Chart Parsing: the CYK Algorithm

Informatics 2A: Lecture 16

Bonnie Webber

School of Informatics
University of Edinburgh
bonnie@inf.ed.ac.uk

30 October 2009

1. Problems with Parsing as Search
   - Grammar Restructuring
   - Complexity
2. The CYK Algorithm
   - Parsing as Dynamic Programming
   - The CYK Algorithm
   - Visualizing the Chart
   - Properties of the Algorithm

Reading:

J&M (2nd ed), ch. 13, Section 13.3 – 13.4
NLTK Book, ch. 8 (Analyzing Sentence Structure), Section 8.4

Grammar Restructuring

Deterministic parsing (Lectures 14 and 15) aims to address a limited amount of local ambiguity – the problem of not being able to decide uniquely which grammar rule to use next in a left-to-right analysis of the input string, even if the string is not globally ambiguous.

By re-structuring the grammar, the parser can make a unique decision, based on a limited amount of look-ahead.

Recursive Descent parsing (Lecture 13) demands grammar restructuring, in order to eliminate left-recursive rules that can get it into a hopeless loop.

Left Recursion

But grammars for natural human languages should be revealing, allowing strings to be associated with meaning (semantics) in a systematic way. Re-structuring the grammar may destroy this.

Example: (Indirectly) left-recursive rules are needed in English.

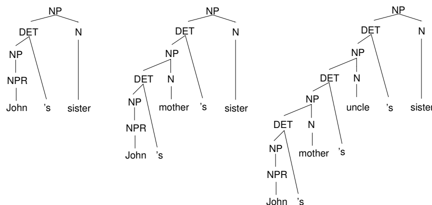
NP → DET N
NP → NPR
DET → NP 's

These rules generate NPs with possesive modifiers such as:

John's sister
John's mother's sister
John's mother's uncle's sister
John's mother's uncle's sister's niece

Left Recursion

Tree structures for possessives:



We don't want to re-structure our grammar rules just to be able to use a particular approach to parsing. Need an alternative.

Complexity

Recursive descent parsing demonstrates 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.

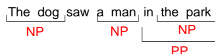
The complexity of this blind backtracking is exponential in the worst case because of repeated re-analysis of the same sub-string.

The next 4 lectures will look at other ways of handling ambiguity:

- Chart parsing: using the parser alone;
- Probabilistic Grammars: using both grammar and parser.

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 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

Dynamic Programming:

- Given a problem, systematically fill a table of solutions to sub-problems: this is called memoization.
- Once solutions to all sub-problems have been accumulated, solve the overall problem by composing them..

For parsing, the sub-problems are analyses of sub-strings, and they are memoized in a chart (aka well-formed substring table, WFST). Each analysis may correspond to:

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

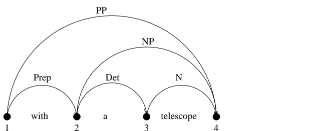
	1	2	3	4	5	6
0	V					
1		Prep		PP		
2			Det	NP		
3				N		
4						
5						

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

Depicting a WFST as a Graph

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

- 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.



Algorithms for Chart Parsing

Important members of the chart parsing family include:

- the CYK 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

CYK Algorithm

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

The simplest version of CYK is for a CFG whose rules have at most two symbols on their RHS.

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, CYK 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 CYK Algorithm

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

BUILD_CYK_CHART(t,[$w_1, \ldots, w_n$])
  for j $\leftarrow$ 1 **to** n
    do
      t(j-1, j) $\leftarrow$ Close({w_j})
  for k $\leftarrow$ 1 **to** n
    for j $\leftarrow$ k **to** n
      for m $\leftarrow$ 1 **to** k-1
      do
        t(j-k, j) $\leftarrow$ t(j-k, j) $\cup$ Close({A | A $\rightarrow$ B C
          for some B $\in$ t(j-k, j-m) and C $\in$ t(j-m, j)})

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

Visualizing the Chart

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

Visualizing the Chart

	1	2	3	4
0				
1				
2				
3				
	the	frogs	ate	fish

Visualizing the Chart (0,1)

	1	2	3	4
0	det			
1				
2				
3				
	the	frogs	ate	fish

Unary branching rules: det → the

Visualizing the Chart (1,2)

	1	2	3	4
0	det			
1		n nom np		
2				
3				
	the	frogs	ate	fish

Unary branching rules: N → frogs, Nom → N, NP → Nom

Visualizing the Chart (2,3)

	1	2	3	4
0	det			
1		n nom np		
2			tv	
3				
	the	frogs	ate	fish

Unary branching rules: tv → ate

Visualizing the Chart (3,4)

	1	2	3	4
0	det			
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

Unary branching rules: N → frogs, Nom → N, NP → Nom,
iv → fish, vp → iv

Visualizing the Chart (0,2)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: NP → Det Nom $(0,1)$ & $(1,2)$ $\rightsquigarrow$ $(0,2)$

Visualizing the Chart (1,3)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

$(1,2)$ & $(2,3)$ $\not\rightsquigarrow$

Visualizing the Chart (2,4)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: VP → TV NP $(2,3)$ & $(3,4)$ $\rightsquigarrow$ $(2,4)$

Visualizing the Chart (1,4)

	1	2	3	4
0	det	np		
1		n nom np		s
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: S → NP VP $(1,2)$ & $(2,4)$ $\rightsquigarrow$ $(1,4)$
$(1,3)$ & $(3,4)$ $\not\rightsquigarrow$

Visualizing the Chart (0,4)

	1	2	3	4
0	det	np		s
1		n nom np		s
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: S → NP VP (0,1) & (1,4) ↛
(0,2) & (2,4) ⇝ (0,4) (0,3) & (3,4) ↛

Questions about CYK

1. Does the CYK algorithm ever stop before it reaches the end of the string?
2. How does the CYK algorithm show that a string belongs to the language?
3. Does CYK have to proceed left-to-right? That is, could one similarly guarantee that no constituent would be missed if CYK proceeded right-to-left?

From CYK Recognizer to CYK Parser

4. Can we tell from the CYK chart what the syntactic analysis (tree structure) is from *the frogs ate fish*?

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).

	1	2	3	4
0	det	np (( ))		S (( ))
1		n nom (( )) np (( ))		S (( ))
2			tv	vp (( ))
3				n nom (( )) np (( ))
	the	frogs	ate	soup

Summary

- Recursive decent parsing cannot handle left-recursive rules and is inefficient (exponential).
- Alternative: Use dynamic programming and memoize sub-analysis in a chart to avoid duplicate work.
- The chart can be visualized as a graph or as a matrix.
- The CYK algorithm builds a chart in $O(n^3)$ time. It is specified as a recognizer, but can be used as a parser, if more information is recorded in the chart.