Institut für Computerlinguistik, Uni Zürich: Effiziente Analyse unbeschränkter Texte

Vorlesung 7: Ein effizienter CYK Parser

Gerold Schneider

Institute of Computational Linguistics, University of Zurich
Department of Linguistics, University of Geneva
gschneid@ifi.unizh.ch

December 1, 2003

Contents

1 Dependency Grammar (DG) and Chomsky Normal Form (CNF)

In a binary CFG, any two constituents A and B which are adjacent during parsing are candidates for the RHS of a rewrite rule. In DG and Bare Phrase Structure, one of these is isomorphic to the RHS, i.e. the head.

$$B \rightarrow AB, e.g.\_NN \rightarrow \_DT\_NN \quad (1)$$

$$A \rightarrow AB, e.g.\_VB \rightarrow \_VB\_PP \quad (2)$$

- All relations are between 1 mother and 1 daughter $\rightarrow$ binary rules
- No null dependents (empty elements)
- No cycles
- All rules are thus in Chomsky Normal Form (CNF)

2 Bottom-up Chart Parsing

Def.: $[From, CAT, To]$ is an edge of category CAT from sentence position $From$ to sentence position To.

General idea:

1. Dynamic programming: fill in tables of partial solutions to sub-problems until they contain all the solutions needed to solve the problem.

2. Build lexical edges $[n, Tag, n]$ for each word in the sentence $1..n..m$ and add them to the chart

3. Combine edges according to syntax rules in all permissible ways

4. Proceed step-wise from shorter to longer structures until $[1, \_, m]$ has been found

2.1 CYK Characteristics

- Structure length is monotonically increasing: every result of a combination is longer than the combined elements

  This means that if we start at the shortest edges (the lexical items) and at every repeated step search for edges that are 1 word longer we never have to backtrack.

  `for` j `= 2 to` n `# length of edge`

- An edge can start anywhere, at the latest at $n - j$.

  `for` i `= 1 to` $n-j+1$ `# beginning of edge`

- Binary non-empty rules mean that every edge $1..(i+j)$ can be separated in exactly $(i+j) - 1$ ways, e.g. $1..4$:

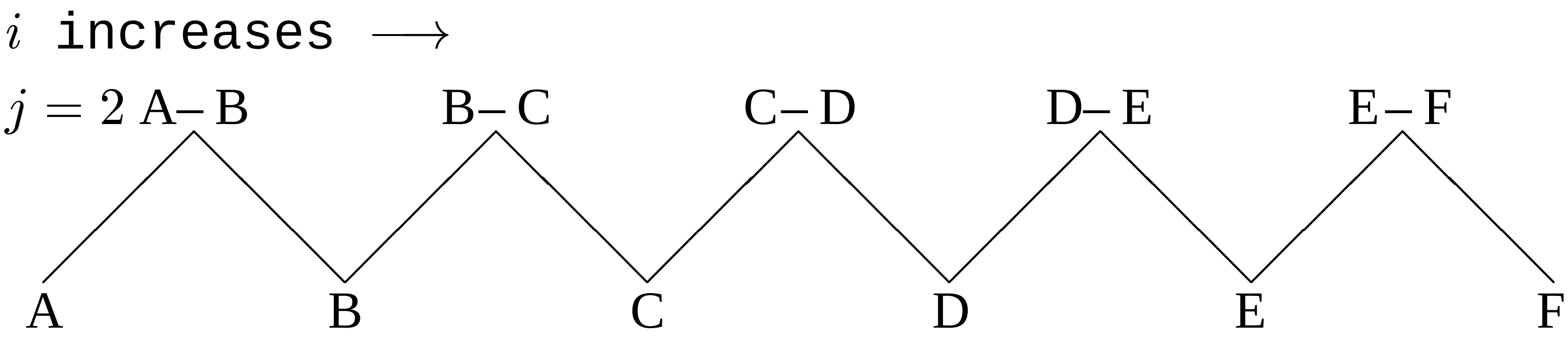
  $$\begin{array}{llll} 1\,2\,3\,4 \longrightarrow & 1 & | & 2\,3\,4 \\ & 1\,2 & | & 3\,4 \\ & 1\,2\,3 & | & 4 \end{array}$$

  `for` k `=` $i+1$ `to` $i+j-1$ `# separator position`

2.2 The CYK algorithm, step 1: j=2

- CYK Parsing: bottom-up parallel processing, passive chart

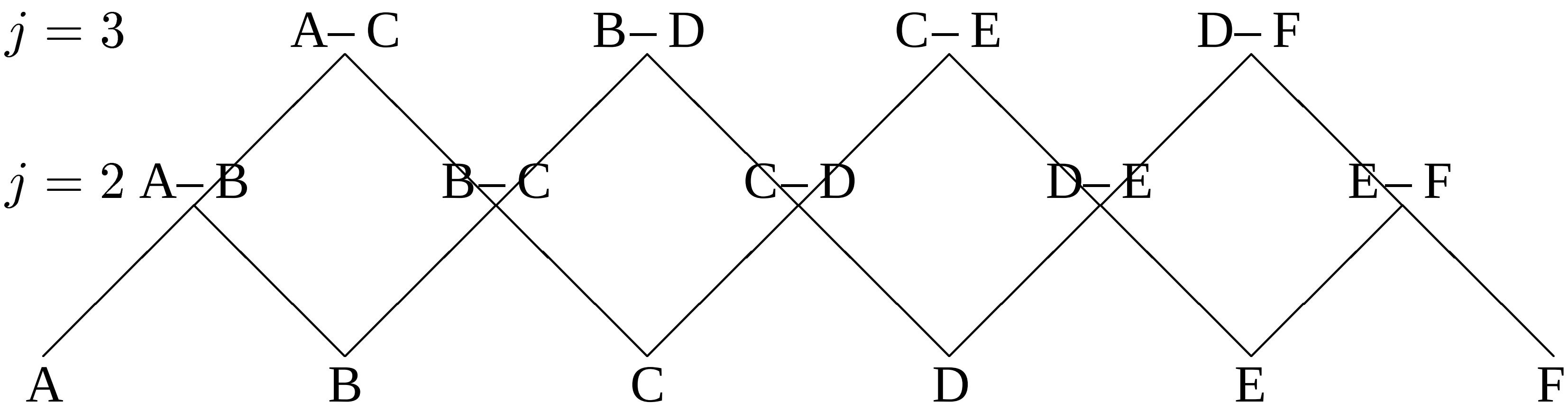
for j = 2 to N # length of span
  for i = 1 to $N - j + 1$ # begin of span
    for k = $i + 1$ to $i + j - 1$ # separator position
      if $Z \rightarrow XY$ and $X \in [i-k], Y \in [k-j]$
      and $Z \notin [i-j]$
      then insert Z at $[i-j]$



2.3 The CYK algorithm, step 2: j=3

- CYK Parsing: bottom-up parallel processing, passive chart

for j = 2 to N # length of span
  for i = 1 to $N - j + 1$ # begin of span
    for k = $i + 1$ to $i + j - 1$ # separator position
      if $Z \to XY$ and $X \in [i - k], Y \in [k - j]$
      and $Z \notin [i - j]$
      then insert Z at $[i - j]$



2.4 The CYK algorithm, step 2: j=3, possible Cell entries

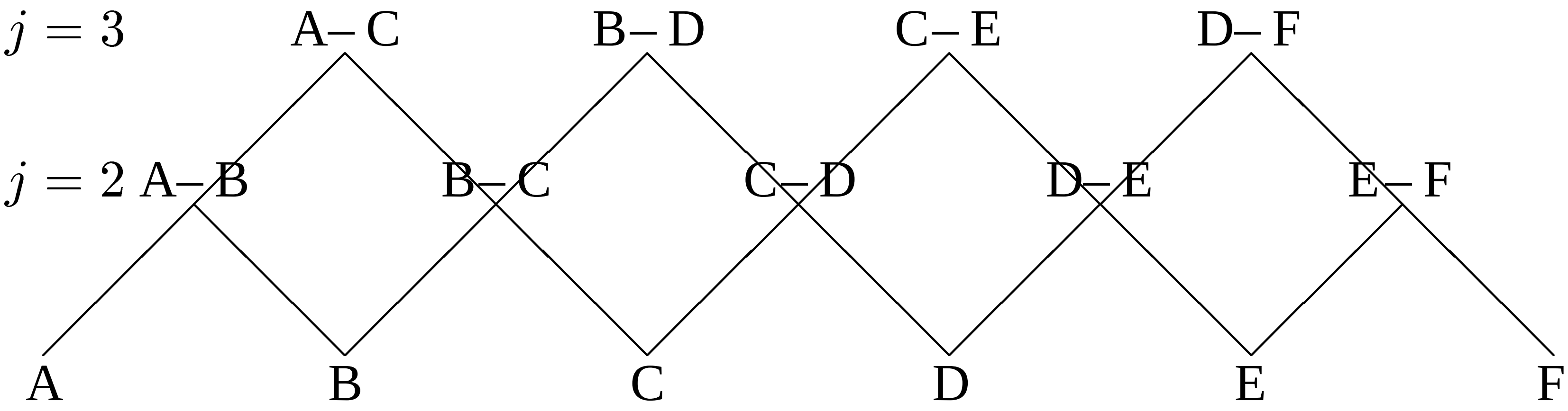
- CYK Parsing: Building up the structure

  `for` j `= 2 to` N `# length of span`

  `etc.`

$$A + (B(C)) = A(B(C)) or B(A, C)$$
$$A + (C(B)) = A(C(B)) or C(A, B)$$
$$C + (A(B)) = C(A(B)) or A(B, C)$$
$$C + (B(A)) = C(B(A)) or B(A, C)$$

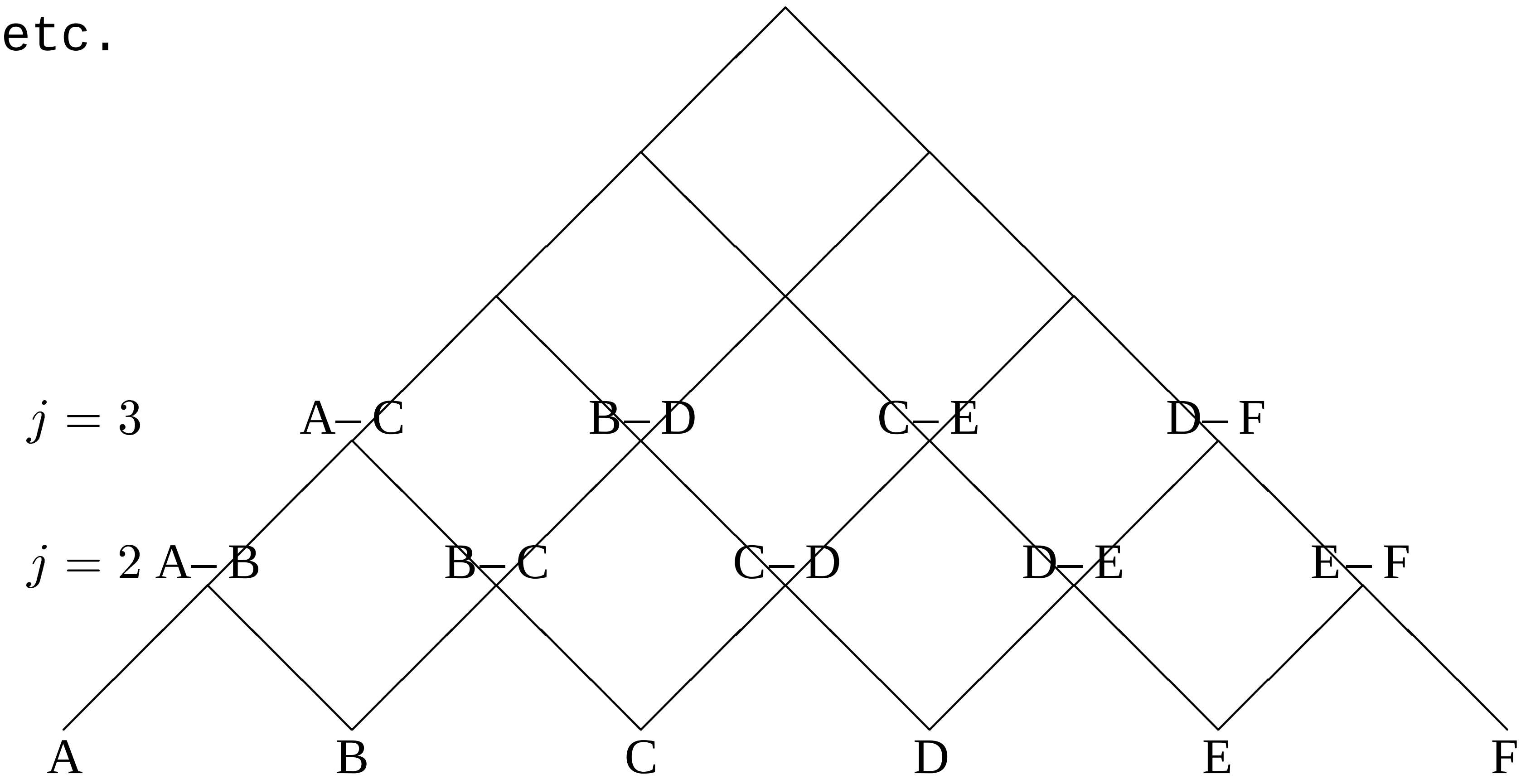
$$\Downarrow$$



2.5 The CYK algorithm, continued

- CYK Parsing: The analysis matrix

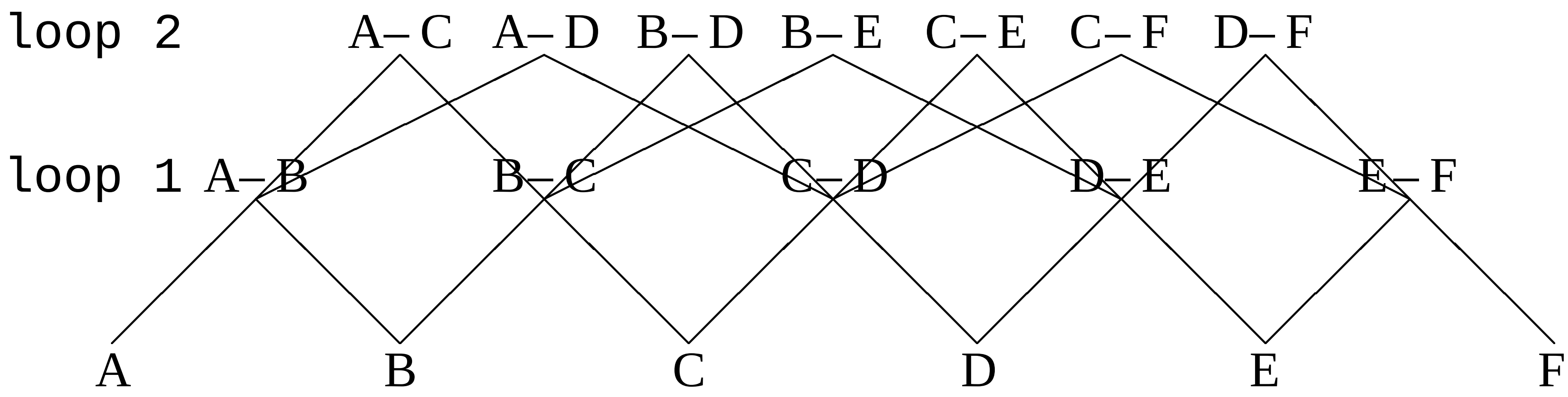
```
for j = 2 to N                # length of span
etc.
```



3 A Prolog CYK Parser

- "chartdata-driven" CYK implementation

1. Add all terminals to chart
2. Loop: `foreach chart entry` $X\lambda i.\lambda k.[i-k]$
   `foreach chart entry` $Y\lambda j.[k-j]$ `# adjacent`
   `if` $\neg$ `tried(`X,Y`)`
   `foreach` $Z \rightarrow X,Y$ `assert` $Z[i-j]$ `to chart (for next Loop)`
   `else assert tried(`X,Y`)`
3. If any rule was successful, prune and then Loop again, else terminate.



3.1 Simplified CYK Parser

```
nextlevel(L) :-
  chart( FID,[FPos,Ffrom-Fto,FScore],[[F,Ftag,FType]],FuncF),
  % foreach chart entry (by backtrack)
  Gto is Ffrom-1,
  chart( GID,[GPos,Gfrom-Gto,GScore],[[G,Gtag,GType]],FuncG),
  % find adjacent chart entries
  bagof(_,sparse(FID,[FPos,Ffrom-Fto,FScore],[[F,Ftag,FType]],FuncF,
                 GID,[GPos,Gfrom-Gto,GScore],[[G,Gtag,GType]],FuncG),_),
                 % parse (non-recursive, chart version)
  write('End of Level '), write(L), write(' reduced items : '),
  perlevel(X), write(X), X > 0, % only recurse successful
  prune(L,X),
  L1 is L+1,
  retractall(perlevel(_)), assert(perlevel(0)),
  nextlevel(L1).
```

3.2 The CYK parsing version of the `sparse` predicate

Fail-driven, non-recursive. A simplified but still too complex version:

```
sparse(FID,[FPos,Ffrom-Fto,FScore],[[F,Ftag,FType]],FuncF,
       GID,[GPos,Gfrom-Gto,GScore],[[G,Gtag,GType]],FuncG) :-
  (tried(FID,GID) -> !, fail; assert(tried(FID,GID))), % already tried
  ... % get various context info for the grammar rules:
  head(Ftag,Gtag,l,Type,Transtag,[FChunk,GChunk,FF,FG,OF,OG],FPos-GPos),
  % rule for LEFT reduction (l)
  (statschart(Ftag,FF,SF,Gtag,FG,SG,Type,Prob,Percent,Dist,FChunk,OG) -> true ;
    (stats(Ftag,FF,SF,Gtag,FG,SG,Type,Prob,Percent,Dist,FChunk,OG),
     assert(statschart(Ftag,FF,SF,Gtag,FG,SG,Type,Prob,Percent,Dist,FChunk,OG)))),
  (Prob < 0.01 -> fail; true), %% early exclusion
  %% Build Func-Struc:
  ...
  asserta(chart(ID,[[FF,Ftag,FChunk,FID,FScore],[FG,Gtag,GChunk,GID,GScore]],
    [FPos,GPos,Gfrom-Fto,PScore,DLen],[[FF,Transtag,Type,ID]],FuncFTRes,Level)),
  retract(perlevel(X)), X1 is X+1, assert(perlevel(X1)),
  (Prob > 0.98 -> !; true), %% early commitment
  fail.
```

4 Differences between CYK and Earley

1. No need for CNF in Earley
2. Empty nodes are allowed
3. Also active edges are entered into the chart: very costly
4. CYK uses a *scanner* and *completer* much like Earley, but no *predictor*:
    - Scanner: move one constituent to the right
    - Completer: Rules are binary $\rightarrow$ always complete any two constituents
    - Predictor: No top-down prediction. CYK is pure bottom-up
    - Earley parsing can also be visualized/processed in a CYK matrix

5 Differences between CYK and Shift-Reduce

see also last lecture

1. No need for CNF in Shift-Reduce
2. No chart, although extensions possible: costly
3. CYK uses a *shift* and *reduce* much like Shift-Reduce, but in parallel
   - Shift: move one constituent to the right
   - Reduce: Rules are binary $\rightarrow$ always reduce any two constituents
   - Try shift and reduce simultaneously