Edge)* scheitert, wird *Edge* assertiert.
  - Falls *call(Edge)* gelingt, wird *Edge* nicht assertiert.

Gerüst des Earley-Algorithmus

Einen Satz earley-parsen heisst,

1. Chart löschen
2. Chart bezüglich Startsymbol initialisieren
3. von den 3 Komponenten verarbeiten lassen
4. prüfen, ob in der Chart eine passive Kante mit dem Startsymbol zu finden ist

```
parse(C, S) :-
   retractall(edge(_,_,_,_,_)),
   init(C),
   process(S, 1, LastPos),
   edge(1, LastPos, C, _, []).
```

Von den 3 Komponenten verarbeiten lassen heisst,

- Abbruch, falls Eingabekette leer ist.
- Rekursives Abarbeiten durch
  1. Predictor
  2. Scanner
  3. Completer

und wieder von vorn, ein Wort weiter.

```
process([], LastPos, LastPos).
process([Word|Words], Pos, LastPos) :-
   predictor(Pos),
   scanner(Word, Pos, NextPos),
   completer(NextPos),
   process(Words, NextPos, LastPos).
```

Legende zur Metasymbolik

Metasymbole für einzelne Nicht-Terminalsymbole

A, B, C

Metasymbole für beliebig lange Ketten von Grammatik-symbolen

α, β, γ

- Die Kette kann auch leer sein! D.H. ε.

Beispiele

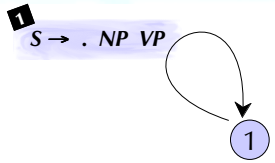
- Die gepunktete Regel $S \rightarrow \bullet\ NP\ VP$ ist von der Form $A \rightarrow \bullet\ \alpha$, gdw. $A = S$ und $\alpha = NP\ VP$.
- Die gepunktete Regel $NP \rightarrow NP \bullet Conj\ NP$ ist von der Form $C \rightarrow \alpha \bullet B\ \gamma$, gdw. $C = NP$, $\alpha = NP$, $B = Conj$ und $\gamma = NP$.
- Die gepunktete Regel $NP \rightarrow NP\ Conj \bullet NP$ ist von der Form $C \rightarrow \alpha \bullet B\ \gamma$, gdw. $C = NP$, $\alpha = NP\ Conj$, $B = NP$ und $\gamma = \varepsilon$.

Initialisierung der Chart

init/1 fügt für jede Regel mit dem Startsymbol als LHS eine aktive Kante ein.

Initialisierungsregel

- Falls C die gesuchte Kategorie ist, füge für jede Grammatikregel der Form $C \rightarrow \gamma$ die Kante $\langle 1, 1 \rangle\ C \rightarrow \bullet\ \gamma$ in die Chart.




```
edge(1, 1, s, [], [np,vp]).
```

```
init(C) :-
  foreach(
    rule(C, RHS),
    store(edge(1,1,C,[],RHS))
  ).
```


foreach/2

Deterministisches Metaprädikat

- **foreach(*B*, *A*)**
  - So oft die Bedingung *B* erfüllt ist, führe die Aktion *A* genau einmal durch.
  - *B* und *A* können trotz Metaprädikate durch gemeinsame Variablen "kommunizieren".
  - Verwendet once/1 als Hilfsprädikat.
  - Emuliert Iteration (while-Schlaufe).

```
foreach(B, A) :-
   call(B),
   once(A),
   fail.

foreach(_, _).
```

- **once(*Ziel*)**
  - Beweise *Ziel* höchstens einmal.

```
once(Goal) :-
   call(Goal),
   !.
```


Predictor-Beschreibung

Der Predictor sagt voraus, welche Grammatikregeln zum Ziel führen können.

- Aktive Konstituenten werden expandiert $S \rightarrow \bullet\ NP\ VP$

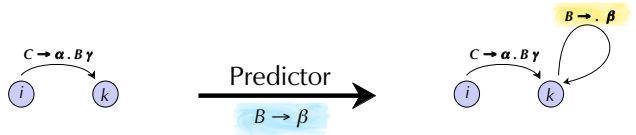
  Die aktive Konstituente steht direkt hinter dem Punkt.
- Vorgehensweise des Predictors: Top-Down

Prediktorregel für die Position *k*

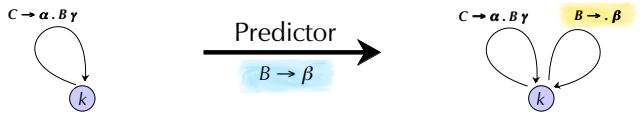
- Schritt I: Für jede Kante der Form $\langle i,k \rangle\ C \rightarrow \alpha \bullet B\ \gamma$ und jede Regel der Form $B \rightarrow \beta$, füge eine Kante $\langle k,k \rangle\ B \rightarrow \bullet\ \beta$ ein.
- Schritt II: Wende die Prediktorregel auf das Resultat ihrer eigenen Anwendung an.

Predictor-Regel graphisch

- **Falls $i < k$:**



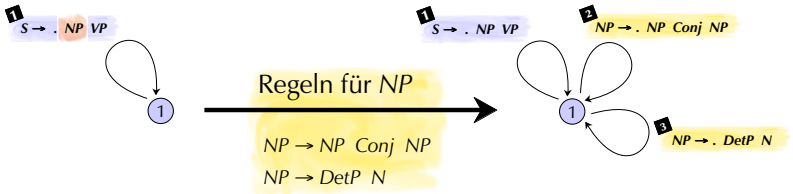
- **Falls $i=k$:**



Predictor: Beispiel Schritt I

Predictor für Position 1

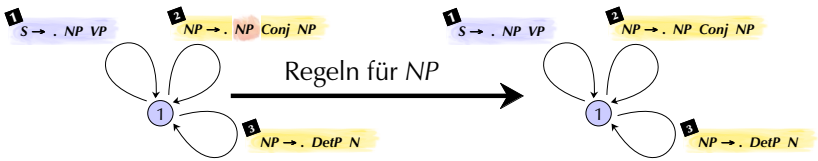
- Weil eine aktive Kante an Position 1 die aktive Konstituente *NP* hat, werden anhand der NP-Regeln Kanten in die Chart eingefügt.
- Da unsere Grammatik zwei Regeln für NPs enthält, werden entsprechend zwei aktive Kanten eingefügt.



Predictor: Beispiel Schritt II

Predictor für Position 1

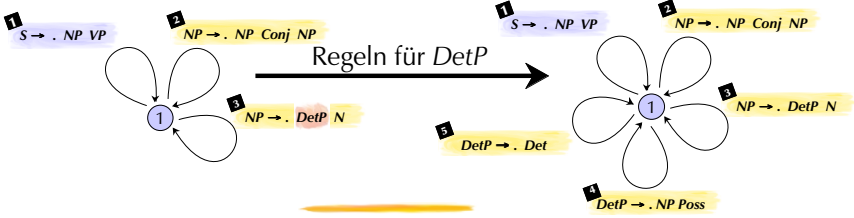
- Die frisch eingefügte Kante 2 hat wiederum eine *NP* als aktive Konstituente.
  - Daher sollen anhand der NP-Regeln neue Kanten in die Chart eingefügt werden.
- Allerdings kommen so keine *neuen* Kanten in die Chart, da alle aktiven Kanten für NPs schon drin sind!
  - Keine endlose Rekursion, trotz linksrekursiver Grammatik!



Predictor: Beispiel Schritt II

Predictor für Position1, Schritt II

- Die frisch eingefügte Kante Nr. 3 hat eine *DetP* als aktive Konstituente.
  - Daher sollen anhand der *DetP*-Regeln neue Kanten in die Chart eingefügt werden.
- Die Grammatik enthält zwei Regeln für *DetP*, also kommen zwei neue aktive Kanten zur Chart hinzu.
  - Dann ist für den Predictor nichts mehr zu tun.



Predictor in Prolog

- **predictor/1**
  - Mache für jede Kante mit einer aktiven Konstituente an der Endposition die Voraussagen.

```
predictor(To) :-
  foreach(
    edge(_, To, _, _, [Active|_]),
    predict(Active, To)
  ).
```

- **predict/2**
  - Schritt I: Finde eine Regel, die die vorherzusagende Kategorie als LHS hat, merke die linke Ecke als neue aktive Konstituente. Speichere die entsprechende Kante in der Chart, oder scheitere, falls sie schon gespeichert ist.
  - Schritt II: Wende predict/2 rekursiv auf die neue aktive Konstituente an.

```
predict(C, Pos) :-
  rule(C, [Active2|Rest]),
  store(edge(Pos,Pos,C,[],[Active2|Rest])),
  predict(Active2, Pos),
  fail.

predict(_, _).
```


Scanner-Beschreibung

Der Scanner nimmt Terminal-Symbole und bestimmt ihre lexikalischen Kategorien.

- könnte z.B. Schnittstelle zu Lexikon/Morphologie aufrufen
  - Hier: Aufruf des spartanischen Lexikons in word/2

N → man
N → woman

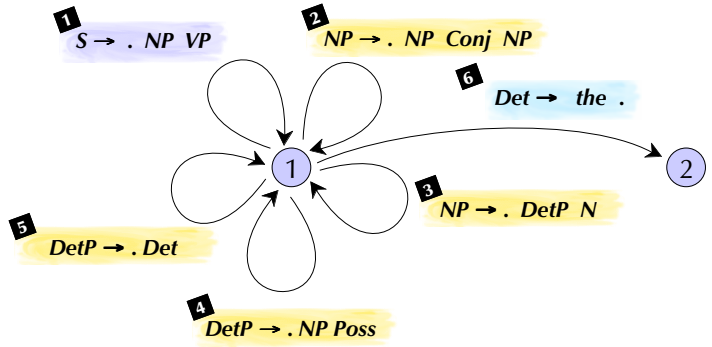
```
word(n, man).
word(n, woman).
...
```

Scanner-Regel für Position *k*

- Falls das Wort *W* zwischen der Position *k* und *k*+1 der Eingabekette steht, füge für jede Lexikon-Regel der Form *C* → *W* die Kante ⟨*k*, *k*+*1*⟩ *C* → *W*. in die Chart.

Scanner: Beispiel

Chart(-Ausschnitt) nach Laufen des Scanners für Knoten 1:



Scanner in Prolog

- **scanner(+Wort, +K, ?Kplus1)**
  - Für jede mögliche Kategorie, wird eine Kante mit der entsprechenden Endposition gespeichert.

```
scanner(Word, From, To) :-
  To is From + 1,
  foreach(
    word(Cat, Word),
    store(edge(From,To,Cat,[Word],[]))
  ).
```

  - Scanner bekommt das Wort von process/3 geliefert.
  - Verbesserung gegenüber der Original-Version von Earley:
    - Earley hätte Präterminal-Regeln wie *Det* → *the* vom Predictor top-down verarbeiten lassen. Weil damit Terminale vorausgesagt werden, die gar nicht Teil der Eingabekette sind, ist das ineffizient – wie beim Standard-DCG-Parser.

Completer-Beschreibung

Der Completer vervollständigt aktive Kanten durch Kombination mit passiven Kanten.

- Anschaulich: Der Punkt wandert in der Regel eine Kategorie nach rechts.
- Vorgehensweise des Completers: Bottom-Up

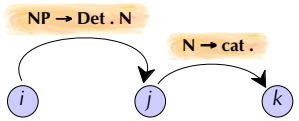
Completer-Regel für Position *k*

- Schritt I:
  Für jede passive Kante der Form $\langle j, k\rangle\ B \rightarrow \beta\ .$ und jede aktive Kante der Form $\langle i, j\rangle\ C \rightarrow \alpha\ .\ B\ \gamma$ füge die Kante $\langle i, k\rangle\ C \rightarrow \alpha\ B\ .\ \gamma$ zur Chart hinzu. (sog. **Fundamentalregel**)
- Schritt II: Wende die Completer-Regel auf das Resultat ihrer eigenen Anwendung an.

Fundamental-Regel des Chart-Parsing

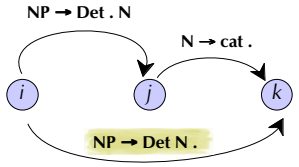
Wenn folgende Kanten in der Chart sind:

- Passive Kante zwischen Positionen *j* und *k*:
  $B \rightarrow \beta\ .$ d.h. $N \rightarrow cat\ .$
- Aktive Kante zwischen Positionen *i* und *j*:
  $C \rightarrow \alpha\ .\ B\ \gamma$ d.h. $NP \rightarrow Det\ .\ N$



Dann füge folgende Kante zur Chart hinzu:

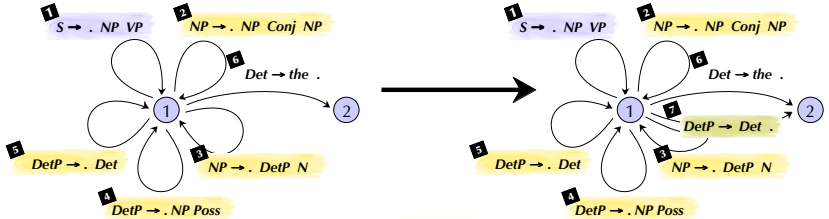
- Aktive oder passive Kante zwischen Positionen *i* und *k*:
  $C \rightarrow \alpha\ B\ .\ \gamma$ d.h. $NP \rightarrow Det\ N\ .$



Completer: Beispiel

Completer für Position 2, Schritt I

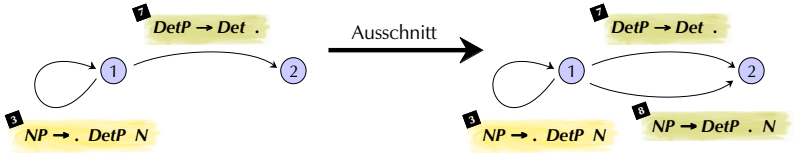
- Passive Kante 6 endet in Position 2. Ihre Kategorie ist *Det*.
- Suche aktive Kanten,
  - die ein *Det* benötigen
  - und am Startknoten der passiven Kante enden (Position 1).
- Kante 5 passt.
  - Füge durch Fundamentalregel die neue Kante (7) in die Chart.



Completer: Beispiel

Completer für Position 2, Schritt II

- Passive Kante 7 endet hier. Ihre Kategorie ist *DetP*.
- Suche aktive Kanten,
  - die ein *DetP* benötigen
  - und am Startknoten der passiven Kante enden (Position I)
- Kante 3 passt
  - füge durch Fundamentalregel neue Kante (8) zur Chart.



Completer in Prolog

completer/1

- Vervollständige die Chart mit jeder passiven Kante, die bei Position *K* endet und bei *J* beginnt.

complete/3

- Schritt I: Suche aktive Kante, die bei *J* endet und Kategorie *B* benötigt. Wende die Fundamentalregel an und speichere die neue Kante.
- Schritt II: Falls sich bei Schritt I eine passive Kante ergeben hat, vervollständige diese rekursiv.
  - *Achtung: Die erkannten Konstituenten sind verkehrt herum abgespeichert…*

```
completer(K) :-
  foreach(
    edge(J, K, Passive, _, []),
     complete(J, K, Passive)
  ).
```

```
complete(J, K, B) :-
   edge(I, J, A, Aphla, [B|Gamma]),
   store(edge(I,K,A,[B|Aphla],Gamma)),
   Gamma == [],
   complete(I, K, A),
   fail.

complete(_, _, _).
```


Analyse: Fortschritt und Ende

Fortschritt

- Wenn der Completer für Knoten 2 nicht mehr weiterkommt …
  - Predictor für Knoten 2 – Scanner für Knoten 2
  - Completer für Knoten 3 – Predictor für Knoten 3 – Scanner für Knoten 3
  - Completer für Knoten 4 – …

Ende

- Wenn die Eingabekette leer ist und alle Kanten vervollständigt sind, ist die syntaktische Analyse beendet

Ergebnis der Analyse

- Die Eingabe-Kette ist genau dann akzeptiert, wenn gilt:
  I. eine passive Kante reicht von der ersten bis zur letzten Position
  II. und die Kategorie dieser Kante ist das Startsymbol.

Literatur

Originalaufsatz

- Earley, Jay(1970): An efficient context-free parsing-algorithm. Communications of the ACM, 14, pp. 453-60. Reprinted in "Readings in Natural Language Processing (B. J. Gross, K. Spark et al. eds), pp. 25-33, Morgan Kaufmann: Los Altos, 1986.

Implementation

- nahe bei Covington, M. (1994: 176ff.)
- aber beeinflusst von Gazdar, G./ Mellish, Ch. (1989: 179ff.): Natural Language Processing in PROLOG