Grammars and Parsing

- Context-Free Grammars and Constancy
- Some common CFG phenomena for English
- Baby parsers: Top-down and Bottom-up Parsing
- Today: Real parsers: Dynamic Programming parsing
  - CKY
  - Probabilistic parsing
  - Optional section: the Earley algorithm

Dynamic Programming

- We need a method that fills a table with partial results that:
  - Does not do (avoidable) repeated work
  - Does not fall prey to left-recursion
  - Can find all the pieces of an exponential number of trees in polynomial time.
- Two popular methods
  - CKY
  - Earley

Converting to CNF

- Rules that mix terminals and non-terminals
  - Introduce a new dummy non-terminal that covers the terminal
    - INFVP -> to VP replaced by:
    - INFVP -> TO VP
    - TO -> to
- Rules that have a single non-terminal on right ("unit productions")
  - Rewrite each unit production with the RHS of their expansions
- Rules whose right hand side length > 2
  - Introduce dummy non-terminals that spread the right-hand side

The CKY (Cocke-Kasami-Younger) Algorithm

- Requires the grammar be in Chomsky Normal Form (CNF)
- All rules must be in following form:
  - A -> B C
  - A -> w
- Any grammar can be converted automatically to Chomsky Normal Form

Automatic Conversion to CNF

<table>
<thead>
<tr>
<th>S</th>
<th>NP VP</th>
<th>S</th>
<th>NP VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Aux NP VP</td>
<td>S</td>
<td>NP PP</td>
</tr>
<tr>
<td>X</td>
<td>Aux NP</td>
<td>X</td>
<td>TWA</td>
</tr>
<tr>
<td>NP</td>
<td>Prepositional</td>
<td>NP</td>
<td>PP</td>
</tr>
<tr>
<td>Nominal</td>
<td>- Noun</td>
<td>Nominal</td>
<td>- Nominal PP</td>
</tr>
<tr>
<td>Nominal</td>
<td>- Nominal PP</td>
<td>Nominal</td>
<td>- Nominal PP</td>
</tr>
<tr>
<td>VP</td>
<td>Verb</td>
<td>VP</td>
<td>- book</td>
</tr>
<tr>
<td>VP</td>
<td>- VP FP</td>
<td>VP</td>
<td>- VP PP</td>
</tr>
<tr>
<td>FF</td>
<td>Prep NP</td>
<td>FF</td>
<td>Prep NP</td>
</tr>
</tbody>
</table>

Figure 10.15 Original L1 Grammar and its conversion to CNF
Sample Grammar

S → NP VP
S → det NP VP
S → VP
NP → Pronoun
NP → Proper-Noun
NP → Det-Nominal
Nominal → Nominal Noun
Nominal → Nominal PP
VP → Verb
VP → Verb NP
VP → Verb NP PP
VP → Verb PP
VP → VP PP
PP → Preposition NP

Back to CKY Parsing

- Given rules in CNF
- 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 goes 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

CKY Table

- Filling the [i,j]th cell in the CKY table

CKY Algorithm

```plaintext
function CKY-PARSE(word, grammar) return table
for j from 1 to LENGTH(word)
do
  table(j-1,j) = { A | d in word[j] \ Grammar }
for i from 2 to LENGTH(word)
do
  for i to j do
    table(i,j) = table(i,j) \ Grammar
    if A in table(i,k) and C in table(k,j) for some i < k < j
    return true
```

Back to CKY Parsing

- Meaning that for a rule like A → 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 → 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 just loop over the possible k values
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).
  - Are there other ways to fill the table?

CYK Example

- $S \rightarrow NP \ VP$
- $NP \rightarrow John, Mary, Denver$
- $VP \rightarrow V \ NP$
- $V \rightarrow called$
- $NP \rightarrow NP \ PP$
- $P \rightarrow from$
- $PP \rightarrow P \ NP$

Example

- $S \rightarrow NP \ VP$
- $NP \rightarrow John, Mary, Denver$
- $VP \rightarrow V \ NP$
- $V \rightarrow called$
- $NP \rightarrow NP \ PP$
- $P \rightarrow from$
- $PP \rightarrow P \ NP$

Example

- $S \rightarrow NP \ VP$
- $NP \rightarrow John, Mary, Denver$
- $VP \rightarrow V \ NP$
- $V \rightarrow called$
- $NP \rightarrow NP \ PP$
- $P \rightarrow from$
- $PP \rightarrow P \ NP$

Example

- $S \rightarrow NP \ VP$
- $NP \rightarrow John, Mary, Denver$
- $VP \rightarrow V \ NP$
- $V \rightarrow called$
- $NP \rightarrow NP \ PP$
- $P \rightarrow from$
- $PP \rightarrow P \ NP$
Example

<table>
<thead>
<tr>
<th>NP</th>
<th>PP</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>P</td>
<td>Denver</td>
</tr>
<tr>
<td>S</td>
<td>VP</td>
<td>NP</td>
</tr>
<tr>
<td>X</td>
<td>V</td>
<td>Mary</td>
</tr>
<tr>
<td>NP</td>
<td>called</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>VP</th>
<th>NP</th>
<th>PP</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>P</td>
</tr>
<tr>
<td>S</td>
<td>VP</td>
<td>NP</td>
<td>from</td>
</tr>
<tr>
<td>X</td>
<td>V</td>
<td>Mary</td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td>called</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>VP, VP</th>
<th>NP</th>
<th>PP</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>P</td>
</tr>
<tr>
<td>S</td>
<td>VP</td>
<td>NP</td>
<td>from</td>
</tr>
<tr>
<td>X</td>
<td>V</td>
<td>Mary</td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td>called</td>
<td></td>
<td></td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>S</th>
<th>VP, VP</th>
<th>NP</th>
<th>PP</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>P</td>
<td>Denver</td>
</tr>
<tr>
<td>S</td>
<td>VP</td>
<td>NP</td>
<td>from</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>V</td>
<td>Mary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td>called</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>S</th>
<th>VP, VP</th>
<th>NP</th>
<th>PP</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>P</td>
<td>Denver</td>
</tr>
<tr>
<td>S</td>
<td>VP</td>
<td>NP</td>
<td>from</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>V</td>
<td>Mary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td>called</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>John</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Back to Ambiguity

Ambiguity

Ambiguity

Converting CKY from Recognizer to Parser

How to do parse disambiguation

- Did we solve it?

- 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.
  - But neither can tell us which one is right.
  - Not a parser – a recognizer
  - The presence of an S state with the right attributes in the right place indicates a successful recognition.
  - But no parse tree... no parser
  - That’s how we solve (not) an exponential problem in polynomial time

- With the addition of a few pointers we have a parser
- Augment each new cell in chart to point to where we came from.

- Probabilistic methods
- Augment the grammar with probabilities
- Then modify the parser to keep only most probable parses
- And at the end, return the most probable parse
Probabilistic CFGs

- The probabilistic model
  - Assigning probabilities to parse trees
  - Getting the probabilities for the model
  -Parsing with probabilities
  - Slight modification to dynamic programming approach
    - Task is to find the max probability tree for an input

Probability Model

- Attach probabilities to grammar rules
- The expansions for a given non-terminal sum to 1
  - VP → Verb .55
  - VP → Verb NP .40
  - VP → Verb NP NP .05
- Read this as P(Specific rule | LHS)

PCFG

<table>
<thead>
<tr>
<th>Rule</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>.80</td>
</tr>
<tr>
<td>S → NP NP</td>
<td>.15</td>
</tr>
<tr>
<td>S → VP</td>
<td>.05</td>
</tr>
<tr>
<td>NP → Det Nom</td>
<td>.20</td>
</tr>
<tr>
<td>NP → Prop Noun</td>
<td>.30</td>
</tr>
<tr>
<td>NP → Nom</td>
<td>.05</td>
</tr>
<tr>
<td>VP → Verb</td>
<td>.40</td>
</tr>
<tr>
<td>Nom → Noun</td>
<td>.75</td>
</tr>
<tr>
<td>Nom → Prop Noun Nom</td>
<td>.05</td>
</tr>
<tr>
<td>VP → Verb</td>
<td>.55</td>
</tr>
<tr>
<td>VP → Verb NP</td>
<td>.40</td>
</tr>
<tr>
<td>VP → Verb NP NP</td>
<td>.05</td>
</tr>
</tbody>
</table>

Probability Model (1)

- A derivation (tree) consists of the set of grammar rules that are in the tree
- The probability of a tree is just the product of the probabilities of the rules in the derivation.
Probability Model (1.1)

- The probability of a word sequence P(S) is the probability of its tree in the unambiguous case.
- It's the sum of the probabilities of the trees in the ambiguous case.

Getting the Probabilities

- From an annotated database (a treebank)
- So for example, to get the probability for a particular VP rule just count all the times the rule is used and divide by the number of VPs overall.
Lots of flat rules

Example sentences from those rules

- Total: over 17,000 different grammar rules in the 1-million word Treebank corpus

Probabilistic Grammar Assumptions

- We’re assuming that there is a grammar to be used to parse with.
- We’re assuming the existence of a large robust dictionary with parts of speech
- We’re assuming the ability to parse (i.e. a parser)
- Given all that... we can parse probabilistically

Typical Approach

- Bottom-up (CKY) dynamic programming approach
- Assign probabilities to constituents as they are completed and placed in the table
- Use the max probability for each constituent going up

What’s that last bullet mean?

- Say we’re talking about a final part of a parse
  - $S \rightarrow NP, VP$
  - The probability of the $S$ is...
    - $P(S \rightarrow NP, VP) \ast P(NP) \ast P(VP)$
  - The green stuff is already known. We’re doing bottom-up parsing

Max

- I said the $P(NP)$ is known.
- What if there are multiple NPs for the span of text in question (0 to i)?
- Take the max (where?)
Problems with PCFGs

- The probability model we’re using is just based on the rules in the derivation...
  - Doesn’t use the words in any real way
  - Doesn’t take into account where in the derivation a rule is used

Solution

- Add lexical dependencies to the scheme...
  - Infiltrate the predilections of particular words into the probabilities in the derivation
  - I.e. Condition the rule probabilities on the actual words

Heads

- To do that we’re going to make use of the notion of the head of a phrase
  - The head of an NP is its noun
  - The head of a VP is its verb
  - The head of a PP is its preposition
    (It’s really more complicated than that but this will do.)
How?

- We used to have
  - VP -> V NP PP \( P(\text{rule} | \text{VP}) \)
    - That’s the count of this rule divided by the number of VPs in a treebank
- Now we have
  - VP(dumped) -> V(dumped) NP(sacks) PP(in)
    - \( P(\text{r} | \text{VP} ^\text{dumped}) \)
    - The verb \( ^\text{dumpe} \)d is the verb
    - \( ^\text{sack} \)s is the head of the NP
    - \( ^\text{in} \) is the head of the PP
  - Not likely to have significant counts in any treebank

Declare Independence

- When stuck, exploit independence and collect the statistics you can...
- We’ll focus on capturing two things
  - Verb subcategorization
    - Particular verbs have affinities for particular VPs
    - Objects affinities for their predicates (mostly their mothers and grandmothers)
    - Some objects fit better with some predicates than others

Subcategorization

- Condition particular VP rules on their head... so
  - VP -> V NP PP \( P(\text{r} | \text{VP}) \)
    - Becomes
      - \( P(\text{r} | \text{VP} ^\text{dumped}) \)
  - What’s the count?
    - How many times was this rule used with (head) dump, divided by the number of VPs that dump appears (as head) in total

Preferences

- Subcat captures the affinity between VP heads (verbs) and the VP rules they go with.
- What about the affinity between VP heads and the heads of the other daughters of the VP
- Back to our examples...

Example (right)

Example (wrong)
Preferences

- The issue here is the attachment of the PP. So the affinities we care about are the ones between dumped and into vs. sacks and into.
- So count the places where dumped is the head of a constituent that has a PP daughter with into as its head and normalize.
- Vs. the situation where sacks is a constituent with into as the head of a PP daughter.

Preferences (2)

- Consider the VPs
  - Ate spaghetti with gusto
  - Ate spaghetti with marinara
  - The affinity of gusto for eat is much larger than its affinity for spaghetti.
  - On the other hand, the affinity of marinara for spaghetti is much higher than its affinity for ate.

Preferences (2)

- Note the relationship here is more distant and doesn’t involve a headword since gusto and marinara aren’t the heads of the PPs.

Summary

- Context-Free Grammars
- Parsing
  - Top Down, Bottom Up Metaphors
  - Dynamic Programming Parsers: CKY, Earley
- Disambiguation:
  - PCFG
  - Probabilistic Augmentations to Parsers
  - Treebanks

Optional section: the Earley alg

Problem (minor)

- We said CKY requires the grammar to be binary (ie. In Chomsky-Normal Form).
- We showed that any arbitrary CFG can be converted to Chomsky-Normal Form so that’s not a huge deal.
- Except when you change the grammar the trees come out wrong.
- All things being equal we'd prefer to leave the grammar alone.
Earley Parsing

- Allows arbitrary CFGs
- Where CKY is bottom-up, Earley is top-down
- 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.
  - $S \rightarrow \cdot VP$  
    A VP is predicted
  - $NP \rightarrow \text{Det} \cdot \text{Nominal}$  
    An NP is in progress
  - $VP \rightarrow V NP \cdot$  
    A VP has been found

States/Locations

- It would be nice to know where these things are in the input so...
  - $S \rightarrow \cdot VP [0,0]$  
    A VP is predicted at the start of the sentence
  - $NP \rightarrow \text{Det} \cdot \text{Nominal} [1,2]$  
    An NP is in progress; the Det goes from 1 to 2
  - $VP \rightarrow V NP \cdot [0,3]$  
    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+1 and is complete.
- If that's the case you're done.
  - $S \rightarrow \alpha \cdot [0,n+1]$  

Earley Algorithm

- March through chart left-to-right.
- At each step, apply 1 of 3 operators
  - Predictor
    - Create new states representing top-down expectations
  - Scanner
    - Match word predictions (rule with word after dot) to words
  - Completer
    - When a state is complete, see what rules were looking for that completed constituent
**Predictor**

- **Given a state**
  - With a non-terminal to right of dot
  - That is not a part-of-speech category
  - Create a new state for each expansion of the non-terminal
  - Place these new states into same chart entry as generated state, beginning and ending where generating state ends.
  - So predictor looking at
    - `S -> . VP [0,0]`
  - results in
    - `VP -> . Verb [0,0]`
    - `VP -> . Verb NP [0,0]`

**Scanner**

- **Given a state**
  - With a non-terminal to right of dot
  - That is a part-of-speech category
  - If the next word in the input matches this part-of-speech
  - Create a new state with dot moved over the non-terminal
  - So scanner looking at
    - `VP -> . Verb NP [0,0]`
  - If the next word, "book", can be a verb, add new state:
    - `VP -> Verb . NP [0,1]`
  - Add this state to chart entry following current one
  - Note: Earley algorithm uses top-down input to disambiguate POS
  - Only POS predicted by some state can get added to chart!

**Completer**

- **Applied to a state when its dot has reached right end of role.**
- **Parser has discovered a category over some span of input.**
- **Find and advance all previous states that were looking for this category**
  - copy state, move dot, insert in current chart entry
  - **Given:**
    - `NP -> Det Nominal . [1,3]`
    - `VP -> Verb NP [0,1]`
  - **Add**
    - `VP -> Verb NP . [0,3]`

**Earley: how do we know we are done?**

- **How do we know when we are done?**
  - Find an S state in the final column that spans from 0 to n+1 and is complete.
  - If that’s the case you’re done.
    - `S -> α · [0,n+1]`

**Earley**

- **So sweep through the table from 0 to n+1.**
  - 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.

**Earley**

- **More specifically...**
  1. Predict all the states you can upfront
  2. Read a word
     1. Extend states based on matches
     2. Add new predictions
     3. Go to 2
  3. Look at N+1 to see if you have a winner
Example

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

Details

- What kind of algorithms did we just describe (both Earley and CKY)
  - Not parsers – recognizers
    - The presence of an S state with the right attributes in the right place indicates a successful recognition.
    - But no parse tree... no parser
    - That's how we solve (not) an exponential problem in polynomial time

Back to Ambiguity

- Did we solve it?
Ambiguity

- 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.
- But neither can tell us which one is right.
- Not a parser – a recognizer
  - The presence of an S state with the right attributes in the right place indicates a successful recognition.
  - But no parse tree… no parser
  - That’s how we solve (not) an exponential problem in polynomial time

Converting Earley from Recognizer to Parser

- With the addition of a few pointers we have a parser
- Augment the “Completer” to point to where we came from.

Augmenting the chart with structural information

Retrieving Parse Trees from Chart

- All the possible parses for an input are in the table
- We just need to read off all the backpointers from every complete S in the last column of the table
- Find all the S -> X. \([0,N+1]\)
- Follow the structural traces from the Completer
- Of course, this won’t be polynomial time, since there could be an exponential number of trees
- So we can at least represent ambiguity efficiently