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 name **feature** has been around in Linguistics for a long time

The basic idea is to capture generalisations by decomposing monolithic categories into collections of simpler features

Originally developed for phonology, where we might have e.g.

/i/	+high, +front
/e/	-high, +front
/o/	-high, -front
/u/	+high, -front

Where we can now 'explain' why /i/ and /u/ behave similarly in certain cases, while /i/ and /e/ go together in other cases.

Those are all **binary** features

- sometimes also used at the level of syntax:
- +/- singular; +/- finite

2. Features, cont'd

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

person: {1st,2nd,3rd}; number: {sg, pl}; ntype: {count, mass}

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

$$\begin{bmatrix} \text{person} & \text{3rd} \\ \text{number} & \text{pl} \end{bmatrix}$$

It will be convenient to generalise and allow features to take feature bundles as values:

$$\begin{bmatrix} \text{ntype} & \text{count} \\ \text{agreement} & \begin{bmatrix} \text{person} & \text{3rd} \\ \text{number} & \text{pl} \end{bmatrix} \end{bmatrix}$$

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')
- In rules, we allow the values of features to be variables
- And we can add constraints in terms of those variables to rules

For example

$$\text{N}\begin{bmatrix} \text{ntype} & \text{count} \\ \text{agreement} & \begin{bmatrix} \text{person} & \text{3rd} \\ \text{number} & \text{pl} \end{bmatrix} \end{bmatrix} \rightarrow \text{men | dogs | cats|...}$$

$$\text{NP}\begin{bmatrix} \text{agreement} & x \end{bmatrix} \rightarrow \text{D}\begin{bmatrix} \text{agreement} & y \end{bmatrix} \text{N}\begin{bmatrix} \text{agreement} & z \end{bmatrix} \quad x = y = z$$

4. Features: more than notational convenience?

At one level, features are just a convenience

The allow us to write lexicon entries and rules more transparently

But they also "capture generalisations"

If we write a pair of rules using some (in principle opaque) complex category labels, they are not obviously related in any way:

$S \rightarrow \text{NP}_{\text{sg}}\text{VP}_{\text{sg}}$ $S \rightarrow \text{NP}_{\text{pl}}\text{VP}_{\text{pl}}$

It appears as if we have to justify each of these independently

- or that we *might* have had one without the other
- or had $S \rightarrow \text{NP}_{\text{sg}}\text{VP}_{\text{pl}}$ just as well

Whereas when we write

$$S \rightarrow \text{NP VP [<NP agreement> = <VP agreement>]}$$

we are making a stronger claim, even though 'behind the scenes' this single line corresponds to a collection of simple atomic-category rules

5. Infinity again: categories

Once you move to feature bundles as the values of features

- You can in principle have an infinite number of categories
  - By having one or more recursive features
- And, as with infinite numbers of rules, that actually changes your position on the Chomsky hierarchy

One strand of modern grammatical theory

- From GPSG to HPSG to so-called sign-based grammatical theories

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

6. Unification

When we write '=' between two feature paths or feature variables, we mean more than an equality test

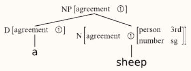
Consider the noun phrase "a sheep", and the following rules

$$\text{N}\begin{bmatrix} \text{ntype} & \text{count} \\ \text{agreement} & \begin{bmatrix} \text{person} & \text{3rd} \end{bmatrix} \end{bmatrix} \rightarrow \text{sheep}$$

$$\text{D}\begin{bmatrix} \text{agreement} & \begin{bmatrix} \text{person} & \text{3rd} \\ \text{number} & \text{sg} \end{bmatrix} \end{bmatrix} \rightarrow \text{a}$$

$$\text{NP}\begin{bmatrix} \text{agreement} & x \end{bmatrix} \rightarrow \text{D}\begin{bmatrix} \text{agreement} & y \end{bmatrix} \text{N}\begin{bmatrix} \text{agreement} & z \end{bmatrix} \quad x = y = z$$

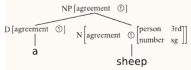
The resulting parse tree reveals that we have not only *tested* for compatibility between the various feature structures, we've actually *merged* them:



where by the ① we mean that all three agreement values are the *the same* feature structure

7. Unification, cont'd

The implications of unification run deep

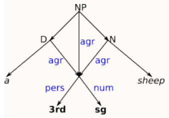


The three occurrences of ① don't just appear the same

- They *are* the same
- That is, a single structure, shared 3 times
- So any change to one in the future will be a change to all
- As would be the case with e.g. "the sheep runs" or "the sheep run"

J&M give a detailed introduction to **unification**, which is what this is called, in section 15.2 (J&M 2nd ed.), and a formal definition in section 15.4.

The directed acyclic graph (DAG) way of drawing feature structures used in J&M 15.4 makes clearer when necessary structure identity is the case, as opposed to contingent value equality



8. 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

- Almost always **top-down** or **bottom-up**

Search strategy
The order in which the search space of possible analyses is explored

- Usually **depth-first**, **breadth-first** or **best-first**

9. Recursive Descent Parsing

A **recursive descent** parser treats 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 interpretation derivation:

Directionality
Top-down:

- starts from the start symbol of the grammar
- works *down* to the terminals

Search strategy
Depth-first:

- expands the left-most unsatisfied non-terminal
- until it gets to a terminal
  - which either matches the next item in the input
  - or it doesn't

10. Recursive Descent Parsing: preliminaries

We're trying to build a parse tree, given

- a grammar
- an input, i.e. a sequence of terminal symbols

As for any other depth-first search, we may have to **backtrack**

- So we must keep track of **backtrack points**
- And whenever we make a *choice* among several rules to try, we add a backtrack point consisting of
  - a partial tree
  - the remaining as-yet-unexplored rules
  - and the as-yet-unconsumed items of input

Note that, to make the search go *depth*-first

- We'll use a **stack** to keep track
- That is, we'll operate **last in, first out** (**LIFO**)

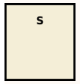
Finally, we'll need a notion of where the focus of attention is in the tree we're building

- We'll call this the **subgoal**

11. Recursive Descent Parsing: Algorithm sketch

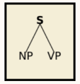
We start with

- a tree consisting of an 'S' node
  - with no children
  - This node is currently the subgoal
- An empty stack
- An input sequence

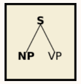


Repeatedly

1. If the subgoal is a non-terminal
   a. ***Choose*** a rule from the set of rules in the grammar whose left-hand sides match the subgoal
      For example, the very first time around the loop, we might choose
      `S → NP VP`
   b. add children to the subgoal node corresponding to the symbols in the right-hand side of the chosen rule, in order
      In our example, that's two children, `NP` and `VP`



   d. Make the first of these the new subgoal
      a.



      b. This is the other thing which makes this a *depth-first* search
   e. Go back to (1)
2. Otherwise (the subgoal is a terminal)
   a. If the input is empty, ***Backtrack***
   b. If the subgoal matches the first item of input
      i. ***Consume*** the first item of the input
      ii. ***Advance*** the subgoal
      iii. Go back to (1)
   c. Otherwise (they don't match), ***Backtrack***

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 has only one member, you're done
2. Otherwise, **push** a new backtrack point onto the stack
   - 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 subgoal has a sibling to its right, pick that
2. Failing which, if the current subgoal is not the root, set the subgoal to the current subgoal's parent, and go back to (1)
3. Failing which, if the input is empty, we win
   - The current subgoal is the 'S' at the root, and it is the top node of a complete parse tree for the original input
4. Otherwise, **Backtrack**

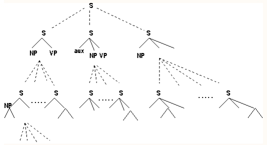
Backtrack
Try to, as it were, change your mind. That is:

1. Unless the stack is empty, **pop** the top backtrack point off the backtrack stack and
   a. Set the tree, subgoal and input from it
   b. **Choose** a rule from its set of rules
   c. Go back to step (1b) of the algorithm
2. Otherwise (the stack *is* empty)
   - We lose!
   - There is no parse for the input with the grammar

We'll see the operation of this algorithm in detail in this week's lab

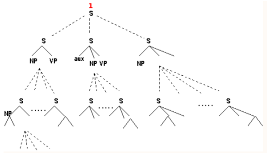
13. Search Strategies

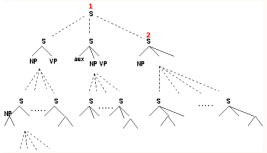
Schematic view of the top-down search space:

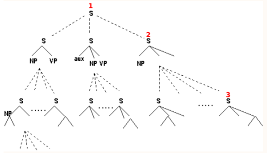


In **depth-first** search the parser

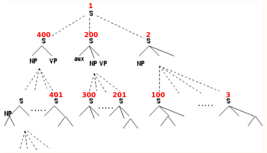
- explores one branch of the search space at a time
- For example, using a **stack** (last-in, first-out) of incomplete trees to try to expand

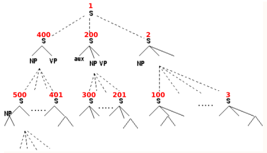






. . .

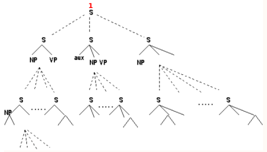


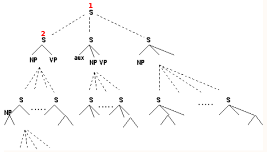


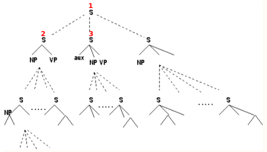
- If a branch of the space is a dead-end, it needs to **backtrack**

In **breadth-first** search the parser

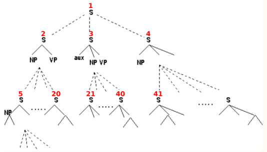
- explores all possible branches in parallel
- For example, using a **queue** (first-in, first out) of incomplete trees to try to expand

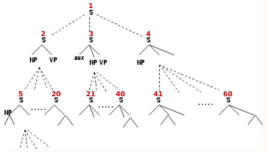






. . .





The **bottom-up** search space works, as the name implies, from the leaves upwards

- Trying to build and combine small trees into larger ones
- The parser we look at in detail in the next lectures works that way

14. 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**

Although **Shift-reduce parsing** also *searches* **depth-first**, in contrast it *builds* structures **bottom-up**.

It does this by repeatedly

1. **shifting** terminal symbols from the input string onto a stack
2. **reducing** some elements of the stack to the LHS side of a rule when they match its RHS

As described, this is just a recogniser

- You win if you end up with a single 'S' on the stack and no more input

Actual *parsing* requires more bookkeeping

Given certain constraints, it is possible to pre-compute auxiliary information about the grammar and exploit it during parsing so that no backtracking is required.

Modern computer languages are often parsed this way

- But grammars for natural languages don't (usually) satisfy the relevant constraints

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 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 analysed in more than one way

- even if not all these sub-string analyses 'survive'
- 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

All 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":

1. **Structural ambiguity** in the grammar and **lexical ambiguity** in the words (that is, words occurring under more than one part of speech) may lead the parser down a wrong path
2. So the same sub-tree may be built several times
   - whenever a path fails, the parser abandons any subtrees computed since the last backtrack point, 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.

- We'll experience this first-hand in this week's lab

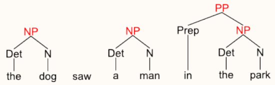
Chart parsing is the name given to a family of solutions to this problem

17. Dynamic Programming

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



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.

18. Parsing as dynamic programming

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

- A process sometimes called **memoisation**

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 can be memoised
- in a **chart**
- also know 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

- **Active** chart parsing goes further, and uses the chart for partial results as well

19. 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 of items from the input
  - That is, 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
- A cell can contain
  - information about the **type** of constituent (or constituents) that span(s) the substring
  - pointers to its sub-constituents
  - (In the case of **active** chart parsers, **predictions** about what constituents might follow the substring)

20. Depicting a WFST as a matrix

Here's a sample matrix, part-way through a parse

	1	2	3	4	5	6
0	V					
1		Prep		PP		
2			Det	NP		
3				N		
4						
5						

$_0$ See $_1$ with $_2$ a $_3$ telescope $_4$ in $_5$ hand $_6$

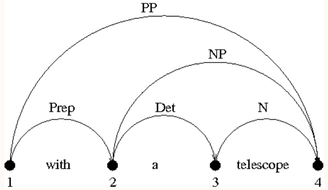
We can read this as saying:

- 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**
        - bottom-up chart parsers
            - May include top-down filtering
        - top-down chart parsers
            - May include bottom-up 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

$A \rightarrow B\ C$

$A \rightarrow a$

- (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*

We can enter constituent `A` in cell `(i,j)` iff either

- there is a rule `A → b` and
    - `b` is found in cell `(i,j)`
- or if there is a rule `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
-

	1	2	3	4	5	6
0	🟩 V	🟩	🟩	🟥		
1		🟩 Prep	🟩	🟩 PP		
2			🟩 Det	🟩 NP		
3				🟩 N		
4						
5						

Note that this process manifests the fundamental weakness of blind *bottom-up* parsing:

- Large numbers of constituents will be found which do not participate in the ultimately spanning 'correct' analyses.

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 occuring (only) on the LHS 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