Today 3/4

• Parsing
  • CKY again
  • Earley

Sample Grammar

\[
\begin{align*}
S & \rightarrow \text{NP}\ VP \\
S & \rightarrow \text{Adv}\ NP\ VP \\
S & \rightarrow \text{VP} \\
\text{NP} & \rightarrow \text{Proper-Noun} \\
\text{NP} & \rightarrow \text{Det}\ \text{Nominal} \\
\text{Nominal} & \rightarrow \text{Nominal}\ \text{Noun} \\
\text{Nominal} & \rightarrow \text{Nominal}\ \text{PP} \\
\text{VP} & \rightarrow \text{Verb} \\
\text{VP} & \rightarrow \text{Verb}\ \text{NP} \\
\text{VP} & \rightarrow \text{Verb}\ \text{NP}\ PP \\
\text{VP} & \rightarrow \text{Verb}\ PP \\
\text{VP} & \rightarrow \text{VP}\ PP \\
\text{PP} & \rightarrow \text{Preposition}\ \text{NP}
\end{align*}
\]

Det — that | this | a
Nominal — book | flight | meal | money
Verb — book | include | prefer
Proper-Noun — Houston | TWA
Adv — does
Preposition — from | to | on | near | through
Dynamic Programming

- DP methods fill tables with partial results and
- Do not do too much avoidable repeated work
- Solve exponential problems in polynomial time (sort of)
- Efficiently store ambiguous structures with shared sub-parts.

CKY Parsing

- First we’ll limit our grammar to epsilon-free, binary rules (more later)
- Consider the rule A -> BC
  - If there is an A in the input then there must be a B followed by a C in the input.
  - If the A spans from i to j in the input then there must be some k st. i<k<j
    - I.e. The B splits from the C someplace.

CKY

- So let’s build a table so that an A spanning from i to j in the input is placed in cell [i,j] in the table.
- So a non-terminal spanning an entire string will sit in cell [0, n]
- If we build the table bottom up we’ll know that the parts of the A must go from i to k and from k to j
• Meaning that for a rule like $A \rightarrow B \ C$ we should look for a $B$ in $[i,k]$ and a $C$ in $[k,j]$.
• In other words, if we think there might be an $A$ spanning $i,j$ in the input... AND
• $A \rightarrow B \ C$ is a rule in the grammar THEN
• There must be a $B$ in $[i,k]$ and a $C$ in $[k,j]$ for some $i<k<j$

• So to fill the table loop over the cell $[i,j]$ values in some systematic way
  • What constraint should we put on that?
  • For each cell loop over the appropriate $k$ values to search for things to add.
CKY Algorithm

```plaintext
function CKY-PARSE(word, grammar) return: table
  for j from 1 to LENGTH(word) do
    table[j, j] = \{ [A] \mid \text{word}[j] \in \text{grammar} \}
  for i from j - 1 to 1 do
    for k from j to i + 1 do
      union \{ [A] \mid \text{word}[j] \in \text{grammar}, \text{word}[k] \in \text{grammar}, \text{word}[l] \in \text{grammar} \}

5/11/08
```

CKY Parsing

• Is that really a parser?

5/11/08

Note

• We arranged the loops to fill the table a column at a time, from left to right, bottom to top.
  • This assures us that whenever we’re filling a cell, the parts needed to fill it are already in the table (to the left and below)

5/11/08
Example

Other Ways to Do It?

• Are there any other sensible ways to fill the table that still guarantee that the cells we need are already filled?
Sample Grammar

<table>
<thead>
<tr>
<th>Non-terminals</th>
<th>Terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>S          -&gt;  NP VP</td>
<td>Det   -&gt;  that</td>
</tr>
<tr>
<td>S          -&gt;  Aux NP VP</td>
<td>Noun  -&gt;  book</td>
</tr>
<tr>
<td>S          -&gt;  VP</td>
<td>Verb   -&gt;  be</td>
</tr>
<tr>
<td>NP         -&gt;  Pronoun</td>
<td>Pronoun  -&gt; I</td>
</tr>
<tr>
<td>NP         -&gt;  Proper-Noun</td>
<td>Proper-Noun  -&gt; Houston</td>
</tr>
<tr>
<td>NP         -&gt;  Det Nominal</td>
<td>Aux  -&gt;  does</td>
</tr>
<tr>
<td>Nominal    -&gt;  Nominal</td>
<td>Preposition  -&gt; from</td>
</tr>
<tr>
<td>Nominal    -&gt;  Nominal Numeral</td>
<td>Nominal  -&gt; Numeral</td>
</tr>
<tr>
<td>Nominal    -&gt;  Nominal PP</td>
<td>PP  -&gt;  Preposition Numeral</td>
</tr>
<tr>
<td>VP         -&gt;  Verb</td>
<td>VP     -&gt;  Verb Numeral</td>
</tr>
<tr>
<td>VP         -&gt;  Numeral PP</td>
<td>Numeral  -&gt; Numeral PP</td>
</tr>
<tr>
<td>VP         -&gt;  Verb PP</td>
<td>VP     -&gt;  Verb PP</td>
</tr>
<tr>
<td>VP         -&gt;  VP PP</td>
<td>VP     -&gt;  VP PP</td>
</tr>
<tr>
<td>PP         -&gt;  Preposition Numeral</td>
<td>Numeral  -&gt; Numeral PP</td>
</tr>
</tbody>
</table>

Problem

• What if your grammar isn’t binary?
  • As in the case of the TreeBank grammar?
  • Convert it to binary… any arbitrary CFG can be rewritten into Chomsky-Normal Form automatically.
  • What does this mean?
    • The resulting grammar accepts (and rejects) the same set of strings as the original grammar.
    • But the resulting derivations (trees) are different.

Problem

• More specifically, rules have to be of the form
  A -> B C
  Or
  A -> w

  That is, rules can expand to either 2 non-terminals or to a single terminal.
Binarization Intuition

- Eliminate chains of unit productions.
- Introduce new intermediate non-terminals into the grammar that distribute rules with length > 2 over several rules. So...

S → A B C

Turns into

S → X C
X → A B

Where X is a symbol that doesn’t occur anywhere else in the grammar.

CNF Conversion

<table>
<thead>
<tr>
<th>5 → NP VP</th>
<th>5 → NP VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 → Aux VP</td>
<td>5 → Aux VP</td>
</tr>
<tr>
<td>5 → VP</td>
<td>5 → VP</td>
</tr>
<tr>
<td>5 → S</td>
<td>5 → S</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
</tr>
</tbody>
</table>

CKY Algorithm

```plaintext
function CKY-PARSE(word, grammar) return table

for j from 1 to LENGTH(word) do table[j][1...j] = { s | s ∈ terminals } in grammar }

for i from 2 to LENGTH(word) do
  for b from i - 1 to j - 1 do
    table[b][i] = table[b][i] ∪ { A | A ∈ terminals, B ∈ table[b][i], C ∈ table[i][j] }
```

```
```
CKY Notes

- Since it’s bottom up, CKY populates the table with a lot of phantom constituents.
  - Segments that by themselves are constituents but cannot really occur in the context in which they are being suggested.
  - To avoid this we can switch to a top-down control strategy or
  - We can add some kind of filtering that blocks constituents where they can not happen in a final analysis.
Break

• Quiz pushed back to Tues 3/18
• Schedule
  • Today: CKY and Earley
  • Thursday: Partial parsing, chunking and more on statistical sequence processing
  • Next week: statistical parsing

Earley Parsing

• Allows arbitrary CFGs
• Top-down control
• Fills a table in a single sweep over the input words
  • Table is length N+1; N is number of words
• Table entries represent
  • Completed constituents and their locations
  • In-progress constituents
  • Predicted constituents

States

• The table-entries are called states and are represented with dotted-rules.
  S → • VP  A VP is predicted
  NP → Det • Nominal  An NP is in progress
  VP → V NP •  A VP has been found
**States/Locations**

- **S -> ● VP [0,0]**
  - A VP is predicted at the start of the sentence
- **NP -> Det ● Nominal [1,2]**
  - An NP is in progress; the Det goes from 1 to 2
- **VP -> V NP ● [0,3]**
  - A VP has been found starting at 0 and ending at 3

---

**Graphically**

---

**Earley**

- As with most dynamic programming approaches, the answer is found by looking in the table in the right place.
- In this case, there should be an S state in the final column that spans from 0 to n and is complete.
- If that’s the case you’re done.
  - S -> α ● [0,n]
• So sweep through the table from 0 to n…
  • New predicted states are created by starting top-down from S
  • New incomplete states are created by advancing existing states as new constituents are discovered
  • New complete states are created in the same way.

• More specifically…
  1. Predict all the states you can upfront
  2. Read a word
     1. Extend states based on matches
     2. Generate new predictions
     3. Go to step 2
  3. Look at n to see if you have a winner

Earley Code

function EARLEY-PARSE(words, grammar) returns chart
  ADDTOCHART(y = • S, 0, ε, 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
        PREDICT(state)
      elseif INCOMPLETE(state) and NEXT-CAT(state) is a part of speech then
        SCANERR(more)
      else
        COMPLETER(state)
    end
  end
  return(chart)
Earley Code

```plaintext
procedure PREDICTOR(A → α β [i, j])
    for each (B → γ) in GRAMMAR.RULES.
    FOR(B, grammar) do
        AddToChart(B → γ [i, j], chart[j])
    end

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

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

Example

• Book that flight
• We should find... an S from 0 to 3 that is a completed state...

Example

<table>
<thead>
<tr>
<th>Char(index)</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
<th>S10</th>
<th>S11</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ → • S</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
<td>[0.0]</td>
</tr>
<tr>
<td>S1</td>
<td>NP</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>Adj</td>
<td>NP</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>S</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>PP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>Adj</td>
<td>NP</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S10</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S11</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Add To Chart

procedure AddToChart(state, chart-entry)
    if state is not already in chart-entry then
        PUSH-ON-END(state, chart-entry)
    end

Example

Chart[1] S12 Verb → book
        [0.1] Scanner
S13 VP → Verb
        [0.1] Completer
S14 VP → Verb • NP
        [0.1] Completer
S15 VP → Verb • VP
        [0.1] Completer
S16 VP → Verb • PP
        [0.1] Completer
S17 S → VP
        [0.1] Completer
S18 VP → VP • PP
        [0.1] Completer
S19 NP → • Pronoun
        [1.1] Predictor
S20 NP → • Proper-Noun
        [1.1] Predictor
S21 NP → • Det-Nominal
        [1.1] Predictor
S22 PP → • Prep-NP
        [1.1] Predictor

Example

Chart[2] S23 Det → the
        [1.2] Scanner
S24 NP → Det • Nominal
        [1.2] Completer
S25 Nominal → • Noun
        [2.2] Predictor
S26 Nominal → • Nominal
        [2.2] Predictor
S27 Nominal → • Nominal • PP
        [2.2] Predictor
Chart[3] S28 Noun → flight
        [2.3] Scanner
S29 Noun → • Noun
        [2.3] Completer
S30 NP → • Det Nominal
        [1.3] Completer
S31 Nominal → • Nominal
        [2.3] Completer
S32 Nominal → • Nominal • PP
        [2.3] Completer
S33 VP → Verb
        [0.3] Completer
S34 VP → Verb • NP
        [0.3] Completer
S35 PP → • Prep-NP
        [3.3] Predictor
S36 S → VP
        [0.3] Completer
S37 VP → VP • PP
        [0.3] Completer
Efficiency

• For such a simple example, there seems to be a lot of useless stuff in there.
• Why?
  • It's predicting things that aren't consistent with the input
  • That's the flipside to the CKY problem.

Details

• As with CKY that isn't a parser until we add the backpointers so that each state knows where it came from.

Back to Ambiguity

• Did we solve it?
Ambiguity

- No...
  - Both CKY and Earley will result in multiple S structures for the \([0,n]\) table entry.
  - They both efficiently store the sub-parts that are shared between multiple parses.
  - And they obviously avoid re-deriving those sub-parts.
  - But neither can tell us which one is right.

Ambiguity

- In most cases, humans don’t notice incidental ambiguity (lexical or syntactic). It is resolved on the fly and never noticed.
- We’ll try to model that with probabilities.
- But note something odd and important about the Groucho Marx example…
Next Time

- Partial Parsing and chunking
- After that we’ll move on to probabilistic parsing