CS344 - CNF + CYK

Note Title 2/1/2012

Announcements

- No office hours today
  check website about tomorrow
- HW is due Friday

Context Free Languages - CFL

Described in terms of productions
(Called Backus-Naur Form, or BNF)

- A set of terminals T
- A set of non-terminals N
- A start symbol S (a non-terminal)
- A set of productions

Ex :

$$S \rightarrow ASA \mid aB$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b \mid \varepsilon$$

Chomsky Normal Forms CNF

Each rule in the grammar is either:

- $A \rightarrow BC$
  where neither B or C is the start variable, & both are nonterminals
- $A \rightarrow a$
  where a is a terminal
- No useless symbols
- only a is at start state

Why CNF?

Parsing: building those parse trees we saw

In general, there are an exponential number of parse trees for a given input.

Given CNF, can get a polynomial time algorithm ($O(n^3)$) to generate a valid parse tree.

Thm: All grammars can be converted to CNF.

Procedure:

First eliminate useless rules.

- Start from the start state, expand set of "reachable states"
- start from terminating rules work backwards

Introduce "dummy" start state

$S_0 \rightarrow S$

Ex: $S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S \mid \varepsilon$

$B \rightarrow b \mid \varepsilon$

$D \rightarrow \varepsilon$

(2) Nullable variables: $B \to \varepsilon$
(only allowed for S_0 = start state in CNF)

Remove all ε productions:

$S_0 \to S$

$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$

$A \to B \mid S \mid \varepsilon$

$B \to b \mid \varepsilon$

(3) Remove unit rules:

$$A \rightarrow B$$

One idea: if $A \rightarrow B$ and $B \rightarrow w$, remove $A \rightarrow B$ ← replace with $A \rightarrow w$

Will work, but 1 problem:

$A \rightarrow B$

$B \rightarrow C \mid b$

$C \rightarrow A \mid c$

$A \rightarrow B \rightarrow C \rightarrow A \mid c$

Detecting unit pairs:

How? Must have:

$$X \to Z_1,\ Z_1 \to Z_2,\ \ldots,\ Z_k \to Y$$

(since we removed ε-transitions in ②)

ComputeUnitPairs(Rules in CFG)

```
NewRules ← {(X → Y) in rules}
do:
    OldRules ← NewRules
    for (X → Y) in NewRules
        for (Y → Z) in NewRules
            NewRules ← NewRules ∪ (X → Z)
while (NewRules ≠ OldRules)
```

Now remove all unit rules $A \to B$.

For any unit pair (X, Y) + rule $Y \to w$, add $X \to w$ to the transitions

Example:

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS \mid \cancel{S}$

$S \to ASA \mid aB \mid a \mid SA \mid AS \mid \cancel{S}$

$A \to \cancel{B} \mid \cancel{S} \mid b \mid ASA \mid aB \mid a \mid SA \mid AS$

$B \to b$

Pairs: $\cancel{(S_0, S)}, \cancel{(S, S)}, \cancel{(A, B)}, (A, S)$

④ Get rid of "long" righthand sides.

Recall goal of CNF:

$X \to YZ$

$Q \to q$

$X \to aX$

$X \to XYZ$

} replace

4a: Create $V_c \rightarrow c$ for every character.

Replace c with V_c everywhere.

Now all rules are either:

$$A \rightarrow CDEF$$

or

$$V_c \rightarrow c.$$

4b: $A \rightarrow B_1 B_2 B_3 \dots B_k$ (C)

How to replace with only 2 nonterminals on the right?

$A \rightarrow B_1 C$

$C \rightarrow B_2 D$

⋮

Ex:

$S_0 \to AX \mid aB \mid a \mid SA \mid AS$

$S \to AX \mid aB \mid a \mid SA \mid AS$

$A \to b \mid AX \mid aB \mid a \mid SA \mid AS$

$B \to b$

$V_a \to a$

$X \to SA$

CNF!

Why CNF?

In general, there are an exponential number of parse trees for a given input.

So how to check quickly?

Even in CNF, might be 2^n possible parse trees:

Solution: dynamic programming!

CYK Algorithm (Cocke-Younger-Kasami '65, '67)

Given a word $w = w_1 w_2 w_3 w_4 \cdots w_k$, we'll look at all possible substrings $w_i w_{i+1} \cdots w_{j-1} w_j$, and look at how they can be parsed.

We'll build a table from the bottom up.

Ex:

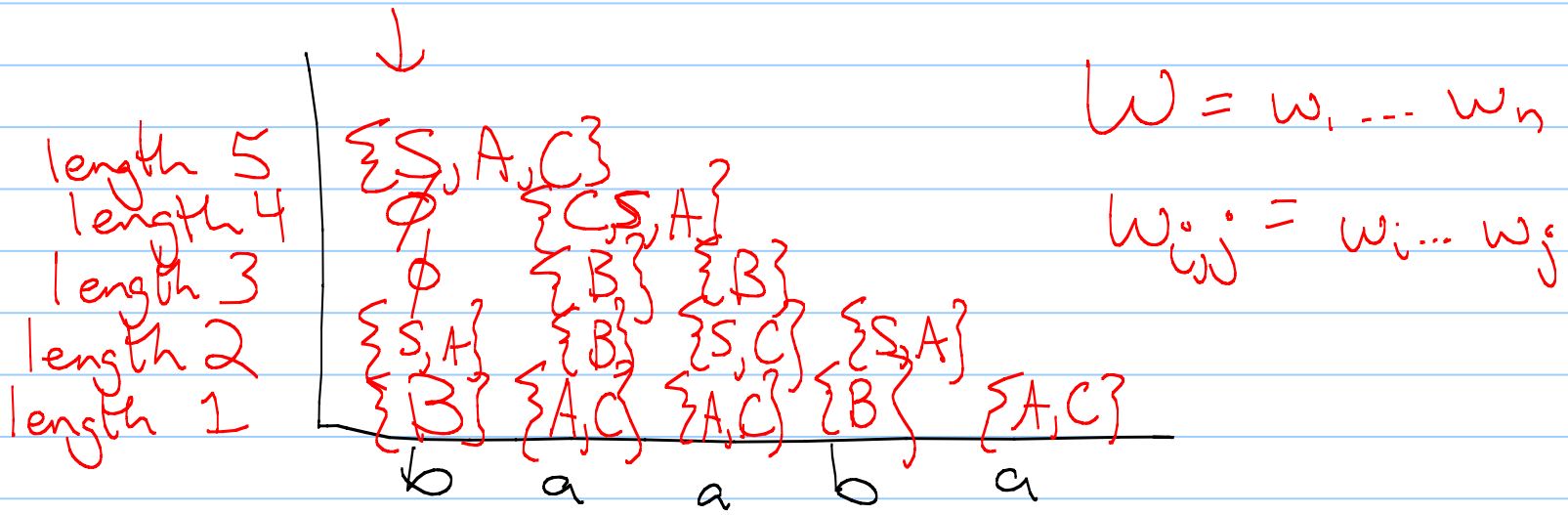
$S \to AB \mid BC$

$A \to BA \mid a$

$B \to CC \mid b$

$C \to AB \mid a$

Test if 'baaba' is in the language



$w = w_1 \dots w_n$

$w_{ij} = w_i \dots w_j$

Ex (cont)

Running times:

Say we have n rules.

Converting to CNF: $O(n^2)$

Running CYK: $O(n^3)$, $O(n^2)$ space

$O(n^2)$ entries in table

$O(n)$ lookups to fill any 1 entry

Other parsing algorithms

CYK is still pretty slow, especially for large programming languages.

After it was developed, a lot of work was put into figuring out what grammars could have faster algorithms.

Two big (& useful) classes have linear time parsers: LL & LR.

LL(1)
LR(2)

left to right

left versus right-most