Natural Language Processing

Lecture 12

Today's Outline

Review Parsing

- top-down
- bottom-up
- top-down with bottom-up filtering
- problems
- add errata to parse algorithm slide

Dynamic Programming/Earley

Review

Parsing with a CFG is the task of assigning a correct tree (or derivation) to a string given some grammar.

A correct tree is consistent with the grammar and the leaves of the tree cover all and only the words in the input.

There may be a huge number of correct trees for any given input.

Errata

```
function TOP-DOWN-PARSE(input, grammar) returns a parse tree

  agenda ← (Initial S tree, Beginning of input)
  current-search-state ← POP(agenda)
  loop
   if SUCCESSFUL-PARSE?(current-search-state) then
     return TREE(current-search-state)
   else
     if CAT(NODE-TO-EXPAND(current-search-state)) is a POS then
       if CAT(node-to-expand)
            ⊂
            POS(CURRENT-INPUT(current-search-state)) then
          PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
       else
          return reject
     else
       PUSH(APPLY-RULES(current-search-state, grammar), agenda)
   if agenda is empty then
      return reject
   else
      current-search-state ← NEXT(agenda)
  end
```

Assume that line numbering starts with the assignment of agenda as line 1.

- The "CAT" function on line 7 returns the grammatical category of a node in a parse tree. So this line is checking to see if the grammatical category of the current node to be expanded in this search state is a part-of-speech category.

- The “if” on line 8 should be similar to the one on line 7, ie. CAT(NODE-TO-EXPAND(current-search-state))
- The “subset” symbol on line 9 should be an “in” instead.
- Remove the “else return reject” on lines 12 and 13; it causes a premature termination of the search.
- The call to NEXT on line 19 should be a POP since this algorithm is explicitly depth-first.

Problems that Arise

Even after filtering...

- left recursion
- ambiguity
- inefficiencies due to backtracking

Left Recursion

Immediate

- NP → NP PP
- S → S and S

Indirect

- abstractly
  - A → B C
  - B → D E
  - D → A F
- example
  - NP → Det Nominal
  - Det → NP ' s

Solutions to Left Recursion

Rule ordering (doesn't work)

Don't use recursive rules

Don't use top-down parsing

Automatically detect and rewrite left-recursion

Limit the depth of recursion in parsing to some analytically or empirically set limit

Rule Ordering

Basic idea...

Bad

- NP $\rightarrow$ NP PP
- NP $\rightarrow$ Det Nominal

Better (but still no good)

- NP $\rightarrow$ Det Nominal
- NP $\rightarrow$ NP PP

Grammar Rewriting

It is always the case that a left-recursive grammar can be re-written into a weakly equivalent non-left-recursive one.

Can be done...

- by hand
- automatically (see the book)

Depth Bound

Set an arbitrary bound.

Set an analytically derived bound.

Run tests and drive a reasonable bound empirically.

Solutions to Ambiguity

Rely on semantics.

Rely on probabilities.

Both.

Reuse: Solution to Inefficiency

Despite all this ambiguity and backtracking there are invariants to be taken advantage of.

Recall our example from last time: *A flight from Indianapolis to Houston on TWA.*

- constituents were computed multiple times (e.g., a flight was parsed as an NP 4 times)

Instead, we want an algorithm that fills a table with solutions to sub-problems that:

- does not do repeated work
- does top-down search with bottom-up filtering (sort of)
- solves the left-recursion problem
- solves an exponential problem in less time

Dynamic Programming

Systematically fill in tables of solutions to subproblems.

When complete, the tables contain the solutions to all of the subproblems needed to solve the whole problem.

For parsing, the tables store subtrees for constituents.

Solves reparsing inefficiencies, as subtrees are not reparsed but looked up.

Solves ambiguity explosions, as the table can *implicitly* store all parses.

Dynamic Programming and Parsing

We'll use the Earley algorithm, which fills a table (*the chart*) in a single pass over the input.

The table will be size N+1 where N is the number of words in the input.

It is fruitful to think of the table entries as sitting in the gaps between the words.

The table entries will represent three distinct kinds of things:

- completed constituents
- in-progress constituents
- predicted constituents

Predicted constituents will be advanced as completed constituents are found. This results in more predictions.

States

We'll call these table entries states and represent the progress made in recognizing the state's rule with what are called <u>Dotted Rules</u>.

The three kinds of states correspond to different dot locations.

- VP $\rightarrow$ V NP $\cdot$
- NP $\rightarrow$ Det $\cdot$ Nominal
- S $\rightarrow$ $\cdot$ VP

States (continued)

Given rules like these we need to keep track of a couple of things: where the represented constituent is in the input, and what its parts are.

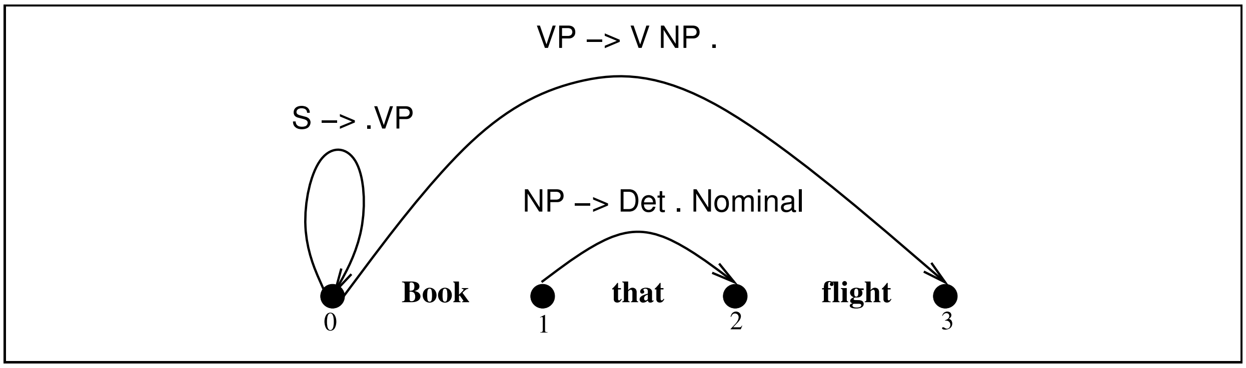
- A $\rightarrow \alpha$, [x, y]
- x indicates where the state begins
- y indicates where the dot lies

Example

Example states in parsing *Book that flight.*

- S $\rightarrow \cdot$ VP, [0,0]
  - the first 0 indicates that the constituent begins at the start of the input
  - the second 0 indicates that the dot is here as well, and thus indicates a top-down prediction
- NP $\rightarrow$ Det $\cdot$ Nominal, [1,2]
  - the NP begins at position 1
  - the dot is at position 2
  - Det has thus been successfully parsed
  - Nominal is thus predicted next
- VP $\rightarrow$ V NP $\cdot$, [0,3]
  - a successful VP parse of the entire input

Graphical Representation of States



Success

The final answer is found by looking at the last entry of the table. In particular, if we find the following kind of state there we've succeeded:

S $\rightarrow \alpha\cdot$, [0,N]

Parsing

So... parsing is sweeping through the table - *without* back-tracking - creating the three kinds of states.

New predicted states are based on existing table entries (predicted, or in-progress) that predict a certain constituent at that spot.

New in-progress states are created by updating older states to reflect the fact that previously expected completed constituents have been located.

New complete states are created when the dot in an in-progress state moves to the end.

More Specifically

1. Predict all the states you can.

2. Read an input.

See what predictions you can match. Extend matched states, add new predictions. Go to next state (goto 2).

3. At the end, see if state [N+1] contains a complete S.

Example: *Book that flight*

Chart[0]		
$\gamma \rightarrow \bullet S$	[0,0]	Dummy start state
$S \rightarrow \bullet NP\ VP$	[0,0]	Predictor
$NP \rightarrow \bullet Det\ NOMINAL$	[0,0]	Predictor
$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	Predictor
$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	Predictor
$S \rightarrow \bullet VP$	[0,0]	Predictor
$VP \rightarrow \bullet Verb$	[0,0]	Predictor
$VP \rightarrow \bullet Verb\ NP$	[0,0]	Predictor

Chart[1]		
$Verb \rightarrow book \bullet$	[0,1]	Scanner
$VP \rightarrow Verb \bullet$	[0,1]	Completer
$S \rightarrow VP \bullet$	[0,1]	Completer
$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
$NP \rightarrow \bullet Det\ NOMINAL$	[1,1]	Predictor
$NP \rightarrow \bullet Proper\text{-}Noun$	[1,1]	Predictor

Chart[2]		
$Det \rightarrow that \bullet$	[1,2]	Scanner
$NP \rightarrow Det \bullet NOMINAL$	[1,2]	Completer
$NOMINAL \rightarrow \bullet Noun$	[2,2]	Predictor
$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[2,2]	Predictor

Chart[3]		
$Noun \rightarrow flight \bullet$	[2,3]	Scanner
$NOMINAL \rightarrow Noun \bullet$	[2,3]	Completer
$NOMINAL \rightarrow Noun \bullet NOMINAL$	[2,3]	Completer
$NP \rightarrow Det\ NOMINAL \bullet$	[1,3]	Completer
$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
$S \rightarrow VP \bullet$	[0,3]	Completer
$NOMINAL \rightarrow \bullet Noun$	[3,3]	Predictor
$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[3,3]	Predictor

Errata

The order of the states in this chart doesn't reflect the order in which the algorithm would enter them. Chart[0] should look like the following:

Gamma $\rightarrow \cdot$ S

S $\rightarrow \cdot$ NP VP

S $\rightarrow \cdot$ Aux NP VP

S $\rightarrow \cdot$ VP

NP $\rightarrow \cdot$ Det NOMINAL

NP $\rightarrow \cdot$ Proper-Noun

VP $\rightarrow \cdot$ Verb

VP $\rightarrow \cdot$ Verb NP

Returning the Parse

Chart[0]

S0	$\gamma \rightarrow \bullet S$	[0,0]	[]	Dummy start state
S1	$S \rightarrow \bullet NP\ VP$	[0,0]	[]	Predictor
S2	$NP \rightarrow \bullet Det\ NOMINAL$	[0,0]	[]	Predictor
S3	$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	[]	Predictor
S4	$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	[]	Predictor
S5	$S \rightarrow \bullet VP$	[0,0]	[]	Predictor
S6	$VP \rightarrow \bullet Verb$	[0,0]	[]	Predictor
S7	$VP \rightarrow \bullet Verb\ NP$	[0,0]	[]	Predictor

Chart[1]

S8	$Verb \rightarrow book \bullet$	[0,1]	[]	Scanner
S9	$VP \rightarrow Verb \bullet$	[0,1]	[S8]	Completer
S10	$S \rightarrow VP \bullet$	[0,1]	[S9]	Completer
S11	$VP \rightarrow Verb \bullet NP$	[0,1]	[S8]	Completer
S12	$NP \rightarrow \bullet Det\ NOMINAL$	[1,1]	[]	Predictor
S13	$NP \rightarrow \bullet Proper\text{-}Noun$	[1,1]	[]	Predictor

Chart[2]

S14	$Det \rightarrow that \bullet$	[1,2]	[]	Scanner
S15	$NP \rightarrow Det \bullet NOMINAL$	[1,2]	[S14]	Completer
S16	$NOMINAL \rightarrow \bullet Noun$	[2,2]	[]	Predictor
S17	$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[2,2]	[]	Predictor

Chart[3]

S18	$Noun \rightarrow flight \bullet$	[2,3]	[]	Scanner
S19	$NOMINAL \rightarrow Noun \bullet$	[2,3]	[S18]	Completer
S20	$NOMINAL \rightarrow Noun \bullet NOMINAL$	[2,3]	[S18]	Completer
S21	$NP \rightarrow Det\ NOMINAL \bullet$	[1,3]	[S14,S19]	Completer
S22	$VP \rightarrow Verb\ NP \bullet$	[0,3]	[S8,S21]	Completer
S23	$S \rightarrow VP \bullet$	[0,3]	[S22]	Completer
S24	$NOMINAL \rightarrow \bullet Noun$	[3,3]	[]	Predictor
S25	$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[3,3]	[]	Predictor

Earley Algorithm

The Earley algorithm has three main functions that process the states in the chart and thus do all the work.

Predictor: Adds predictions into the chart.

Completer: Moves the dot to the right when new constituents are found.

Scanner: Reads the input word and enter states representing those words into the chart.

The Three Operators

Operator I/O

- input: single state
- output: new states that are added to the chart if not already present
- note that states are never removed

Predictor and Completer

- new states are added to the chart entry being processed

Scanner

- output: new states are added to the next chart entry

Predictor

Intuition: new states represent top-down expectations.

Applied when non part-of-speech non-terminals are to the right of a dot.

Same chart entry as generating state.

Completer

Intuition: parser has discovered a constituent, so must find and advance states that were waiting for this.

Applied when dot has reached right end of rule.

Same chart entry as generating state.

Scanner

New states for predicted part of speech.

Applicable when part of speech is to the right of a dot.

Next chart entry.

Earley Algorithm

```
function EARLEY-PARSE(words, grammar) returns chart

  ENQUEUE((γ → • S, [0,0]), chart[0])
  for i ← from 0 to LENGTH(words) do
   for each state in chart[i] do
     if INCOMPLETE?(state) and
              NEXT-CAT(state) is not a part of speech then
        PREDICTOR(state)
     elseif INCOMPLETE?(state) and
              NEXT-CAT(state) is a part of speech then
        SCANNER(state)
     else
        COMPLETER(state)
   end
  end
  return(chart)

 procedure PREDICTOR((A → α • B β, [i, j]))
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
        ENQUEUE((B → • γ, [j, j]), chart[j])
    end

 procedure SCANNER((A → α • B β, [i, j]))
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
       ENQUEUE((B → word[j], [j, j + 1]), chart[j+1])

 procedure COMPLETER((B → γ •, [j, k]))
    for each (A → α • B β, [i, j]) in chart[j] do
        ENQUEUE((A → α B • β, [i, k]), chart[k])
    end

 procedure ENQUEUE(state, chart-entry)
    if state is not already in chart-entry then
        PUSH(state, chart-entry)
    end
```

Useful Properties

Error handling

Alternative control strategies

Error Handling

What happens when we look at the contents of the last table column and don't find a S $\rightarrow \alpha\cdot$ rule?

Is it a total loss? Is there some way to proceed?

Yes. The chart contains every constituent and combination of constituents that could have been found for that input given the grammar.

Also useful for partial parsing or shallow parsing as in information extraction.

Alternative Control Strategies

It's trivial to take the Earley top-down strategy and change it to bottom-up or ...

... to change it to a best first strategy based on the probabilities of the constituents. Simply compute and store the probabilities of constituents in the chart as you are parsing. Then instead of expanding the columns/states in a fixed order you allow the probabilities to control the order of expansion.

For Next Time

Chapter 11.

Acknowledgements

Some of this material was taken from the course web pages of James Martin.