General Problem

- Someone gives you a PCFG $G$
- For any given sentence, you might want to:
  - Find the best parse according to $G$
  - Find a bunch of reasonably good parses
  - Find the total probability of all parses licensed by $G$
- Techniques:
  - CKY (for best; can extend to k-best (at high space and
time cost - $k^3$ time cost of all parses - the inside
algorithm))
  - Beam search
  - Agenda/chart-based search

Parsing as Search

Grammar Symbols:

- NN
- DT
- S $\rightarrow$ NP and VP

Parse triangle:

Cards in the same stack represent different symbols
over the same span.

CKY Parsing

- In CKY parsing, we visit edges tier by tier:
  - Guarantees correctness by working inside-out.
  - Build all small bits before any larger bits that could possibly require them.
  - Exhaustive; the goal is in the last tier!

Beam Search

- State space search
- States are partial parses with an associated probability
  - Keep only the top scoring elements at each stage of the
  beam search
- Find a way to ensure that all parses of a
  sentence have the same number $N$ steps
- Leftmost top-down CFG derivations in true CNF
- Shift-reduce derivations in true CNF
  - Give a binary grammar or binarize what you've got, and
  remove unaries!

Beam Search

- Time-synchronous beam search
  - Beam at time $i$
  - Successors of beam elements
  - Beam at time $i+1$
Kinds of beam search

- Constant beam size $k$
- Constant beam width relative to best item
  - Defined either additively or multiplicatively
- Sometimes combination of the above two
- Sometimes do fancier stuff like trying to keep the beam elements diverse
- Beam search can be made very fast
- No measure of how often you find model optimal answer
  - But can track correct answer to see how often/far gold standard optimal answer remains in the beam

Beam search for HW#4?

- Would probably want to do bottom up parsing (shift-reduce parsing or a version of left-corner parsing)
  - For treebank grammars, not much grammar constraint, so want to use data-driven constraint
- Don’t actually want to store states as partial parses
  - Store them as the last rule applied, with backpointers to the previous states that built those constituents (and a probability)

Agenda-Based Parsing

- For general grammars
- Start with a table recording $s(X,i,j)$
  - Records the best score of a parse of $X$ over $i,j$
    - If the scores are negative log probabilities, then entries scan at $m$, and small is good
    - This can be a sparse or a dense map
    - Again, you may want to record backtraces as well like CKY
- Step 1: Initialize with the sentence and lexicon:
  - For each word $w$ and each tag $t$
    - Set $s(w,t) = lex.score(w,t)$

Agenda-based parsing

- Keep a list of edges called an agenda
  - Edges are triples $[X,i,j]$
  - The agenda is a priority queue
- Every time the score of some $s(X,i,j)$ improves (i.e. gets lower):
  - Add the edge $[X,i,j]$-score into the agenda
  - (Update the backtrace for $s(X,i,j)$)

Agenda-Based Parsing

- The agenda is a holding zone for edges.
- Visit edges by some ordering policy.
  - Combined edge with already-visited edges.
  - Resulting new edges go on in the agenda.

- A new way to form an edge might be a better way.

- Step II: While agenda not empty
  - Get the “next” edge $[X,i,j]$ from the agenda
  - Fetch all compatible neighbors $[Y,j,k]$ or $[Z,k,l]$ (bigger than $[X,i,j]$)
  - Compatible means that there are rules $A \rightarrow X Y$ or $B \rightarrow X Z$
  - Build all parent edges $[A,i,j,k]$ and $[B,k,l]$ found
  - If we’ve improved $s(A,i,j,k)$ then stick it on the agenda
  - Also project unary rules:
    - For all unary rules $A \rightarrow \epsilon$, score $[A,i,j]$ built from this rule on $[X,i,j]$ and put on agenda if we’ve improved $s(A,i,j)$
  - When do we know we have a parse for the root?
Agenda-based parsing

- Open questions:
  - Agenda priority: What did “next” mean?
  - Efficiency: how do we do as little work as possible?
  - Optimality: how do we know when we find the best parse of a sentence?
- If we use $b(X,i,j)$ as the priority:
  - Each edge goes on the agenda at most once
  - When an edge pops off the agenda, it’s best parse is known (why?)
  - This is basically uniform cost search (i.e., Dijkstra’s algorithm)

What can go wrong?

- We can build too many edges.
  - Most edges that can be built, shouldn’t.
  - CKY builds them all

  **Speed:** build promising edges first.

- We can build in an bad order.
  - Might find bad parses before good parses.
  - Will trigger best-first propagation.

  **Correctness:** keep edges on the agenda until you’re sure you’ve seen their best parse.

Uniform-Cost Parsing

- Let $\beta$ be the score of an edge’s Viterbi parse.

  ![Uniform-Cost Parsing Diagram](image)

- “Distance” or “cost” is the negative log probability of the rules in a tree structure.
- Uniform-cost parsing: visit edges in order of increasing $\beta$ (rather than increasing span)

Uniform-Cost Parsing

- We want to work on good parses in side-out.
  - CKY does this synchronously, by pairs.
  - Uniform-cost does it asynchronously, ordering edges by their best known parse score.

- Why it’s correct:
  - Adding a sub-tree incurs probability cost.
  - Trees have lower probabilities than their sub-parses.
  - The best-scored edge in the agenda cannot be bettering any of its sub-edges.

Speeding up agenda-based parsers

- Two options for doing less work
  - The optimal way: A* parsing
    - Klein and Manning (2003)
  - The ugly but practical way: “best-first” parsing
    - Garabato and Charniak (1998)
    - Charniak, Johnson, and Goldwaer (1998)

Modern statistical parsers

  - Done in a restricted space of lexicalized PCFGs that “factors”, allowing very efficient A* search
  - Collins (1999) exploits both the ideas of beams and agenda-based parsing
    - He places a separate beam over each span (and then, roughly, doing uniform cost search
  - Charniak (2000) uses inadmissible heuristics to guide search
    - He uses very good (but inadmissible) heuristics – dub “best-first search” to find good parses quickly
    - Perhaps unsurprisingly this is the fastest of the 3.