A Recursive Parser

```python
bestScore(X, i, j, s)
if (j == i+1)
    return tagScore(X, s[i])
else
    return max(
        maxScore(X->YZ) * 
        bestScore(Y, i, k) * 
        bestScore(Z, k, j)
    )
```

• Will this parser work?
• Why or why not?
• Memory requirements?

A Memoized Parser

One small change:

```python
bestScore(X, i, j, s)
if (scores[X][i][j] == null)
    if (j == i+1)
        score = tagScore(X, s[i])
    else
        score = maxScore(X->YZ) * 
        bestScore(Y, i, k) * 
        bestScore(Z, k, j)
    scores[X][i][j] = score
return scores[X][i][j]
```

Memory: Theory

• How much memory does this require?
  • Have to store the score cache
  • Cache size: |symbols|*n^2 doubles
  • For the plain treebank grammar:
    1. X ~ 20K, n = 40, double ~ 8 bytes = ~ 256MB
    2. Big, but workable.
  • What about sparsity?

Time: Theory

• How much time will it take to parse?
  • Have to fill each cache element (at worst)
  • Each time the cache fails, we have to:
    1. Iterate over each rule X → Y Z and split point k
    2. Do constant work for the recursive calls
  • Total time: |rules|*n^3
  • Cubic time
  • Something like 5 sec for an unoptimized parse of a 20-word sentences

Unary Rules

```python
bestScore(X, i, j, s)
if (j == i+1)
    return tagScore(X, s[i])
else
    return max(
        maxScore(X->YZ) * 
        bestScore(Y, i, k) * 
        bestScore(Z, k, j)
    )
```
Same-Span Reachability

CNF + Unary Closure

- We need unaries to be non-cyclic
- Can address by pre-calculating the unary closure
- Rather than having zero or more unaries, always have exactly one

Alternate unary and binary layers
Reconstruct unary chains afterwards

Alternating Layers

Alternating Layers

A Bottom-Up Parser (CKY)

- Can also organize things bottom-up
- Optimal layout of the dynamic program depends on grammar, input, even system details.

Efficient CKY

- Lots of tricks to make CKY efficient
- Most of them are little engineering details:
  - E.g., first choose k, then enumerate through the Y[i,k] which are non-zero, then loop through rules by left child.
  - Optimal layout of the dynamic program depends on grammar, input, even system details.
- Another kind is more critical:
  - Many X[i,j] can be suppressed on the basis of the input string
  - We’ll see this next class as figures-of-merit or A* heuristics

Memory: Practice

- Memory:
  - Still requires memory to hold the score table
- Pruning:
  - score[X][i][j] can get too large (when?)
  - can instead keep beams scores[i][j] which only record scores for the top K symbols found to date for the span [i,j]

Efficient CKY

- Lots of tricks to make CKY efficient
- Most of them are little engineering details:
  - E.g., first choose k, then enumerate through the Y[i,k] which are non-zero, then loop through rules by left child.
  - Optimal layout of the dynamic program depends on grammar, input, even system details.
- Another kind is more critical:
  - Many X[i,j] can be suppressed on the basis of the input string
  - We’ll see this next class as figures-of-merit or A* heuristics

Memory: Practice

- Memory:
  - Still requires memory to hold the score table
- Pruning:
  - score[X][i][j] can get too large (when?)
  - can instead keep beams scores[i][j] which only record scores for the top K symbols found to date for the span [i,j]
Time: Theory

- How much time will it take to parse?
  - For each diff (<= n)
    - For each i (<= n)
      - For each rule X \rightarrow Y Z
      - For each split point k
        Do constant work
  - Total time: |rules|*n^3

Runtime: Practice

- Parsing with the vanilla treebank grammar:
  - 20K Rules
    (not an optimized parser)
    Observed exponent: 3.6
  - Why's it worse in practice?
    - Longer sentences "unlock" more of the grammar
    - All kinds of systems issues don't scale

Rule State Reachability

Example: NP CC •

Example: NP CC NP •

- Many states are more likely to match larger spans!

(Speech) Lattices

- There was nothing magical about words spanning exactly one position.
- When working with speech, we generally don’t know how many words there are, or where they break.
- We can represent the possibilities as a lattice and parse these just as easily.

A Simple Chart Parser

- Chart parsers are sparse dynamic programs
- Ingredients:
  - Nodes: positions between words
  - Edges: spans of words with labels, represent the set of trees over those words rooted at a
  - A chart: records which edges we’ve built
  - An agenda: a holding pen for edges (a queue)
- We’re going to figure out:
  - What edges can we build?
  - All the ways we built them.

<table>
<thead>
<tr>
<th>AGENDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>critics[0,1], write[1,2], reviews[2,3], with[3,4], computers[4,5]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHART [EMPTY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>critics</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
**Unary Projection**

- When we pop an edge off the agenda, we check the lexicon to see what tag edges we can build from it:

<table>
<thead>
<tr>
<th>critics[0,1]</th>
<th>write[1,2]</th>
<th>reviews[2,3]</th>
<th>with[3,4]</th>
<th>computers[4,5]</th>
</tr>
</thead>
</table>

**The “Fundamental Rule”**

- When we pop edges off of the agenda:
  - Check for unary projections (NNS \rightarrow critics, NP \rightarrow NNS)

  \[ Y[i,j] \text{ with } X \rightarrow Y \text{ forms } X[i,j] \]

  - Combine with edges already in our chart (this is sometimes called the fundamental rule)

  \[ Y[i,j] \text{ and } Z[j,k] \text{ with } X \rightarrow YZ \text{ form } X[i,k] \]

  - Enqueue resulting edges (if newly discovered)
  - Record backtraces (called traversals)
  - Stick the popped edge in the chart

**An Example**

**Exploiting Substructure**

- Each edge records all the ways it was built (locally)
  - Can recursively extract trees
  - A chart may represent too many parses to enumerate (how many?)

**Order Independence**

- A nice property:
  - It doesn’t matter what policy we use to order the agenda (FIFO, LIFO, random).

  - Why? Invariant: before popping an edge:
    - Any edge \( X[i,j] \) that can be directly built from chart edges and a single grammar rule is either in the chart or in the agenda.
    - Convince yourselves this invariant holds!

  - This will not be true once we get weighted parsers.

**Empty Elements**

- Sometimes we want to posit nodes in a parse tree that don’t contain any pronounced words:

  - I want John to parse this sentence
  - I want [ ? ] to parse this sentence!

  - These are easy to add to our chart parser!
    - For each position \( i \), add the “word” edge \( \epsilon[i] \)
    - Add rules like NP \rightarrow \epsilon to the grammar
    - That’s it!
UCS / A*
- With weighted edges, order matters
  - Must expand optimal parse from bottom up (subparses first)
  - CKY does this by processing smaller spans before larger ones
  - UCS pops items off the agenda in order of decreasing Viterbi score
  - A* search also well defined
- You can also speed up the search without sacrificing optimality
  - Can select which items to process first
  - Can do with any "figure of merit"
    - If your figure-of-merit is a valid A* heuristic, no loss of optimality

Non-Independence I
- Independence assumptions are often too strong.
  - All NPs: 11% 9% 6% 9% 9% 21% 22% 7%
  - NPs under S: 9% 9% 6% 9% 9% 21% 22% 7%
  - NPs under VP: 6% 9% 9% 6% 9% 21% 22% 7%
- Example: the expansion of an NP is highly dependent on the parent of the NP (i.e., subjects vs. objects).
- Also: the subject and object expansions are correlated!

Non-Independence II
- Who cares?
  - NB, HMMs, all make false assumptions!
  - For generation, consequences would be obvious.
  - For parsing, does it impact accuracy?
- Symptoms of overly strong assumptions:
  - Rewrites get used where they don’t belong.
  - Rewrites get used too often or too rarely.

Breaking Up the Symbols
- We can relax independence assumptions by encoding dependencies into the PCFG symbols:
  - Parent annotation
  - Marking possessive NPs

Lexicalization
- Lexical heads important for certain classes of ambiguities (e.g., PP attachment):
  - Lexicalizing grammar creates a much larger grammar. (cf. next week)
  - Sophisticated smoothing needed
  - Smarter parsing algorithms
  - More data needed
- How necessary is lexicalization?
  - Bilexical vs. monolexical selection
  - Closed vs. open class lexicalization

Typical Experimental Setup
- Corpus: Penn Treebank, WSJ
  - Training: sections 02-21
  - Development: section 22 (here, first 20 files)
  - Test: section 23
- Accuracy – F1: harmonic mean of per-node labeled precision and recall.
- Here: also size – number of symbols in grammar.
  - Passive / complete symbols: NP, NP^S
  - Active / incomplete symbols: NP → NP CC
**Horizontal Markovization**

![Diagram showing horizontal markovization](image1)

**Vertical Markovization**

![Diagram showing vertical markovization](image2)

**Vertical and Horizontal**

![Diagram showing vertical and horizontal markovization](image3)

**Unary Splits**

- Problem: unary rewrites used to transmute categories so a high-probability rule can be used.
- Solution: Mark unary rewrite sites with -U

**Tag Splits**

- Problem: Treebank tags are too coarse.
- Example: Sentential, PP, and other prepositions are all marked IN.
- Partial Solution:
  - Subdivide the IN tag.

**Other Tag Splits**

- UNARY-DT: mark demonstratives as DT\(^U\) (“the X” vs. “those”)
- UNARY-RB: mark phrasal adverbs as RB\(^U\) (“quickly” vs. “very”)
- TAG-PA: mark tags with non-canonical parents (“not” is an RB\(^VP\))
- SPLIT-AUX: mark auxiliary verbs with –AUX (cf. Charniak 97)
- SPLIT-CC: separate “but” and “&” from other conjunctions
- SPLIT-%: “%” gets its own tag.
A Fully Annotated (Unlex) Tree

Some Test Set Results

<table>
<thead>
<tr>
<th>Parser</th>
<th>LP</th>
<th>LR</th>
<th>F1</th>
<th>CB</th>
<th>0 CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magerman 95</td>
<td>84.9</td>
<td>84.6</td>
<td><strong>84.7</strong></td>
<td>1.26</td>
<td>56.6</td>
</tr>
<tr>
<td>Collins 96</td>
<td>86.3</td>
<td>85.8</td>
<td><strong>86.0</strong></td>
<td>1.14</td>
<td>59.9</td>
</tr>
<tr>
<td>Unlexicalized</td>
<td><strong>86.9</strong></td>
<td><strong>85.7</strong></td>
<td><strong>86.3</strong></td>
<td>1.10</td>
<td>60.3</td>
</tr>
<tr>
<td>Charniak 97</td>
<td>87.4</td>
<td>87.5</td>
<td><strong>87.4</strong></td>
<td>1.00</td>
<td>62.1</td>
</tr>
<tr>
<td>Collins 99</td>
<td>88.7</td>
<td>88.6</td>
<td><strong>88.6</strong></td>
<td>0.90</td>
<td>67.1</td>
</tr>
</tbody>
</table>

- Beats “first generation” lexicalized parsers.
- Lots of room to improve – more complex models next.

The Game of Designing a Grammar

Manual Annotation

- Annotation refines base treebank symbols to improve statistical fit of the grammar
  - Parent annotation [Johnson '98]

- Annotation refines base treebank symbols to improve statistical fit of the grammar
  - Parent annotation [Johnson '98]
  - Head lexicalization [Collins '99, Charniak '00]

- Annotation refines base treebank symbols to improve statistical fit of the grammar
  - Parent annotation [Johnson ‘98]
  - Head lexicalization [Collins ’99, Charniak ’00]

- Manually split categories
  - NP: subject vs object
  - DT: determiners vs demonstratives
  - IN: sentential vs prepositional

- Advantages:
  - Fairly compact grammar
  - Linguistic motivations

- Disadvantages:
  - Performance leveled out
  - Manually annotated

<table>
<thead>
<tr>
<th>Model</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Treebank Grammar</td>
<td>72.6</td>
</tr>
<tr>
<td>Klein &amp; Manning '03</td>
<td>86.3</td>
</tr>
</tbody>
</table>
Automatic Annotation Induction

- Advantages:
  - Automatically learned:
    - Label all nodes with latent variables.
    - Same number \( k \) of subcategories for all categories.
- Disadvantages:
  - Grammar gets too large
  - Most categories are oversplit while others are undersplit.

<table>
<thead>
<tr>
<th>Model</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Klein &amp; Manning '03</td>
<td>86.3</td>
</tr>
<tr>
<td>Matsuzaki et al. '05</td>
<td>86.7</td>
</tr>
</tbody>
</table>

Learning Latent Annotations

EM algorithm:
- Brackets are known
- Base categories are known
- Only induce subcategories

Just like Forward-Backward for HMMs.

Refinement of the DT tag

<table>
<thead>
<tr>
<th>DT</th>
<th>DT-1</th>
<th>DT-2</th>
<th>DT-3</th>
<th>DT-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>the (0.56)</td>
<td>a (0.24)</td>
<td>(0.08)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a (0.61)</td>
<td>the (0.19)</td>
<td>an (0.11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>this (0.39)</td>
<td>that (0.28)</td>
<td>(0.12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>some (0.20)</td>
<td>all (0.19)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT-1</td>
<td>DT-2</td>
<td>DT-3</td>
<td>DT-4</td>
<td></td>
</tr>
</tbody>
</table>

Hierarchical refinement

Adaptive Splitting

- Want to split complex categories more
- Idea: split everything, roll back splits which were least useful

Data likelihood with split reversed

Adaptive Splitting

- Evaluate loss in likelihood from removing each split
  \[
  \text{Data likelihood with split reversed} = \text{Data likelihood with split}
  \]
- No loss in accuracy when 50% of the splits are reversed.
Adaptive Splitting Results

Number of Phrasal Subcategories

Number of Lexical Subcategories

Final Results

Learned Splits

Learned Splits

- Proper Nouns (NNP):
  - NNP 2: John, Robert, James
  - NNP 3: York, Francisco, Street

- Personal pronouns (PRP):
  - PRP 0: it, he, I
  - PRP 1: they
  - PRP 2: them, him

- Cardinal Numbers (CD):
  - CD 1: one, two, three
  - CD 2: million, billion, trillion

Parser | F1 ≤ 40 words | F1 all words |
---|---|---|
Klein & Manning '03 | 86.3 | 85.7 |
Matsuzaki et al. '05 | 86.7 | 86.1 |
Collins '99 | 88.6 | 88.2 |
Charniak & Johnson '05 | 90.1 | 89.6 |
Petrov et. al. '06 | 90.2 | 89.7 |