Accelerated Natural Language Processing 2018

Lecture 13: (Features), Parsing as search and as dynamic programming

Henry S. Thompson
Drawing on slides by Mark Steedman via Philipp Koehn
15 October 2018

1. Features [Material in slides 1--8 are not examinable]

The same feature has been around in Linguistics for a long time.
The basic idea is to capture generalizations by decomposing monolithic categories into collections of simpler features.
Originally developed for phonology, where we might have e.g.

\[ \begin{align*}
  :v & = \text{high, rising} \\
  :c & = \text{high, front} \\
  :a & = \text{low, front} \\
  :u & = \text{high, front}
\end{align*} \]

Where we can now 'replace' any \( :v \) and \( :c \) behave similarly in certain cases, while \( :a \) and \( :u \) do not.

These are all binary features:

- sometimes also used at the level of syntax:
  - + (singular, + /plural)

2. Features, cont’d

But more often we find features whose values are some enumerated type:

\[
\begin{array}{cccccc}
\text{person} & \text{number} & \text{agreement} & \text{count} & \text{mass} \\
 1 & 2 & 3 & \text{sg} & \text{pl} \\
\end{array}
\]

We'll follow J&M and write collections of features like this:

\[ \{ \text{person: 3rd, number: pl, agreement: y} \} \]

3. Features in use

We can now add feature bundles to categories in our grammars.
In practice we allow some further notational conveniences:

- Not all features need be specified (e.g. the number feature for ‘sheep’).
- It is convenient to allow the values of features to be variables.
- In rules, we can add constraints in terms of those variables to rules.

For example:

\[ \{ \text{number: x, } \} \]

4. Features: more than notational convenience?

At one level, features are just a convenience.
But they also “capture generalisations”

Consider the noun phrase “a sheep”, and the following rules:

\[
\begin{align*}
  \text{agreement} & : v = + \\
  \text{person} & : 3 \\
  \text{number} & : \text{sg} \\
  \text{category} & : \text{N}
\end{align*}
\]

The name “sheep” and the following rules:

\[
\begin{align*}
  \text{agreement} & : v = + \\
  \text{person} & : 3 \\
  \text{number} & : \text{pl} \\
  \text{category} & : \text{N}
\end{align*}
\]

\[
\begin{align*}
  \text{agreement} & : v = + \\
  \text{person} & : 3 \\
  \text{number} & : \text{sg} \\
  \text{category} & : \text{N}
\end{align*}
\]

The three occurrences of “sheep” don’t just appear the same:

- They all agree.
- They all have a single structure.
- We can change one if the future will be a change to all.
- ‘a sheep’ and ‘the sheep’ are different in terms of 
  – Their voice.
  – Their number.

Puts essentially all the expressive power of the grammar into feature structures.

5. Infinity again: categories

Once you move to feature bundles as the values of features, the implications of unification run deep.

The resulting parse tree reveals that we have not only agreement, but also different grammatical classes:

\[
\begin{array}{cccc}
\text{agreement} & \text{person} & \text{number} & \text{category} \\
 1 & 2 & 3 & \text{N} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{agreement} & \text{person} & \text{number} & \text{category} \\
 1 & 2 & 3 & \text{N} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{agreement} & \text{person} & \text{number} & \text{category} \\
 1 & 2 & 3 & \text{N} \\
\end{array}
\]

whereby the \( \{ \} \) mean that all three agreement values are the same feature structure.

6. Unification

When we write \( \text{\{agreement: z\}} \), we are making a stronger claim, even though technically correct.

This single line corresponds to a collection of single-scope category rules.

\[
\begin{align*}
  \text{agreement: } & z \\
  \text{person: } & 3 \\
  \text{number: } & \text{sg} \\
  \text{category: } & \text{N}
\end{align*}
\]

This is what this is called, in section 15.2

\[ \{ \text{agreement: x, } \} \]

\[ \{ \text{agreement: y, } \} \]

\[ \{ \text{agreement: z, } \} \]

where by the \( \{ \} \) we mean that all three agreement values are the same feature structure.

7. Unification, cont’d

The implications of unification run deep.

The three occurrences of “sheep” don’t just appear the same:

- They all agree.
- They all have a single structure.
- We can change one if the future will be a change to all.
- ‘a sheep’ and ‘the sheep’ are different in terms of:
  - Their number.
  - Their voice.

The directed acyclic graph (DAG) way of drawing feature structures used in J&M 15.4 makes

\[
\begin{array}{cccc}
\text{agreement} & \text{person} & \text{number} & \text{category} \\
 1 & 2 & 3 & \text{N} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{agreement} & \text{person} & \text{number} & \text{category} \\
 1 & 2 & 3 & \text{N} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{agreement} & \text{person} & \text{number} & \text{category} \\
 1 & 2 & 3 & \text{N} \\
\end{array}
\]

8. Parsers

A parser is an algorithm that computes a structure for an input string given a grammar.

All parsers have two fundamental properties:

\[
\begin{align*}
  \text{agreement} & : v = + \\
  \text{person} & : 3 \\
  \text{number} & : \text{sg} \\
  \text{category} & : \text{N}
\end{align*}
\]
Almost always with no children
We'll use a grammar as a specification of how to break down a top-level goal into subgoals.
This means that it works very similarly to a particular blind approach to constructing a rewriting computation description.

**Directionality**
- Top-down: the order in which the search space of possible analyses is explored
  - Usually, depth-first, breadth-first or best-first

**Search strategy**
- The order in which the structures are constructed
  - Almost always top-down or bottom-up

**11. Recursive Descent Parsing: Algorithm sketch**

We start with:
- a tree consisting of an "S" node with no children
- This node is currently the subgoal
- It is empty
- It is a right sequence

![Tree diagram](image)

1. Repeatedly,
   a. If the subgoal is a non-terminal
      i. Choose a rule from the set of rules in the grammar whose left-hand sides
         match the subgoal and the current as-yet-unconsumed items of input
         - For example, the very first time around the loop, we might choose:
           1. S → NP VP
      b. Add the first item of the input to the top of the stack
      c. Advance the subgoal
      d. Make the first of these the new subgoal
   b. Otherwise (the subgoal is a terminal)
      i. Consume the first item of the input
      j. Advance the subgoal
      k. Go back to 1
   c. Otherwise (they don't match),
      i. (unconsumed) input sequence
      j. Failing which, if the current subgoal is not the root, set the subgoal to the current
         stack
      k. Go back to 1
   d. Otherwise (the subgoal doesn't have a sibling)
      i. Add children to the subgoal node corresponding to the symbols in the right-hand side of the chosen rule, in order
      j. Go back to 1

**12. Recursive Descent Parsing: Algorithm sketch, concluded**

The three imperative actions in the preceding algorithm are defined as follows:

**Choose**
- Pick one member from the set of rules
  1. If the set is empty, you're done
  2. Otherwise, *pick* one member from the set of rules
    - With the unchosen rules, the current tree and subgoal and the current (unconsumed) input sequence

**Advance**
- Change the subgoal, as follows:
  1. If the current (unconsumed) input has a sibling to its right, pick that
  2. If the current subgoal is not the root, set the subgoal to the current backtrack pointer, and back 12
  3. If the top node of the stack is empty, we win!
  4. Otherwise, *make* the first of these the new subgoal
    - With the unchosen rules of the current backtrack pointer and the current backtrack pointer of the current backtrack pointer

**Backtrack**
- Try to change your mind. That is:
  1. Unless the stack is empty, *pop* the top backtrack pointer off the backtrack stack and push it on top of the backtrack stack
  2. Otherwise, *pop* the top backtrack pointer off the backtrack stack
  3. Choose a rule from the set of rules
  4. Go back to the top of the algorithm

**13. Search Strategies**

Schematic view of the top-down search space:

- In depth-first search the parser
  - explores one branch of the search space at a time
  - For example, using a stack, it first and last complete trees are to try to expand
15. 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 different attachment options
- Syntactic rules that allow a word to be in more than one word class.

Within a single analysis, same sub-strings can be analyzed in more than one way:

- Even if all the rules of the grammar are non-ambiguous
- That is, if they are not compatible with any complete analysis of the entire string
- This is called local ambiguity

Local ambiguity is very common in natural languages as described by formal grammars.

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

16. Complexity

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

- Structural ambiguity can be a problem even in isolated sub-strings:
- If the parser doesn’t find a complete structure, it can't backtrack to find other possibilities.

The complexity of this "blind" backtracking is exponential in the worst case because of repetitive analysis of the same substring.

- Older approaches that first-read in this week’s lab

Chart-parsing is the name given to a family of solutions to this problem.

17. Dynamic Programming

It seems like we should be able to avoid the kind of repeated reparsing a simple recursive descent parser must do.

A CFG parser, that is, a context-free parser, should be able to avoid re-analyzing sub-strings

- because the analysis of any sub-string is independent of the end of the parse

The parser’s exploration of this search space can exploit this independence

- if the parser uses dynamic programming.

Dynamic programming is the basis for all chart parsing algorithms.

18. Parsing as dynamic programming

Given a problem, dynamic programming systematically fills in a table of solutions to sub-problems:

- a process sometimes called memoisation

Once solutions to all sub-problems have been accumulated

- Of course the overall problem is composed of them

For parsing, sub-problems are analyses of sub-strings:

- which can be memoised
- in a chart
- Also known as a well-formed substring table, WFST

Each entry in the chart or WFST corresponds to a complete constituent (sub-tree), indexed by the start and end of the sub-string that it covers.

- A chart-parser gains power from the fact that it is not required to backtrack to a previous state when a failure is encountered.

19. Depicting a WFST/Chart

A well-formed substring table (chart) can be depicted as either a chart or a matrix:

- 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 of tokens.

- Each entry in the chart is a cell that describes the relationship between the tokens it contains.

- A cell can contain:
- a single branch of a possible path
- a string of tokens corresponding to a sequence of constituent (or constituents) that span the substring
- a CFG parser, should be able to avoid re-analyzing sub-strings
- a cell that corresponds to a set of predictions about what constituents might be the constituent.
20. Depicting a WFST as a matrix

Here is a sample matrix, part-way through a parse:

<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>Prop</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>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can read this as saying:
• There is a PP from 1 to 4
• Because there is a Prep from 1 to 2
• and an NP from 2 to 4.

21. Depicting a WFST as a graph

A sample graph, for the same situation mid-parse:

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

• Arcs (or 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.

22. Algorithms for chart parsing

Important examples of parser types which use a WFST include:
• The CKY algorithm, which memoises only complete constituents.
• Three algorithm families that involve memoisation of both complete and incomplete constituents.
  • Incomplete constituents can be understood as predictions:
    ■ Top-down chart parsers
    ■ May include top-down filtering
    ■ The Earley algorithm

23. CKY Algorithm

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

CKY was originally defined for Chomsky Normal Form:

A → B C
A → a

• (Much more recently, this restriction has been lifted in a version by Lang and Leiss)

• The example below follows them in part, also allowing unary rules of the form A → B C and there is at least one k between i and j such that
  • B is found in cell (i,k)
  • C is found in cell (k,j)

24. CKY parsing, cont’d

Proceeding systematically bottom-up, CKY guarantees that the parser only looks for rules which might yield a constituent from i to j after it has found all the constituents that might contribute to it, that is:
• That are shorter than it is
• That end at or to the left of j
• This guarantees that every possible constituent will be found.

25. Visualising the chart: YACFG

Grammatical rules Lexical rules
S → NP VP
Det → a | the (determiner)
NP → Det Nom
N → fish | frogs | soup (noun)
NP → Nom
Prep → in | for (preposition)
Nom → N SRel
TV → saw | ate (transitive verb)
Nom → N IV → fish | swim (intransitive verb)
VP → TV NP
Relpro → that (relative pronoun)
VP → IV PP
VP → IV
PP → Prep NP
SRel → Relpro VP

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

Non-terminals occurring only in the lists of lexical rules are sometimes called pre-terminals:
• In the above grammar, that’s Det, N, Prep, TV, IV, Relpro.

Sometimes instead of sequences of words:
• we just parse sequences of pre-terminals
• At least during grammar development.