Advanced Natural Language Processing
Lecture 12
Parsing

Philipp Koehn
(based on slides by Mark Steedman)

15 October 2013
Parsers

A parser is an algorithm that computes a structure for an input string given a grammar. All parsers have two fundamental properties:

- **Directionality**: the sequence in which the structures are constructed (e.g., top-down or bottom-up).

- **Search strategy**: the order in which the search space of possible analyses is explored (e.g., depth-first, breadth-first).
Recursive Descent Parsing

A recursive descent parser treats a grammar as a specification of how to break down a top-level goal into subgoals. Therefore:

- Directionality = top-down: It starts from the start symbol of the grammar, and works its way down to the terminals.

- Search strategy = depth-first: It expands a given terminal as far as possible before proceeding to the next one.

Algorithm for RD Parsing: Goal is to derive the start symbol (S)

1. Choose a grammatical rule with S as its LHS (e.g., $S \rightarrow NP \ VP$), and replace S with the RHS of the rule (the subgoals; e.g., NP and VP).
Recursive Descent Parsing

2. Choose a rule with the leftmost subgoal as its LHS (e.g., NP → Det N). Replace the subgoal with the RHS of the rule.

3. Whenever you reach a lexical rule (e.g., Det → the), match its RHS against the current position in the input string.
   - If it matches, move on to the next position in the input.
   - If it doesn’t, try the next lexical rule with the same LHS.
   - If no more lexical rules have the same LHS, backtrack to the most recent set of alternatives and choose another rule with the same LHS.
   - If no more rules to choose from, back up to the previous subgoal.

4. Iterate until the whole input string is consumed, or you fail to match one of the positions in the input. Backtrack on failure.
Search Strategies

Schematic view of the search space:

In depth-first search, the parser explores one branch of the search space at a time. If this branch is a dead-end, it needs to backtrack.

In breadth-first search, the parser explores all possible branches in parallel (often impossible due to memory requirements).
Shift-Reduce Parsing

Search strategy does not imply a particular directionality in which structures are built.

Recursive descent parsing searches depth-first and builds top-down.

Shift-reduce parsing, in contrast, searches depth-first but builds structures bottom-up.

It does this by shifting terminal symbols from the input string onto a stack, and reducing them to the LHS side of a rule when the top elements in the stack match its RHS. The LHS of the rule then gets pushed on the stack for later RHS-rule matching and subsequent reduction.

All depth-first parsing is inherently serial, and serial parsers can be massively inefficient when faced with local ambiguity.
Global and Local Ambiguity

A string can have more than one structural analysis (called *global ambiguity*) for one or both of two reasons:

- Grammatical rules allow for different attachment options;
- Lexical rules that allow a word to be in more than one word class.

Within a single analysis, some sub-strings can be analyzed in more than one way (called *local ambiguity*), even if not all these sub-string analyses are compatible with some global analysis of the entire string.

Local ambiguity is very common in Natural Languages.
Complexity

Depth-first parsing strategies demonstrate other problems with “parsing as search”:

1. **Structural ambiguity** in the grammar and **lexical ambiguity** in the words can lead the parser down a wrong path.

2. So the same sub-tree may be built several times: whenever a path fails, the parser undoes its work, backtracks and starts again.

The complexity of this **blind backtracking** is exponential in the worst case because of repeated **re-analysis** of the same sub-string.

Here we will look at handling ambiguity via **Chart parsing**.
Dynamic Programming

With a CFG, a parser should be able to avoid re-analyzing sub-strings because the analysis of any sub-string is independent of the rest of the parse.

The dog saw a man in the park

The parser’s exploration of its search space can exploit this independence if the parser uses dynamic programming.

Dynamic programming is the basis for all chart parsing algorithms.
Parsing as Dynamic Programming

- Given a problem, Dynamic Programming systematically fills a table of solutions to sub-problems (memoization).

- Once solutions to all sub-problems have been accumulated, DP solves the overall problem by composing them.

For parsing, sub-problems are analyses of sub-strings, which are memoized in a chart (aka well-formed substring table, WFST). Each analysis corresponds to:

- a complete constituent (sub-tree), indexed by the start and end of the sub-string that it covers;

- or a hypothesis about what complete constituent might be found, indexed by the start and end of the sub-strings that support it.
Depicting a WFST/Chart

A well-formed substring table (aka chart) can be depicted as either a matrix or a graph. Both contain the same information.

When a WFST (aka chart) is depicted as a matrix:

- Rows and columns of the matrix correspond to the start and end positions of a span (ie, starting right before the first word, ending right after the final one);
- A cell in the matrix corresponds to the sub-string that starts at the row index and ends at the column index. It can contain information about the type of constituent (or constituents) that span(s) the substring, pointers to its sub-constituents, and/or predictions about what constituents might follow the substring.
**Depicting a WFST as a Matrix**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Prep</td>
<td>PP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Det</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0  *See* 1 *with* 2 *a* 3 *telescope* 4 *in* 5 *hand* 6
Depicting a WFST as a Graph

• Here, nodes/vertices represent positions in the text string, starting before the first word, ending after the final word.

• arcs/edges connect vertices at the start and the end of a span to represent a particular substring. Edges can be labelled with the same information as in a cell in the matrix representation.

\[ \begin{align*}
\text{with a telescope} & \\
1 & 2 & 4 & 3 \\
\text{N} & \text{Prep} & \text{Det} & \text{NP} \\
\text{PP} & \\
1 & \text{with} & 2 & \text{a} & 3 & \text{telescope} & 4
\end{align*} \]
Algorithms for Chart Parsing

Important members of the chart parsing family include:

○ the CKY algorithm, which memoizes only constituents;

○ three algorithms that memoize both constituents and predictions:
  
  • a bottom-up chart parser

  • a top-down chart parser

  • the Earley algorithm
CKY Algorithm

CKY (Cocke, Kasami, Younger) is an algorithm for recognizing constituents and recording them in the chart (WFST).

CKY was originally defined for Chomsky Normal Form (A→BC; A→a), and later generalized by ??.

We can enter constituent A in cell \((i, j)\) if there is a rule \(A \rightarrow B\) and \(B\) is found in cell \((i, j)\), or if \(A \rightarrow B C\) and \(B\) is found in cell \((i, k)\) and \(C\) is found in cell \((k, j)\).

Proceeding systematically, CKY guarantees that the parser only looks for rules that use a constituent from \(i\) to \(j\) after it has processed all the constituents that end at \(i\). This avoids anything being missed.
Chart Parsing with the CKY Algorithm

Let $\text{Close}(X) = \{ B \mid B \rightarrow^* A, \text{using unary productions, and } A \in X \}$

**B**\text{UILD}_C\text{KY}_C\text{HART}(t, [w_1, \ldots, w_n])

\textbf{for} j \leftarrow 1 \textbf{ to } n

\textbf{do}

$\text{t}(j-1, j) \leftarrow \text{Close}(\{w_j\})$

\textbf{for} k \leftarrow 1 \textbf{ to } n

\textbf{for} j \leftarrow k \textbf{ to } n

\textbf{for} m \leftarrow 1 \textbf{ to } k-1

\textbf{do}

$\text{t}(j-k, j) \leftarrow \text{t}(j-k, j) \cup \text{Close}(\{A \mid A \rightarrow B C \text{ for some } B \in \text{t}(j-k, j-m) \text{ and } C \in \text{t}(j-m, j)\})$

This algorithm is complete and performs recognition in time $O(n^3)$. 
Visualizing the Chart

Grammatical rules

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$\rightarrow$ NP VP</td>
</tr>
<tr>
<td>NP</td>
<td>$\rightarrow$ Det Nom</td>
</tr>
<tr>
<td>NP</td>
<td>$\rightarrow$ Nom</td>
</tr>
<tr>
<td>Nom</td>
<td>$\rightarrow$ N SRel</td>
</tr>
<tr>
<td>Nom</td>
<td>$\rightarrow$ N</td>
</tr>
<tr>
<td>VP</td>
<td>$\rightarrow$ TV NP</td>
</tr>
<tr>
<td>VP</td>
<td>$\rightarrow$ IV PP</td>
</tr>
<tr>
<td>VP</td>
<td>$\rightarrow$ IV</td>
</tr>
<tr>
<td>PP</td>
<td>$\rightarrow$ Prep NP</td>
</tr>
<tr>
<td>SRel</td>
<td>$\rightarrow$ Relpro VP</td>
</tr>
</tbody>
</table>

Lexical rules

<table>
<thead>
<tr>
<th>Lexicon</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Det</td>
<td>$\rightarrow$ a</td>
</tr>
<tr>
<td>N</td>
<td>$\rightarrow$ fish</td>
</tr>
<tr>
<td>Prep</td>
<td>$\rightarrow$ in</td>
</tr>
<tr>
<td>TV</td>
<td>$\rightarrow$ saw</td>
</tr>
<tr>
<td>IV</td>
<td>$\rightarrow$ fish</td>
</tr>
<tr>
<td>Relpro</td>
<td>$\rightarrow$ that (relative pronoun)</td>
</tr>
</tbody>
</table>

Nom: nominal (the part of the NP after the determiner, if any).
SRel: subject relative clause, as in the frogs that ate fish.
Visualizing the Chart

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the frogs ate fish
Visualizing the Chart (0,1)

Unary branching rules: det → the
Visualizing the Chart (1,2)

Unary branching rules: N → frogs, Nom → N, NP → Nom
Visualizing the Chart (2,3)

Unary branching rules: tv → ate
Visualizing the Chart (3,4)

 Unary branching rules: $N \rightarrow$ frogs, $\text{Nom} \rightarrow N$, $\text{NP} \rightarrow \text{Nom}$, $\text{iv} \rightarrow$ fish, $\text{vp} \rightarrow$ iv
Visualizing the Chart (0,2)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n</td>
<td>nom</td>
<td>np</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>n</td>
<td>nom</td>
</tr>
<tr>
<td></td>
<td>frogs</td>
<td>ate</td>
<td>fish</td>
<td></td>
</tr>
</tbody>
</table>

Binary branching rule: \( NP \rightarrow \text{Det Nom} \ (0,1) \land (1,2) \Rightarrow (0,2) \)
Visualizing the Chart (1,3)

(1,2) & (2,3) ↯
Visualizing the Chart (2,4)

Binary branching rule: $\text{VP} \rightarrow \text{TV NP}$

$(2,3) \& (3,4) \sim (2,4)$
Visualizing the Chart (1,4)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>nom</td>
<td>np</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td>vp</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>nom</td>
<td>np</td>
</tr>
</tbody>
</table>

the frogs ate fish

Binary branching rule: $S \rightarrow \text{NP VP}$  
$(1,2) \& (2,4) \rightsquigarrow (1,4)$  
$(1,3) \& (3,4) \not\rightsquigarrow$
Visualizing the Chart (0,4)

Binary branching rule: $S \rightarrow NP \ VP$  
$(0,1) \& (1,4) \not\rightarrow \ (0,2) \& (2,4) \leadsto (0,4) \ (0,3) \& (3,4) \not\rightarrow$
From CKY Recognizer to CKY Parser

We cannot tell from the CKY chart as specified, the syntactic analysis of the input string.

We just have a chart recognizer, a way of determining whether a string belongs to the language generated by the grammar.

Changing this to a parser requires recording which existing constituents were combined to make each new constituent. This requires another field to record the one or more ways in which a constituent spanning \((i,j)\) can be made from constituents spanning \((i,k)\) and \((k,j)\).
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>det</td>
<td>np (( ))</td>
<td>S (( ))</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>n nom (( ))</td>
<td>np (( ))</td>
<td>S (( ))</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>tv</td>
<td>vp (( ))</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>np (( ))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the</td>
<td>frogs</td>
<td>ate</td>
<td>soup</td>
</tr>
</tbody>
</table>
Adding Prediction to the Chart: Chart entries

Like all Dynamic Programming algorithms, CKY avoids redundant work by memorizing all the constituents it finds.

What it doesn’t record is any justification for a chart entry – why it was built.

So if we have two VP rules:

\[
\text{VP} \rightarrow V \text{ NP} \\
\text{VP} \rightarrow \text{VP PP}
\]

and the input string

\[
\text{The boy opened the box on the floor}
\]
Chart entries

We don’t know which production the VP arc [2, 8] represents (here using a graph representation of the chart). Is it VP → V NP?
Chart entries

We don’t know which production the VP arc [2, 8] represents (here using a graph representation of the chart). Or VP → VP PP?

The boy opened the box on the floor.
Chart entries

If the entire production were recorded, rather than just its LHS (ie, the constituent that it analyses), then we’d know.
References
