

Unit-II

Syntax

- Parsing Natural Languages
- Tree Banks: A Data Driven Approach to Syntax
- Representation of Syntactic Structure
- Parsing Algorithms
- Models for Ambiguity Resolution in Parsing
- Multilingual Issues

Syntax-Introduction

- Parsing uncovers the hidden structure of linguistic input.
- In Natural Language Applications the predicate structure of sentences can be useful.
- In NLP the syntactic analysis of the input can vary from:
 - Very low level- POS tagging.
 - Very high level- recovering a structural analysis that identifies the dependency between predicates and arguments in the sentence.
- The major problem in parsing natural language is the problem of ambiguity.

Parsing Natural Languages

- Let us look at the following spoken sentences:
 - He wanted to go for a drive in movie.
 - He wanted to go for a drive in the country.
- There is a natural pause between drive and in in the second sentence.
- This gap reflects an underlying hidden structure to the sentence.
- Parsing provides a structural description that identifies such a break in the intonation.

Parsing Natural Languages

- Let us look at another sentence:
 - The cat who lives dangerously had nine lives.
- A text-to-speech system needs to know that the first instance of the word lives is a verb and the second instance a noun.
- This is an instance of POS tagging problem.
- Another important application where parsing is important is text summarization.

Parsing Natural Languages

- Let us look at examples for summarization:
 - Beyond the basic level, the operations of the three products vary widely.
- The above sentence can be summarized as follows:
 - The operations of the products vary.
- To do this task we first parse the first sentence.

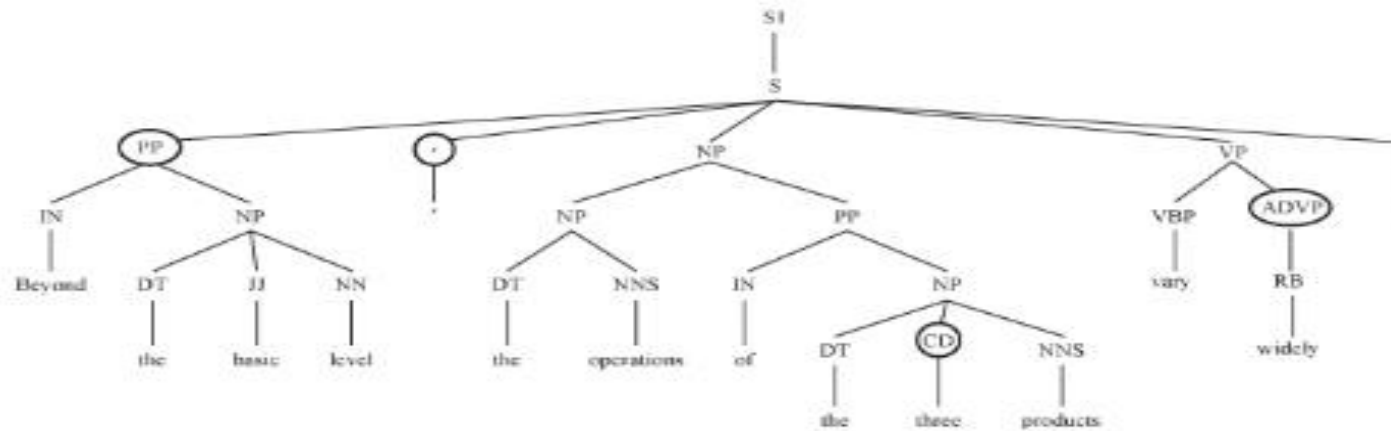


Figure 3–1. Parser output for sentence

Parsing Natural Languages

- Deleting the circled constituents PP, CD and ADVP in the previous diagram results in the short sentence.
- Let us look at another example:
 - Open borders imply increasing racial fragmentation in EUROPEAN COUNTRIES.
- In the above example the capitalized phrase can be replaced with other phrases without changing the meaning of the sentence.
 - Open borders imply increasing racial fragmentation in *the countries of Europe*.
 - Open borders imply increasing racial fragmentation in *European states*.
 - Open borders imply increasing racial fragmentation in *Europe*.
 - Open borders imply increasing racial fragmentation in *European Nations*.
 - Open borders imply increasing racial fragmentation in *the European countries*.

Parsing Natural Languages

- In NLP syntactic parsing is used in many applications like:
 - Statistical Machine Translation
 - Information extraction from text collections
 - Language summarization
 - Producing entity grids for language generation
 - Error correction in text
 - Knowledge acquisition from language

Trebanks: A Data-Driven Approach to Syntax

- Parsing recovers information that is not explicit in the input sentence.
- Parsers require some additional knowledge beyond the input sentence that should be produced as output.
- We can write down the rules of the syntax of a sentence as a CFG.
- Here we have a CFG which represents a simple grammar of transitive verbs in English (verbs that have a subject and object noun phrase (NP), plus modifiers of verb phrases (VP) in the form of prepositional phrases (PP)).

Treebanks: A Data-Driven Approach to Syntax

S -> NP VP

NP -> 'John' | 'pockets' | D N | NP PP

VP -> V NP | VP PP

V -> 'bought'

D -> 'a'

N -> 'shirt'

PP -> P NP

P -> 'with'

The above CFG can produce the syntax analysis of a sentence like:
John bought a shirt with pockets

Treebanks: A Data-Driven Approach to Syntax

- Parsing the previous sentence gives us two possible derivations.

(S (NP John)
 (VP (VP (V bought)
 (NP (D a)
 (N shirt)))
 (PP (P with)
 (NP pockets))))

(S (NP John)
 (VP (V bought)
 (NP (NP (D a)
 (N shirt))
 (PP (P with)
 (NP pockets))))))

Treebanks: A Data-Driven Approach to Syntax

- Writing a CFG for the syntactic analysis of natural language is problematic.
- A simple list of rules does not consider interactions between different components in the grammar.
- Listing all possible syntactic constructions in a language is a difficult task.
- It is difficult to exhaustively list lexical properties of words. This is a typical knowledge acquisition problem.
- One more problem is that the rules interact with each other in combinatorially explosive ways.

Treebanks: A Data-Driven Approach to Syntax

- Let us look at an example of noun phrases as a binary branching tree.
- $N \rightarrow NN$ (Recursive Rule)
- $N \rightarrow \text{'natural' | 'language' | 'processing' | 'book'}$
- For the input 'natural language processing' the recursive rules produce two ambiguous parses.

```
(N (N (N natural)
      (N language))
  (N processing))
```

```
(N (N natural)
  (N (N language)
      (N processing)))
```

Treebanks: A Data-Driven Approach to Syntax

- For CFGs it can be proved that the number of parsers obtained by using the recursive rule n times is the Catalan number (1,1,2,5,14,42, 132, 429, 1430, 4862,...) of n :

$$\text{Cat}(n) = \frac{1}{n+1} \binom{2n}{n}$$

- For the input “natural language processing book” only one out of the five parsers obtained using the above CFG is correct:

**(N (N (N (N natural)
(N language))
(N processing))
(N book))**

Trebanks: A Data-Driven Approach to Syntax

- This is the second knowledge acquisition problem- We need to know not only the rules but also which analysis is most plausible for a given input sentence.
- The construction of a **tree bank** is a data driven approach to syntax analysis that allows us to address both the knowledge acquisition bottlenecks in one stroke.
- A treebank is a collection of sentences where each sentence is provided a complete syntax analysis.
- The syntax analysis for each sentence has been judged by a human expert.

Treebanks: A Data-Driven Approach to Syntax

- A set of annotation guidelines is written before the annotation process to ensure a consistent scheme of annotation throughout the tree bank.
- No set of syntactic rules are provided by a treebank.
- No exhaustive set of rules are assumed to exist even though assumptions about syntax are implicit in a treebank.
- The consistency of syntax analysis in a treebank is measured using interannotator agreement by having approximately 10% overlapped material annotated by more than one annotator.
- Treebanks provide annotations of syntactic structure for a large sample of sentences.

Trebanks: A Data-Driven Approach to Syntax

- A supervised machine learning method can be used to train the parser.
- Treebanks solve the first knowledge acquisition problem of finding the grammar underlying the syntax analysis because the analysis is directly given instead of a grammar.
- The second problem of knowledge acquisition is also solved by treebanks.
- Each sentence in a treebank has been given its most plausible syntactic analysis.
- Supervised learning algorithms can be used to learn a scoring function over all possible syntax analyses.

Trebanks: A Data-Driven Approach to Syntax

- For real time data the parser uses the scoring function to return the syntax analysis that has the highest score.
- Two main approaches to syntax analysis that are used to construct treebanks are:
 - Dependency graphs
 - Phase structure trees
- These two representations are very closely connected to each other and under some assumptions one can be converted to the other.
- Dependency analysis is used for free word order languages like Indian languages.
- Phrase structure analysis is used to provide additional information about long distance dependencies for languages like English and French.

Treebanks: A Data-Driven Approach to Syntax

- In the discussion to follow we examine three main components for building a parser:
- **The representation of syntactic structure**- it involves the use of a varying amount of linguistic knowledge to build a treebank.
- **The training and decoding algorithms**- they deal with the potentially exponential search space.
- **Methods to model ambiguity**- provides a way to rank parses to recover the most likely parse.

Representation of Syntactic Structure

- Syntax Analysis using dependency graphs
- Syntax Analysis using phrase structure trees

Syntax Analysis using Dependency Graphs

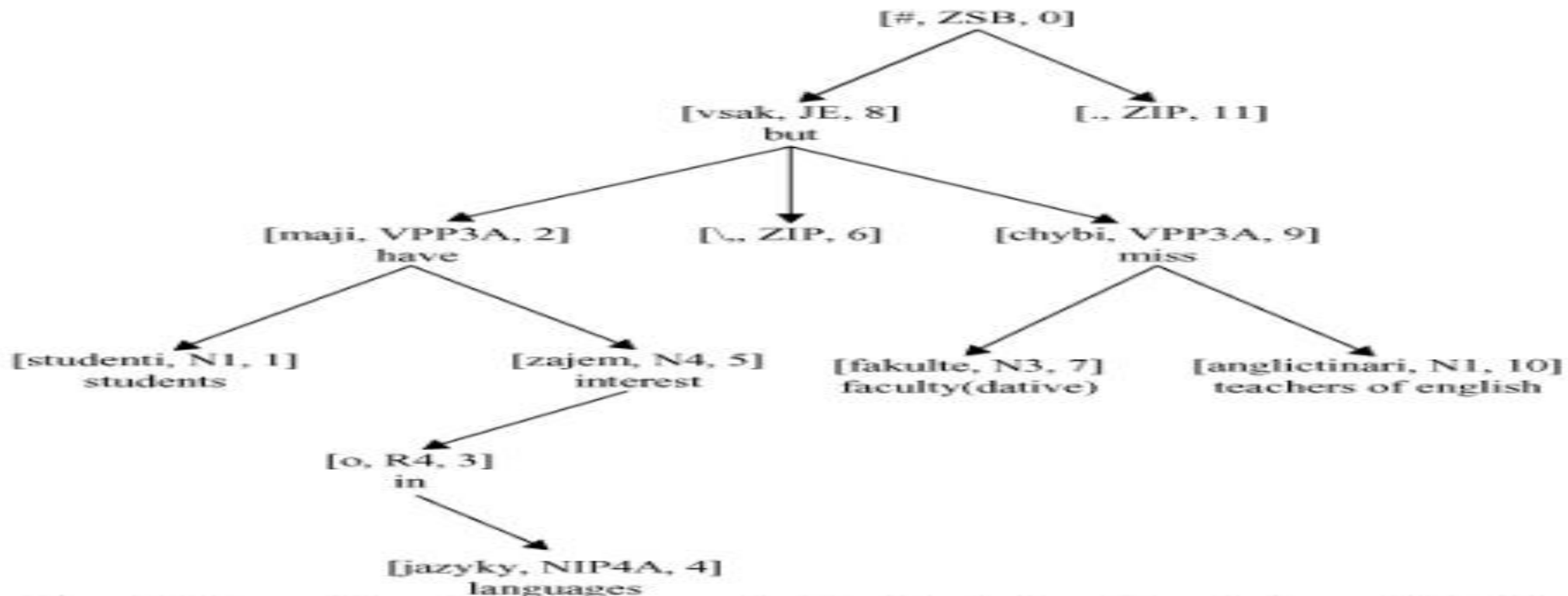
- In dependency graphs the head of a phrase is connected with the dependents in that phrase using directed connections.
- The head-dependent relationship can be semantic (head-modifier) or syntactic (head-specifier).
- The main difference between dependency graphs and phrase structure trees is that dependency analysis make minimal assumptions about syntactic structure.
- Dependency graphs treat the words in the input sentence as the only vertices in the graph which are linked together by directed arcs representing syntactic dependencies.

Syntax Analysis using Dependency Graphs

- One typical definition of dependency graph is as follows:
 - In dependency syntactic parsing the task is to derive a syntactic structure for an input sentence by identifying the syntactic head of each word in the sentence.
 - The nodes are the words of the input sentence and the arcs are the binary relations from head to dependent.
 - It is often assumed that all words except one have a syntactic head.
 - It means that the graph will be a tree with the single independent node as the root.
 - In labeled dependency parsing the parser assigns a specific type to each dependency relation holding between head word and dependent word.
- In the current discussion we will be discussing about dependency trees only where each word depends on exactly one parent either another word or a dummy symbol.

Syntax Analysis using Dependency Graphs

- In dependency trees the 0 index is used to indicate the root symbol and the directed arcs are drawn from head word to the dependent word.



The students are interested in languages, but the faculty is missing teachers of English.

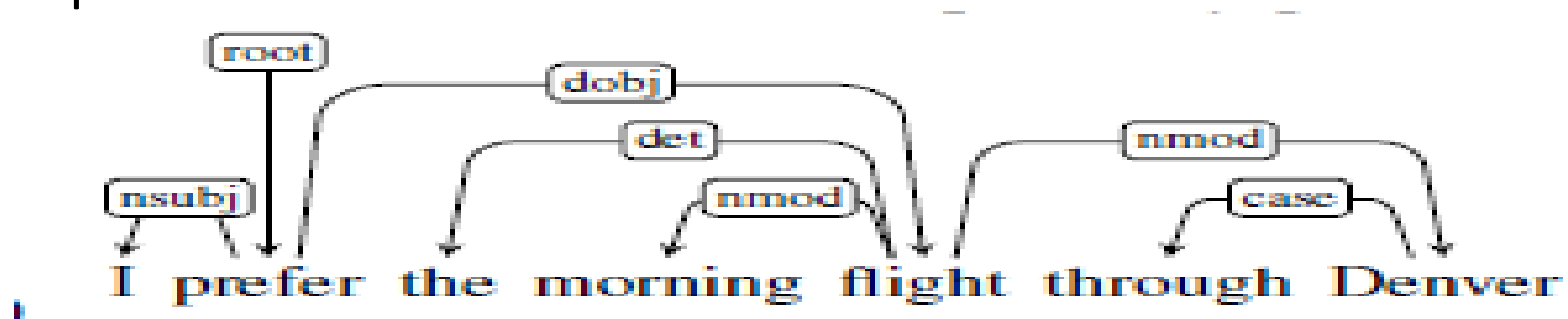
Syntax Analysis using Dependency Graphs

- In the figure in the previous slide [fakulte,N3,7] is the seventh word in the sentence with POS tag N3 and it has dative case.
- Here is a textual representation of a labeled dependency tree:

Index	Word	Part of Speech	Head	Label
1	They	PRP	2	SBJ
2	persuaded	VBD	0	ROOT
3	Mr.	NNP	4	NMOD
4	Trotter	NNP	2	IOBJ
5	to	TO	6	VMOD
6	take	VB	2	OBJ
7	it	PRP	6	OBJ
8	back	RB	6	PRT
9	.	.	2	P

Syntax Analysis using Dependency Graphs

- An important notion in dependency analysis is the notion of **projectivity**.
- A projective dependency tree is one where we put the words in a linear order based on the sentence with the root symbol in the first position.
- The dependency arcs are then drawn above the words without any crossing dependencies.
- Example:



Syntax Analysis using Dependency Graphs

Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	Description
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	Description
CONJ	Conjunct
CC	Coordinating conjunction

Figure 14.2 Selected dependency relations from the Universal Dependency set. (de Marneffe et al., 2014)

Syntax Analysis using Dependency Graphs

- Let us look at an example where a sentence contains an extra position to the right of a noun phrase modifier phrase which requires a crossing dependency.



Figure 3–3. An unlabeled nonprojective dependency tree with a crossing dependency

Syntax Analysis using Dependency Graphs

- English has very few cases in a treebank that needs such a non projective analysis.
- In languages like Czech, Turkish, Telugu the number of non productive dependencies are much higher.
- Let us look at a multilingual comparison of crossing dependencies across a few languages:

	Ar	Ba	Ca	Ch	Cz	En	Gr	Hu	It	Tu
% deps	0.4	2.9	0.1	0.0	1.9	0.3	1.1	2.9	0.5	5.5
% sents	10.1	26.2	2.9	0.0	23.2	6.7	20.3	26.4	7.4	33.3

- Ar=Arabic;Ba=Basque;Ca=Catalan;Ch=Chinese;Cz=Czech;En=English;Gr=Greek;Hu=Hungarian;It=Italian;Tu=Turkish

Syntax Analysis using Dependency Graphs

- Dependency graphs in treebanks do not explicitly distinguish between projective and non-projective dependency tree analyses.
- Parsing algorithms are sometimes forced to distinguish between projective and non-projective dependencies.
- Let us try to setup dependency links in a CFG.

Syntax Analysis using Dependency Graphs

$X0_2 \rightarrow X0_1^* X2_1$

$X0_1 \rightarrow x0^*$

$X2_1 \rightarrow X1_1 X2_2^*$

$X1_1 \rightarrow x1^*$

$X2_2 \rightarrow X2_3^* X3_1$

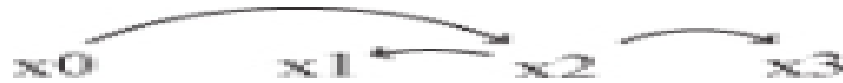
$X2_3 \rightarrow x2^*$

$X3_1 \rightarrow x3^*$

- In the CFG the terminal symbols are $x0, x1, x2, x3$.
- The asterisk picks out a single symbol in the right hand side of each rule that specifies the dependency link.
- We can look at the asterisk as either a separate annotation on the non-terminal or simply as a new nonterminal in the probabilistic context-free grammar(PCFG).

Syntax Analysis using Dependency Graphs

- The dependency tree equivalent to the preceding CFG is as follows:



- If we can convert a dependency tree into an equivalent CFG then the dependency tree must be projective.

Syntax Analysis using Dependency Graphs

- In a CFG converted from a dependency tree we have only the following three types of rules:
 - One type of rule to introduce the terminal symbol
 - Two rules where Y is dependent on X or vice-versa.
- The head word of X or Y can be traced by following the asterisk symbol.

$Z \rightarrow X^* Y$

$Z \rightarrow X Y^*$

$A \rightarrow a^*$

Syntax Analysis using Dependency Graphs

- Now let us look at an example non-projective dependency tree:



- When we convert this dependency tree to a CFG with * notation it can only capture the fact that X3 depends on X2 or X1 depends on X3.

X2_3 -> X1_1 X2_2*

X1_1 -> x1

X2_2 -> X2_1* X3_1

X2_1 -> x2

X3_1 -> x3

X2_3 -> X1_1 X3_2*

X1_1 -> x1

X3_2 -> X2_1 X3_1*

X2_1 -> x2

X3_1 -> x3

Syntax Analysis using Dependency Graphs

- There is no CFG that can capture the non-projective dependency.
- Projective dependency can be defined as follows:
 - For each word in the sentence its descendants form a contiguous substring of the sentence.
- Non-Projectivity can be defined as follows:
 - A non-projective dependency means that there is a word in the sentence such that its descendants do not form a contiguous substring of the sentence.
- In non-projective dependency there is a non-terminal Z such that Z derives spans (x_i, x_k) and (x_{k+p}, x_j) for some $p > 0$.

Syntax Analysis using Dependency Graphs

- It means there must be a rule $Z \rightarrow PQ$ where P derives (x_1, x_k) and Q derives (x_{k+p}, x_j) .
- By the definition of CFG this is valid if $p=0$ because P and Q must be continuous substrings.
- Hence non-projective dependency can not be converted to a CFG.

Syntax Analysis using Phrase Structure Trees

- A phrase structure syntax analysis can be viewed as implicitly having a predicate argument structure associated with it.
- Now let us look at an example sentence:
 - Mr. Baker seems especially sensitive

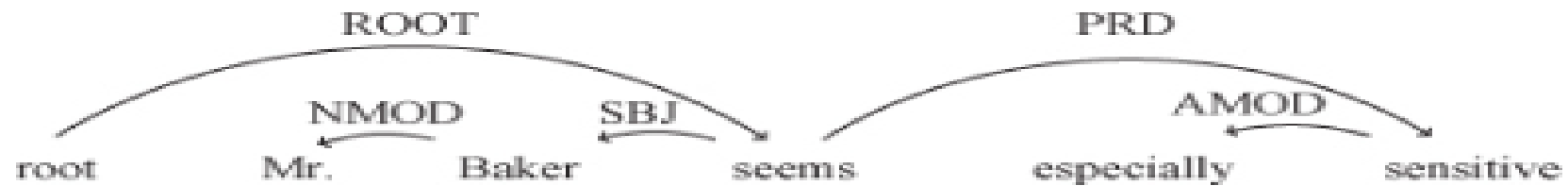
```
(S (NP-SBJ (NNP Mr.)  
          (NNP Baker))  
  (VP (VBZ seems)  
      (ADJP-PRD (RB especially)  
                (JJ sensitive))))
```

Predicate-argument structure:

```
seems((especially(sensitive))(Mr. Baker))
```

Syntax Analysis using Phrase Structure Trees

- The same sentence gets the following dependency tree analysis.



- We can find some similarity between Phrase Structure Trees and Dependency Trees.
- We now look at some examples of Phrase Structure Analysis in tree banks and see how null elements are used to localize certain predicate-argument dependencies in the tree structure.

Syntax Analysis using Phrase Structure Trees

- In this example we can see that an NP dominates a trace *T* which is a null element (same as ϵ).
- The empty trace has an index and is associated with the WHNP (WH-noun phrase) constituent with the same index.
- This co-indexing allows us to infer the predicate-argument structure shown in the tree.

```
(SBARQ (WHNP-1 What)
  (SQ is (NP-SBJ Tim)
    (VP eating (NP *T*-1))))
?)
```

Predicate-argument structure:
`eat(Tim, what)`

Syntax Analysis using Phrase Structure Trees

- In this example “the ball” is actually not the logical subject of the predicate.
- It has been displaced due to the passive construction.
- “Chris” the actual subject is marked as LGS (Logical subjects in passives) enabling the recovery of predicate argument structure for the sentence.

```
(S (NP-SBJ-1 The ball)
  (VP was (VP thrown)
    (NP *-1)
    (PP by (NP-LGS Chris))))
```

Predicate-argument structure:
throw(Chris, the ball)

Syntax Analysis using Phrase Structure Trees

- In this example we can see that different syntactic phenomena are combined in the corpus.
- Both the analysis are combined to provide the predicate argument structure in such cases.

```
(SBARQ (WHNP-1 Who)
  (SQ was (NP-SBJ-2 *T*-1)
    (VP believed (S (NP-SBJ-3 *-2)
      (VP to (VP have
        (VP been
          (VP shot
            (NP *-3))))))))))
  ?)
```

Predicate-argument structure:

```
believe(*someone*, shoot(*someone*, who))
```


Syntax Analysis using Phrase Structure Trees

- Here we see a pair of examples to show how null elements are used to annotate the presence of a subject for a predicate even if it is not explicit in the sentence.
- In the first case the analysis marks the missing subject for “take back” as the object of the verb *persuaded*.
- In the second case the missing subject for “take back ” is the subject of the verb *promised*.

Syntax Analysis using Phrase Structure Trees

(S (NP-SBJ (PRP They))
(VP (VP (VBD persuaded)
(NP-1 (NNP Mr.)
(NNP Trotter))
(S (NP-SBJ (-NONE- *-1))
(VP (TO to)
(VP (VB take)
(NP (PRP it))
(PRT (RB back))))))))))

Predicate argument structure:

persuade(they, Mr. Trotter, take_back(Mr. Trotter, it))

(S (NP-SBJ-1 (PRP They))
(VP (VP (VBD promised)
(NP (NNP Mr.)
(NNP Trotter))
(S (NP-SBJ (-NONE- *-1))
(VP (TO to)
(VP (VB take)
(NP (PRP it))
(PRT (RB back))))))))))

Predicate argument structure:

promise(they, Mr. Trotter, take_back(they, it))

Syntax Analysis using Phrase Structure Trees

- The dependency analysis for “persuaded” and “promised” do not make such a distinction.
- The dependency analysis for the two sentences is as follows:

1	They	PRP	2	SBJ
2	persuaded	VBD	0	ROOT
3	Mr.	NNP	4	NMOD
4	Trotter	NNP	2	IOBJ
5	to	TO	6	VMOD
6	take	VB	2	OBJ
7	it	PRP	6	OBJ
8	back	RB	6	PRT
9	.	.	2	P

1	They	PRP	2	SBJ
2	promised	VBD	0	ROOT
3	Mr.	NNP	4	NMOD
4	Trotter	NNP	2	IOBJ
5	to	TO	6	VMOD
6	take	VB	2	OBJ
7	it	PRP	6	OBJ
8	back	RB	6	PRT
9	.	.	2	P

Syntax Analysis using Phrase Structure Trees

- Most statistical parsers trained using Phrase Structure treebanks ignore these differences.
- Here we look at one example from the Chinese treebank which uses IP instead of S.
- This is a move from transformational grammar-based phrase structure to government-binding (GB) based phrase structure.
- It is very difficult to take a CFG-based parser initially developed for English parsing and adapt it to Chinese parsing by training it on Chinese phrase structure treebank.

Syntax Analysis using Phrase Structure Trees

```
(IP (NP-SBJ (NP (NN 结售/settlement and sale)
                (NN 制度/system))
         (CC 和/and)
         (NP (CP (WHNP-2 (-NONE- *OP*))
                 (CP (IP (NP-SBJ (-NONE- *T*-2))
                         (VP (VA 新/new)))
                    (DEC 的)))
                (NP (NN 核销/verification and cancellation)
                    (NN 制度/system))))))
 (VP (PP-LOC (P 在/in)
            (NP-PN (NR 西藏/Tibet)))
     (ADVP (AD 全面/fully))
     (VP (VV 实施/operating))))
```

English translation:

A (foreign exchange) settlement and sale system and a verification and cancellation system that is newly created is fully operational in Tibet.

Parsing Algorithms

- Introduction to Parsing Algorithms
- Shift-Reduce Parsing
- Hypergraphs and Chart Parsing
- Minimum Spanning Trees and Dependency Parsing

Parsing Algorithms

- Given an input sentence a parser produces an output analysis of that sentence.
- The analysis will be consistent with the tree-bank used to train the parser.
- Tree-banks do not need to have an explicit grammar.
- Here for better understanding we assume the existence of a CFG.

Parsing Algorithms

- Let us look at an example of a CFG that is used to derive strings such as “a and b or c” from the start symbol N:

$N \rightarrow N \text{ 'and' } N$

$N \rightarrow N \text{ 'or' } N$

$N \rightarrow \text{'a' | 'b' | 'c'}$

```
N
=> N 'or' N
=> N 'or c'
=> N 'and' N 'or c'
=> N 'and b or c'
=> 'a and b or c'
```

- An important concept of parsing is a **derivation**.
- In the derivation each line is called **sentential form**.
- The method of derivation used here is called **rightmost derivation**.

Parsing Algorithms

- An interesting property of rightmost derivation is revealed if we arrange the derivation in reverse order.

		(N (N (N a)
'a and b or c'		and
=> N 'and b or c'	# use rule N -> a	(N b))
=> N 'and' N 'or c'	# use rule N -> b	or
=> N 'or c'	# use rule N -> N and N	(N c))
=> N 'or' N	# use rule N -> c	
=> N	# use rule N -> N or N	

- The above sequence corresponds to the construction of the above parse tree from left to right, one symbol at a time.

Parsing Algorithms

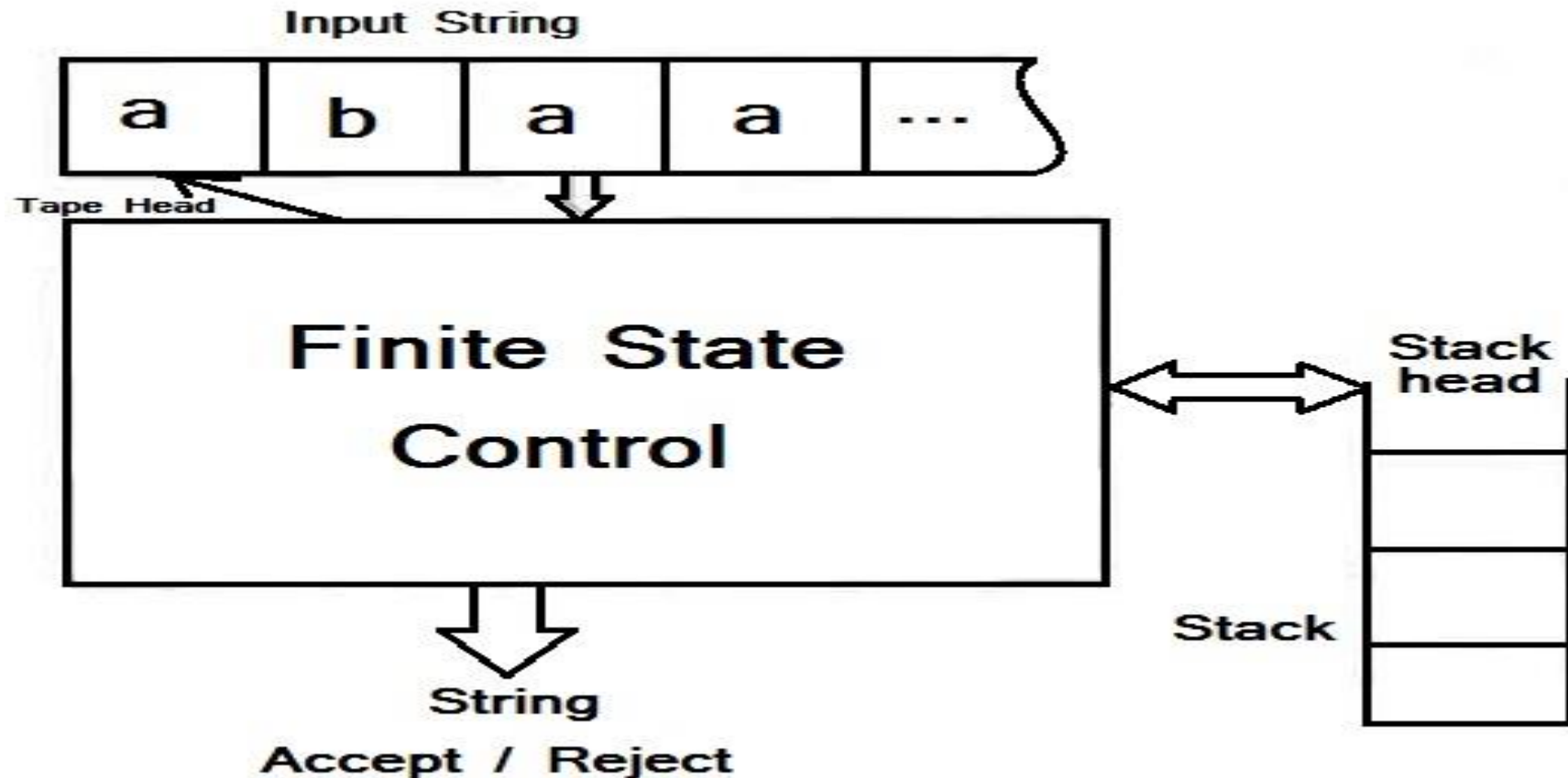
- There can be many parse trees for the given input string.
- Another rightmost derivation for the given input string is as follows:

```
(N (N a)
  and
  (N (N b)
    or
    (N c)))
```

```
'a and b or c'
=> N 'and b or c'      # use rule N -> a
=> N 'and' N 'or c'    # use rule N -> b
=> N 'and' N 'or' N    # use rule N -> c
=> N 'and' N           # use rule N -> N or N
=> N                  # use rule N -> N and N
```

Shift-Reduce Parsing

- Every CFG has an automaton equivalent to it called the pushdown automaton.



Shift-Reduce Parsing


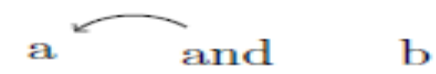
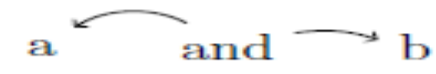
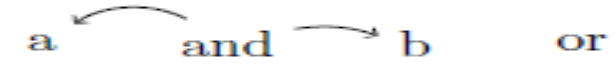

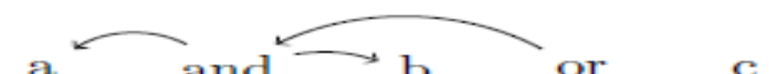
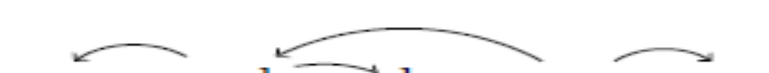
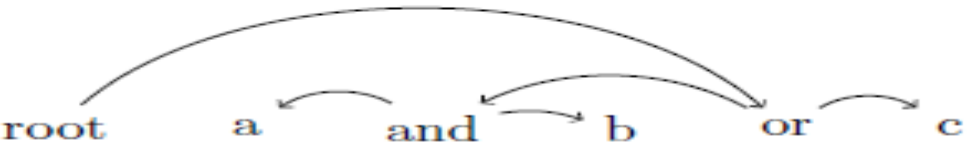
- The shift-reduce parsing algorithm is defined as follows:
 1. Start with an empty stack and the buffer contains the input string.
 2. Exit with success if the top of the stack contains the start symbol of the grammar and if the buffer is empty.
 3. Choose between the following two steps (if the choice is ambiguous, choose one based on an oracle):
 - Shift a symbol from the buffer onto the stack.
 - If the top k symbols of the stack are $\alpha_1 \dots \alpha_k$ which corresponds to the right-hand side of a CFG rule $A \rightarrow \alpha_1 \dots \alpha_k$ then replace the top k symbols with the left-hand side non-terminal A .
 4. Exit with failure if no action can be taken in previous step.
 5. Else, go to Step 2.

Shift-Reduce Parsing

- For the example “a and b or c” the parsing steps are s follows:

Parse Tree	Stack	Input	Action
		a and b or c	Init
a	a	and b or c	shift a
(N a)	N	and b or c	reduce N \rightarrow a
(N a) and	N and	b or c	shift and
(N a) and b	N and b	or c	shift b
(N a) and (N b)	N and N	or c	reduce N \rightarrow b
(N (N a) and (N b))	N	or c	reduce N \rightarrow a
(N (N a) and (N b)) or	N or	c	shift or
(N (N a) and (N b)) or c	N or c		shift c
(N (N a) and (N b)) or (N c)	N or N		reduce N \rightarrow c
(N (N (N a) and (N b)) or (N c))	N		reduce N \rightarrow N or N
(N (N (N a) and (N b)) or (N c))	N		Accept!

Shift-Reduce Parsing- for dependency parsing

Dependency tree	Stack	Input	Action
root	root	a and b or c	Init
root a	root a	and b or c	shift a
root a and	root a and	b or c	shift and
	root and	b or c	a ← and
	root and b	or c	shift b
	root and	or c	and → b
	root and or	c	shift or
	root or	c	and ← or
	root or c		shift c
	root or		or → c
	root		root → or

Hyper Graphs and Chart Parsing

- Shift-reduce parsing allows a linear time parser but requires access to an oracle.
- CFGs in the worst case need backtracking and have a worst case parsing algorithm which run in $O(n^3)$ where n is the size of the input.
- Variants of this algorithm are used in statistical parsers that attempt to search the space of possible parse trees without the limitation of left-to-right parsing.

Hyper Graphs and Chart Parsing

- Our example CFG G is rewritten as new CFG G_c which contains up to two non-terminals on the right hand side.

$N \rightarrow N \text{'and'} N$

$N \rightarrow N \text{'or'} N$

$N \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'}$

$N \rightarrow N N^\wedge$

$N^\wedge \rightarrow \text{'and'} N$

$N \rightarrow N N_v$

$N_v \rightarrow \text{'or'} N$

$N \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'}$

Hyper Graphs and Chart Parsing

- We can specialize the CFG G_c to a particular input string by creating a new CFG that represents all possible parse trees that are valid in grammar G_c for this particular input sentence.
- For the input “a and b or c” the new CFG C_f that represents the forest of parse trees can be constructed.
- Let the input string be broken up into spans 0 a 1 and 2 b 3 or 4 c 5.

Hyper Graphs and Chart Parsing

```
N[0,5] -> N[0,1] N^[1,5]
N[0,3] -> N[0,1] N^[1,3]
N^[1,3] -> 'and'[1,2] N[2,3]
N^[1,5] -> 'and'[1,2] N[2,5]
N[0,5] -> N[0,3] Nv[3,5]
N[2,5] -> N[2,3] Nv[3,5]
Nv[3,5] -> 'or'[3,4] N[4,5]
N[0,1] -> 'a'[0,1]
N[2,3] -> 'b'[2,3]
N[4,5] -> 'c'[4,5]
```

Hyper Graphs and Chart Parsing

- Here a parsing algorithm is defined as taking as input a CFG and an input string and producing a specialized CFG that represents all legal parsers for the input.
- A parser has to create all the valid specialized rules from the start symbol nonterminal that spans the entire string to the leaf nodes that are the input tokens.

Hyper Graphs and Chart Parsing

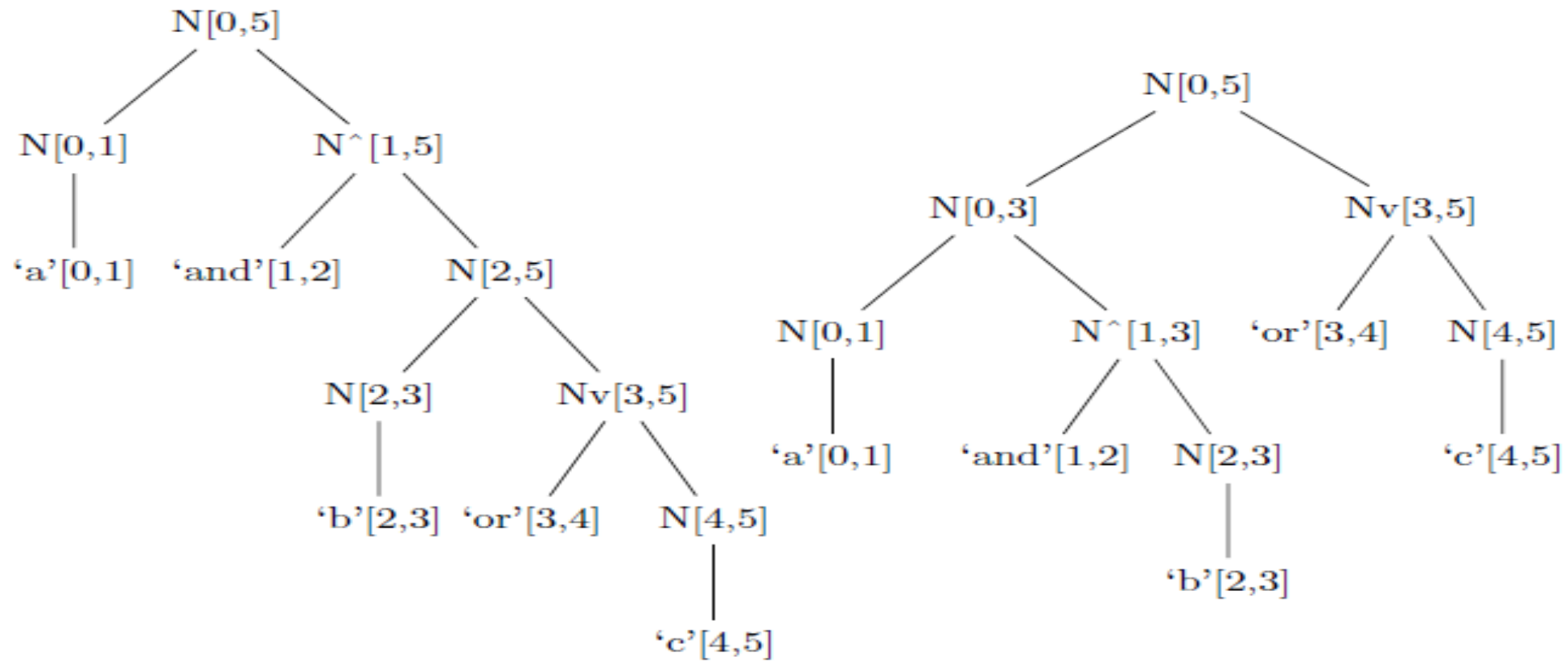


Figure 4: Parse trees embedded in the *specialized* CFG for a particular input string. The nodes with the same label, e.g. $N[0,5]$, $N[0,1]$, $and[1,2]$, $N[2,3]$, and $Nv[3,5]$ can be merged to form a hypergraph representation of all parses for the input.

Hyper Graphs and Chart Parsing

- Now let us look at the steps the parser has to take to construct a specialized CFG.
- Let us consider the rules that generate only lexical items:

$N[0,1] \rightarrow 'a'[0,1]$

$N[2,3] \rightarrow 'b'[2,3]$

$N[4,5] \rightarrow 'c'[4,5]$

- These rules can be constructed by checking for the existence of the rule of the type $N \rightarrow x$ for any input token x and creating a specialized rule for token x .

Hyper Graphs and Chart Parsing

for $i = 0 \dots n$ do

 if $N \rightarrow x$ with score s for any x spanning $i, i + 1$ exists

 then

 add specialized rule $N[i, i + 1] \rightarrow x[i, i + 1]$ with score s

 written as $N[i, i + 1] : s$

 end if

end for

Hyper Graphs and Chart Parsing

- We now can create new specialized rules based on previously created rules in the following way:
 - Let $Y[i,k]$ and $Z[k,j]$ be the left hand side of previously created specialized rules
 - If the CFG has a rule of type $X \rightarrow YZ$
 - We infer that there must be a specialized rule $X[i,j] \rightarrow Y[i,k]Z[k,j]$
 - Each nonterminal span is assigned a score s , $X[i,j]:s$
 - The highest scoring span for each non terminal is retained $X[i,j] = \max_s X[i,j]:s$

Hyper Graphs and Chart Parsing

$X[i, j] = \max_s X[i, j] : s.$

for $j = 2 \dots n$ do

 for $i = j - 1 \dots 0$ do

 for $k = i + 1 \dots j$ do

 if $Y[i, k] : s_1$ and $Z[k, j] : s_2$ are in the specialized grammar then

 if $X \rightarrow YZ$ with score s exists in the original grammar then

 add specialized rule $X[i, j] \rightarrow Y[i, k]Z[k, j]$ with score $s + s_1 + s_2$

 end if

 end if

 end for

 end for

end for

Hyper Graphs and Chart Parsing

- The previous algorithm is called as CYK algorithm.
- It considers every span of every length, splits up that span in every possible way to see if a CFG rule can be used to derive the span.
- The algorithm takes n^3 time for an input of size n .
- Picking the most likely trees by using supervised learning will take no more than n^3 time.
- For a given span i,j and for each non-terminal X we only keep the highest scoring way to reach $X[i,j]$.
- Starting from the start symbol that spans the entire string $S[0,n]$ gives us the best parsing tree.

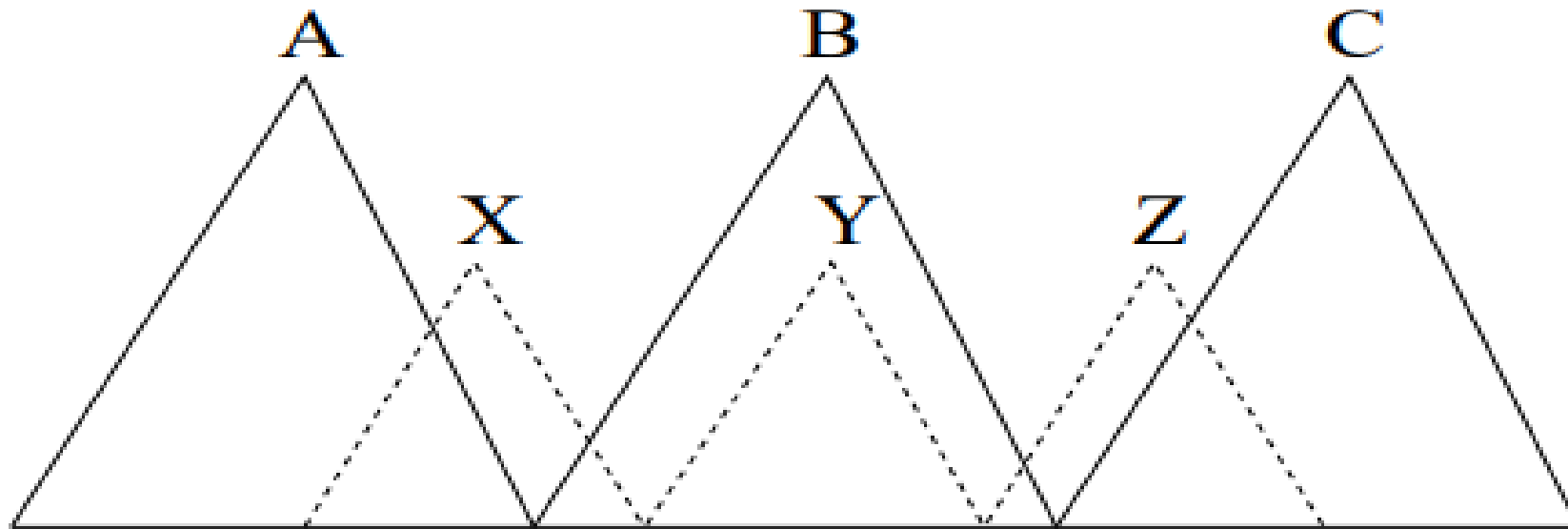
Hyper Graphs and Chart Parsing

- In a probabilistic setting where the score can be interpreted as log probabilities, this is called a **Viterbi-best** parse.
- Each cell contains the log probability of deriving the string $w[i,j]$ starting with nonterminal X , which can be written as $\Pr(X \Rightarrow^* w[i,j])$. (generating)
- The utility of a nonterminal X at a particular span i,j depends on reaching the start symbol S which is captured by the outside probability $\Pr(S \Rightarrow^* w[0,i-1]Xw[j+1,N])$. (reachable)

Hyper Graphs and Chart Parsing

- There are ways to speed up the parser by throwing away less likely parts of the search space.
- We look at three techniques here:
 - Beam thresholding
 - Global thresholding
 - Coarse to fine parsing
- We compare the score of $X[i,j]$ with the score of the current highest scoring entry $Y[i,j]$ and throw away any rule starting with $X[i,j]$ if its score is not good enough. This technique is called **beam thresholding**.
- If we want to speed up the parser a little more then we eliminate rules that do not have neighboring rules to combine with it. This technique is called **global thresholding**.

Hyper Graphs and Chart Parsing

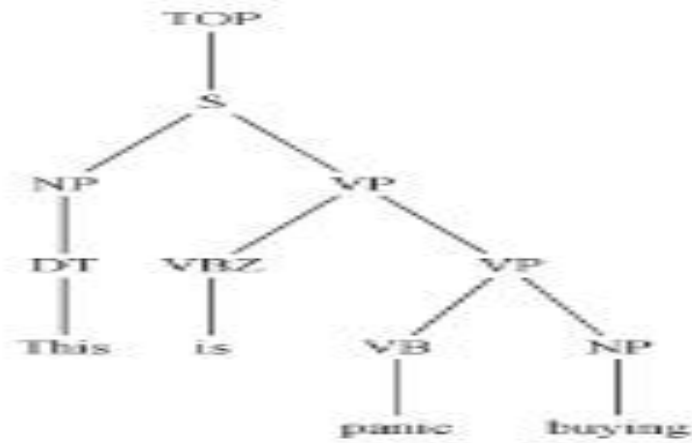


In the above figure nodes A, B and C will not be thresholded as each is part of a sequence from the beginning to the end of the chart. Nodes X, Y and Z will be thresholded as none of them is part of such a sequence. This is called global thresholding.

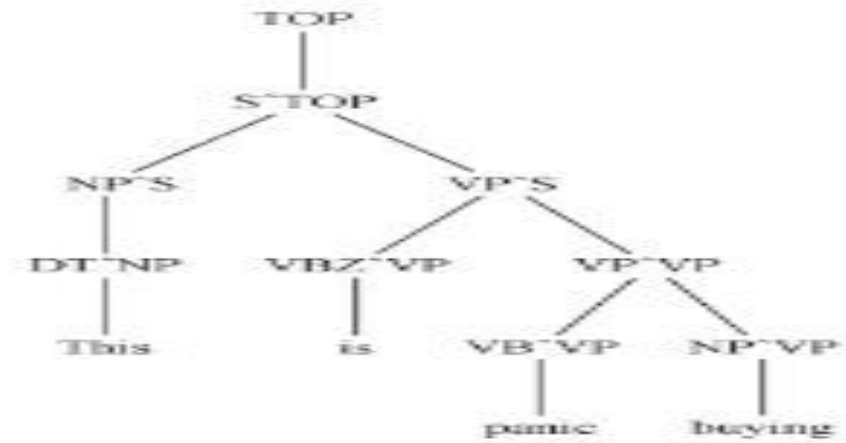
Hyper Graphs and Chart Parsing

- Let us assume we have a very complex set of Non-terminals.
- We parse first with a coarser non-terminal to prune the finer grained non-terminal. (using a VP non-terminal instead of VP^S)
- We now use the score associated with the coarse VP[i,j] to prune the finer grained non-terminals for the same span.
- This approach is called **coarse to fine parsing**.
- It is useful because the outside probability of the coarse step can be used in addition to the inside probability for more effective pruning.

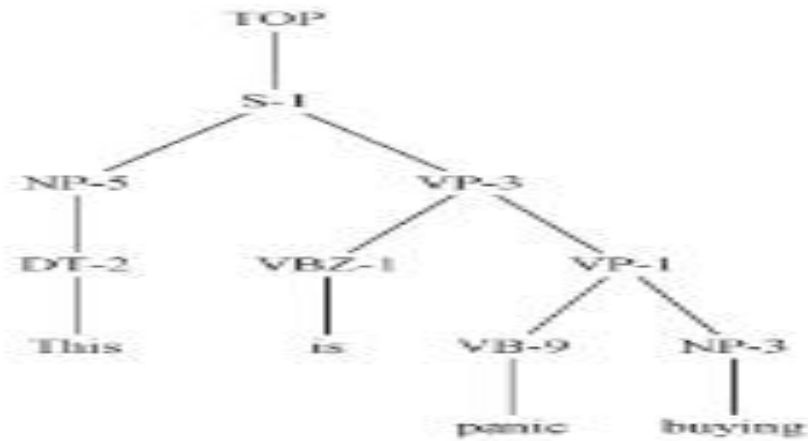
Hyper Graphs and Chart Parsing



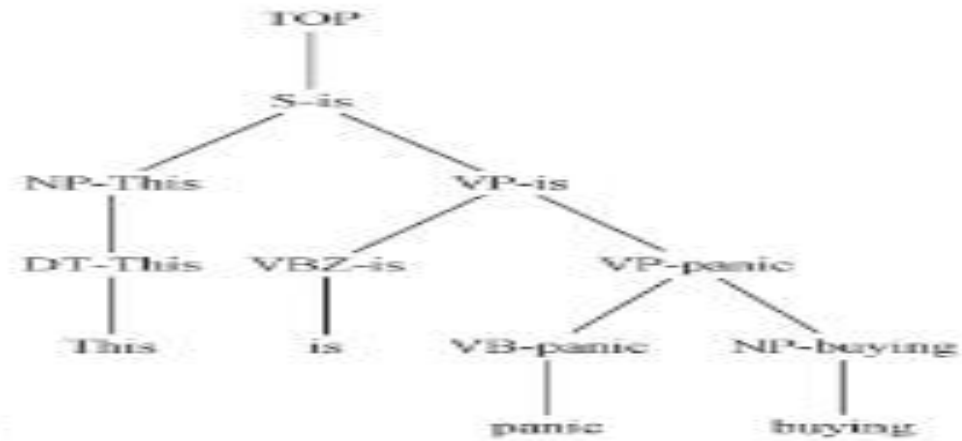
(a) A treebank tree



(b) Parent annotation of nonterminals



(c) Automatic split-merge of nonterminals into subcategories



(d) Lexicalization of the nonterminals

Hyper Graphs and Chart Parsing

- The parser can be sped up even further by using A* search rather than the exhaustive search used in the previous algorithm.
- A good choice of heuristic can be very helpful to provide faster times for parsing.
- The worst case complexity remains the same as CYK algorithm.
- In the case of projective dependency parsing CYK algorithm can be used by creating a CFG that produces dependency parsers.
- For dependency parsing the algorithm takes a worst-case of n^5 .

Hyper Graphs and Chart Parsing

- For dependency parsing instead of using non-terminals augmented with words we can represent sets of different dependency trees for each span of the input string.
- The idea is to collect left and right dependencies of each head independently and combine them at a later stage.
- Here we have the notion of **split head** where the head word is split into two:
 - One for the left dependents
 - One for the right dependents
- In addition to the head word for each span in each item we store the following:
 - Whether the head is gathering left or right dependencies.
 - Whether the item is complete. (cannot be extended with more dependents)

Hyper Graphs and Chart Parsing

- This provides an n^3 worst case dependency parsing algorithm.
- In the algorithm the spans are stored in a chart data-structure C e.g. $C[i,j]$ refers to the dependency analysis of span i,j .
- Incomplete spans are referred to as C^i , complete spans are C^c .
- The spans that are grown towards the left are referred to as $C_{<-}$, and the spans that are grown towards the right are referred to as $C_{->}$.
- For $C_{<-}[i][j]$ the head must be j and for $C_{->}[i][j]$ the head must be i .

Hyper Graphs and Chart Parsing

Initialize: for $s = 1..n$ chart $C_d^c[s][s] = 0.0$ for $d \in \{\leftarrow, \rightarrow\}$ and $c \in \{i, c\}$
for $k = 1 \dots n$ do
 for $s = 1 \dots n$ do
 $t = s + k$
 break if $t > n$
 first: create incomplete items
 $C_{\leftarrow}^i[s][t] = \max_{s \leq r < t} C_{\rightarrow}^c[s][r] + C_{\leftarrow}^c[r+1][t] + s(t, s)$
 $C_{\rightarrow}^i[s][t] = \max_{s \leq r < t} C_{\rightarrow}^c[s][r] + C_{\leftarrow}^c[r+1][t] + s(s, t)$
 second: create complete items
 $C_{\leftarrow}^c[s][t] = \max_{s \leq r < t} C_{\leftarrow}^c[s][r] + C_{\leftarrow}^i[r][t]$
 $C_{\rightarrow}^c[s][t] = \max_{s \leq r < t} C_{\rightarrow}^i[s][r] + C_{\rightarrow}^c[r][t]$
 end for
end for

Hyper Graphs and Chart Parsing

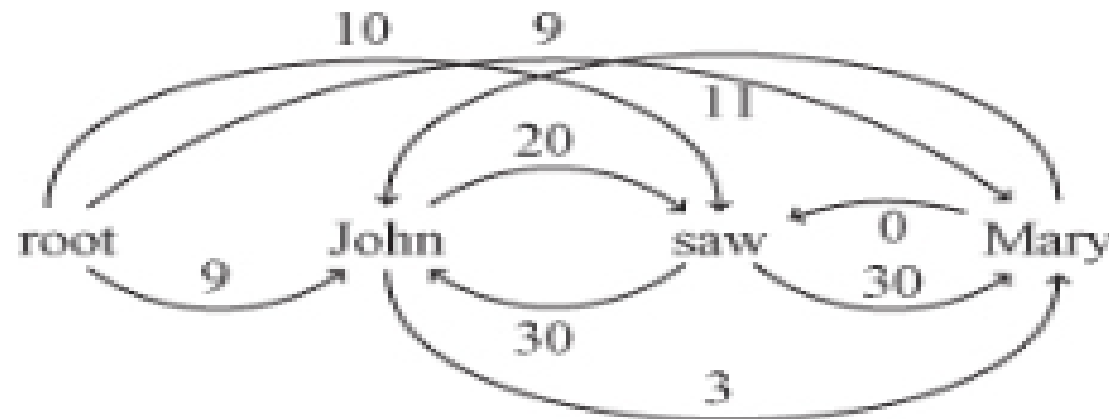
- We assume the unique root node as the leftmost token.
- The score of the best tree for the entire sentence is in $C_{\rightarrow}^c[1][n]$.
- This algorithm can be extended to provide k-best parses with a complexity of $O(n^3k \log k)$.

Minimum Spanning Trees and Dependency Parsing

- Finding the optimum branching in a directed graph is closely related to the problem of finding a minimum spanning tree in an undirected graph.
- A prerequisite is that each potential dependency link between words should have a score.
- In NLP minimum spanning tree (MST) is used to refer to the optimum branching problem in directed graphs.
- The scores we have can be used to find the MST, which is the highest scoring dependency tree.

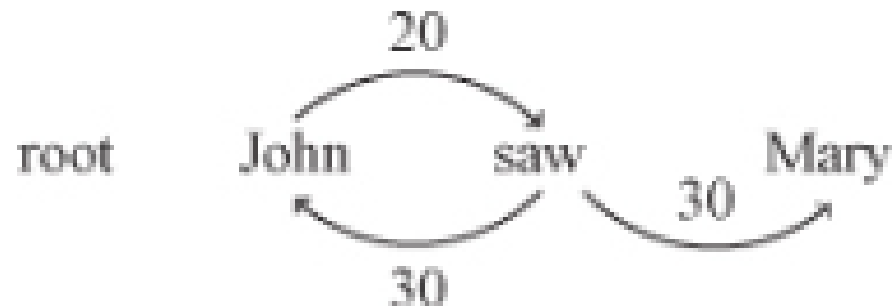
Minimum Spanning Trees and Dependency Parsing

- Nonprojective dependencies can be recovered because the linear order of the words in the input is not considered.
- Let us look at an example sentence “John saw Mary” with precomputed scores.



Minimum Spanning Trees and Dependency Parsing

- The first step is to find highest scoring incoming edges.
- If the step results in a tree we report this as the parse because it is the MST.
- In our example the highest scoring edges result in a cycle.



Minimum Spanning Trees and Dependency Parsing

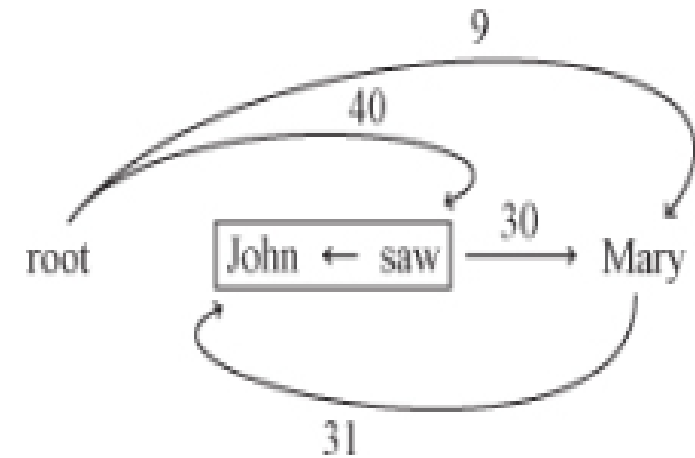
- We contract the cycle into a single node and recalculate the edge weights.
- The resultant graph is as follows:

$root \rightarrow \boxed{saw \rightarrow John} : wt = 40$

$root \rightarrow \boxed{John \rightarrow saw} : wt = 29$

$Mary \rightarrow \boxed{saw \rightarrow John} : wt = 30$

$Mary \rightarrow \boxed{John \rightarrow saw} : wt = 31$

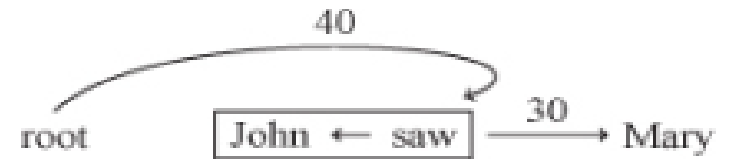


Minimum Spanning Trees and Dependency Parsing

- We now run the MST algorithm on the previous graph.
- The resultant graph is as follows:

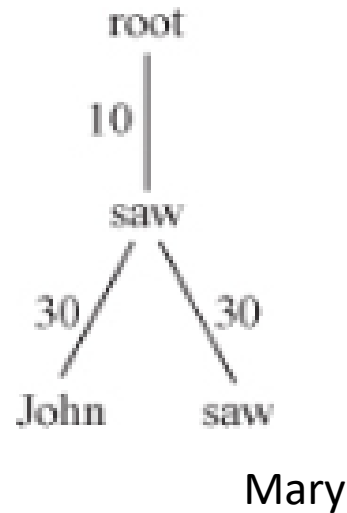
$root \rightarrow Mary \rightarrow \boxed{John-saw}$: wt = 9 + 31

$root \rightarrow \boxed{John-saw} \rightarrow Mary$: wt = 40 + 30



Minimum Spanning Trees and Dependency Parsing

- The MST that is the highest score dependency parse tree is as follows:



Models for Ambiguity Resolution in Parsing

- Here we discuss about the design features and ways to resolve ambiguity in parsing.
- The different issues discussed here are as follows:
 - Probabilistic Context-Free Grammars
 - Generative models for parsing
 - Discriminative Models for parsing

Probabilistic Context-Free Grammars

- Let us look at the example we already used to describe ambiguity. The sentence “John bought a shirt with pockets”

(S (NP John)

(VP (VP (V bought)

(NP (D a)

(N shirt))))

(PP (P with)

(NP pockets))))))

(S (NP John)

(VP (V bought)

(NP (NP (D a)

(N shirt))

(PP (P with)

(NP pockets))))))

Probabilistic Context-Free Grammars

- We want to provide a grammar so that the second tree is preferred over the first.
- The original grammar for the parse trees is:

$S \rightarrow NP VP$

$NP \rightarrow \text{'John'} \mid \text{'pockets'} \mid D N \mid NP PP$

$VP \rightarrow V NP \mid VP PP$

$V \rightarrow \text{'bought'}$

$D \rightarrow \text{'a'}$

$N \rightarrow \text{'shirt'}$

$PP \rightarrow P NP$

$P \rightarrow \text{'with'}$

Probabilistic Context-Free Grammars

- We can add scores or probabilities to the rules in this CFG in order to provide a score or probability for each derivation.
- The probability of a derivation is the sum of scores or product of probabilities of all the CFG rules used in the derivation.
- The scores are viewed as log probabilities and we use the term Probabilistic Context-Free Grammar (PCFG).
- We assign probabilities to the CFG rule such that for a rule $N \rightarrow \alpha$, the probability is $P(N \rightarrow \alpha / N)$
- Each rule probability is conditioned on the left hand side of the rule.
- The probability is distributed among all the expansions of the Non-terminals

$$1 = \sum_{\alpha} P(N \rightarrow \alpha)$$

Probabilistic Context-Free Grammars

$S \rightarrow NP VP (1.0)$

$NP \rightarrow \text{'John'} (0.1) \mid \text{'pockets'} (0.1) \mid D N (0.3) \mid NP PP (0.5)$

$VP \rightarrow V NP (0.9) \mid VP PP (0.1)$

$V \rightarrow \text{'bought'} (1.0)$

$D \rightarrow \text{'a'} (1.0)$

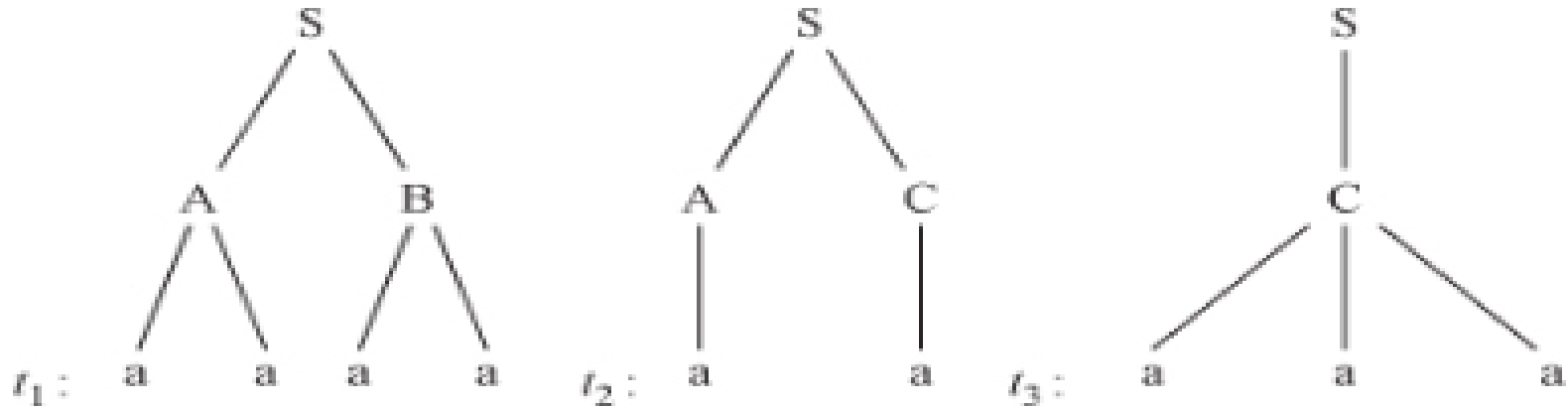
$N \rightarrow \text{'shirt'} (1.0)$

$PP \rightarrow P NP (1.0)$

$P \rightarrow \text{'with'} (1.0)$

Probabilistic Context-Free Grammars

- The rule probabilities can be derived from a tree bank.
- Consider a treebank with three trees t_1, t_2 and t_3 .



- Let t_1 occur 10 times in the tree bank t_2 20 times and t_3 50 times.

Probabilistic Context-Free Grammars

- The PCFG we obtain from the treebank would be:

$$\frac{10}{10+20+50} = 0.125 \quad S \rightarrow A B$$

$$\frac{20}{10+20+50} = 0.25 \quad S \rightarrow A C$$

$$\frac{50}{10+20+50} = 0.625 \quad S \rightarrow C$$

$$\frac{10}{10+20} = 0.334 \quad A \rightarrow a a$$

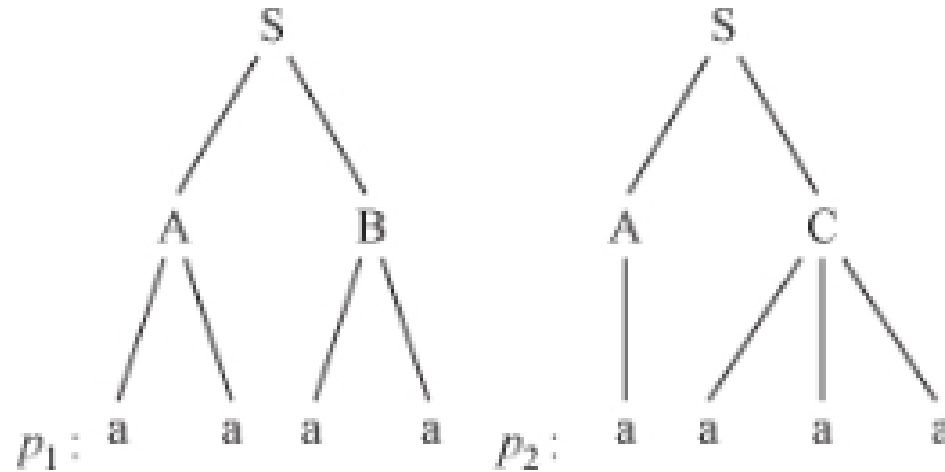
$$\frac{20}{10+20} = 0.667 \quad A \rightarrow a$$

$$\frac{20}{20+50} = 0.285 \quad B \rightarrow a a$$

$$\frac{50}{20+50} = 0.714 \quad C \rightarrow a a a$$

Probabilistic Context-Free Grammars

- For input aaaa there are two parses using the above PCFG

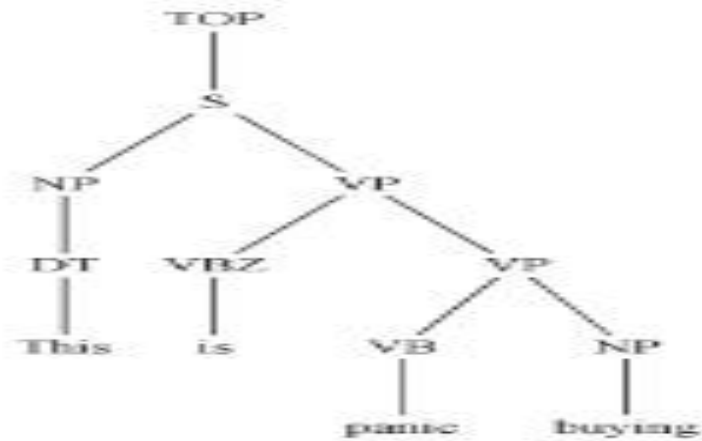


- The probability is: $p_1 = .125 * .334 * .285 = .01189$ $p_2 = .25 * .667 * .714 = .119$

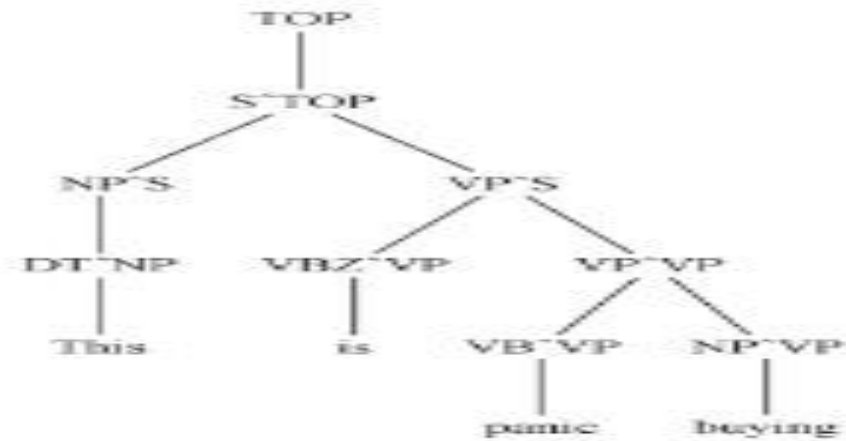
Probabilistic Context-Free Grammars

- The most likely parse tree does not even occur in the treebank.
- The reason is the context free nature of the PCFG where a non-terminal can be expanded by any rule with that non-terminal on the left hand side.
- To solve this problem the approach in statistical parsers is to augment the node labels in order to avoid bad independent assumptions.
- In the example already discussed we used three methods to augment the node labels:
 - Annotate each non-terminal with the label of its parent.
 - Learn the non-terminal splits by using an unsupervised learning algorithm
 - Lexicalize the non-terminals

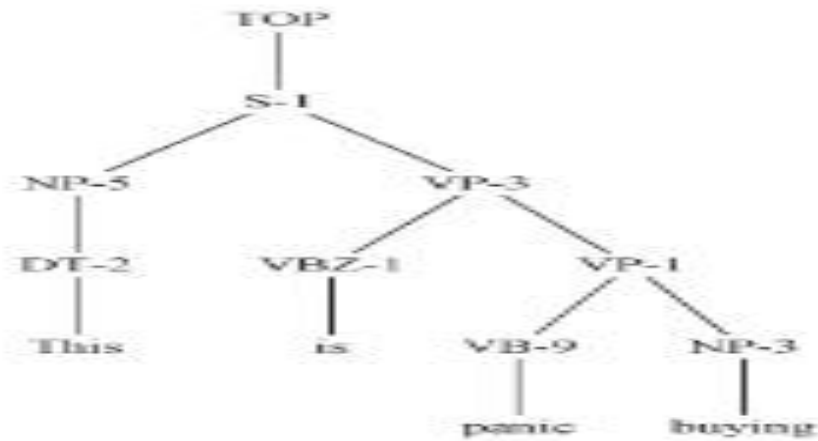
Probabilistic Context-Free Grammars



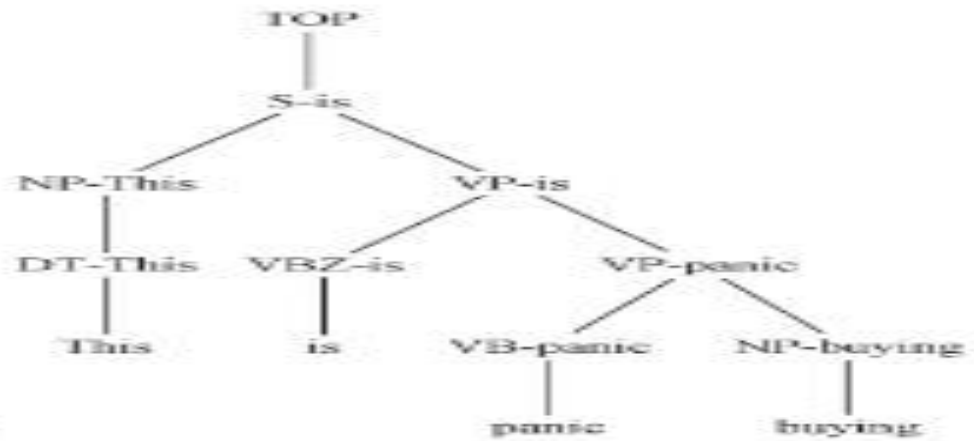
(a) A treebank tree



(b) Parent annotation of nonterminals



(c) Automatic split-merge of nonterminals into subcategories



(d) Lexicalization of the nonterminals

Probabilistic Context-Free Grammars

- For lexicalized nonterminal PCFG, the unfolding history is created head outward:
 - First the head is predicted, then the left sibling is produced and then the right sibling is produced.
- An alternative way is to find the best tree not with highest score but with the highest number of correct constituents.
- Appropriate changes like replacing the scoring function for each $X[i,j]$ to be the product of the inside and outside probability rather than just using the inside probability.
- This technique is called max-rule parsing and can produce trees that are not valid PCFG parse trees.

Generative Models for Parsing

- Let each derivation $D=d_1,d_2,\dots,d_n$ which is a sequence of decisions used to build the parse tree.
- For input sequence x , the output parse tree y is defined by the sequence of steps in the derivation.
- The probability for each derivation can be:

$$P(x, y) = P(d_1, \dots, d_n) = \prod_{i=1}^n P(d_i \mid d_1, \dots, d_{i-1})$$

- The conditioning context in the probability is called the history and corresponds to a partially built parse tree.

Generative Models for Parsing

- We group the histories into equivalence classes using a function ϕ .

$$P(d_1, \dots, d_n) = \prod_{i=1}^n P(d_i \mid \Phi(d_1, \dots, d_{i-1}))$$

- Using ϕ each history $H_i = d_1, d_2, \dots, d_{i-1}$ for all x, y is mapped to some fixed finite set of feature functions of the history $\phi_1(H_i), \dots, \phi_k(H_i)$
- In terms of these k feature functions:

$$P(d_1, \dots, d_n) = \prod_{i=1}^n P(d_i \mid \phi_1(H_i), \dots, \phi_k(H_i))$$

- The definition of PCFGs means that various other rule probabilities must be adjusted to obtain the right scoring parsers.

Generative Models for Parsing

- The independent assumptions in PCFG which are dictated by the underlying CFG lead to bad models.
- Such ambiguities can be modeled using arbitrary features of the parse tree.
- Discriminative methods provide such class of models.

Discriminative models for Parsing

- A framework that describes various discriminative approaches to learning for parsing is called a global linear model.
- Let \mathbf{x} be set of inputs and \mathbf{y} be set of possible outputs that can be a sequence of POS tags or a parse tree or a dependency analysis.
 - Each $x \in \mathbf{x}$ and $y \in \mathbf{y}$ is mapped to a d -dimensional feature vector $\phi(x,y)$, with each dimension being a real number, summarizing partial information contained in (x,y) .
 - A weight parameter vector $w \in \mathbb{R}^d$ assigns a weight to each feature in $\phi(x,y)$ representing the importance of that feature.
 - The value of $\phi(x,y) \cdot w$ is the $\text{score}(x,y)$.
 - The higher the score, the more plausible it is that y is the output for x .
 - The function $\text{GEN}(x)$ generates the set of possible outputs y for a given x .

Discriminative models for Parsing

- Having $\phi(x,y)$, w and $GEN(x)$ specified we would like to choose the highest scoring candidate y^* from $GEN(x)$ as the most plausible output. That is,

$$F(x) = \underset{y \in GEN(x)}{\operatorname{argmax}} p(y | x, \mathbf{w})$$

- Were $F(x)$ returns the highest scoring output y^* from $GEN(x)$.
- A conditional random field defines the conditional probability as a linear score for each candidate y and a global normalization term:

$$\log p(y | x, \mathbf{w}) = \Phi(x, y) \cdot \mathbf{w} - \log \sum_{y' \in GEN(x)} \exp(\Phi(x, y') \cdot \mathbf{w})$$

Discriminative models for Parsing

- A simpler global linear model that ignores the normalization term is :

$$F(x) = \operatorname{argmax}_{y \in GEN(x)} \Phi(x, y) \cdot \mathbf{w}$$

- Experimental results in parsing have shown that the simpler global linear model provides the same accuracy compared to the normalized models.
- A perceptron was originally introduced as a single-layered neural network
- During training the perceptron adjusts a weight parameter vector that can be applied to the input vector to get the corresponding output.
- The perceptron ensures that the current weight vector is able to correctly classify the current training examples.

Discriminative models for Parsing

Algorithm 3–1. The Original Perceptron Learning Algorithm

Inputs: Training Data $\langle (x_1, y_1), \dots, (x_m, y_m) \rangle$;

number of iterations T

Initialization: Set $w = \mathbf{0}$

Algorithm:

1: **for** $t = 1, \dots, T$ **do**

2: **for** $i = 1, \dots, m$ **do**

Discriminative models for Parsing

3:

Calculate y'_i , where $y'_i = \operatorname{argmax}_{y \in \text{GEN}(x)} \Phi(x_i, y) \cdot \mathbf{w}$

4: **if** $y'_i \neq y_i$ **then**

5: $\mathbf{w} = \mathbf{w} + \Phi(x_i, y_i) - \Phi(x_i, y'_i)$

6: **end if**

7: **end for**

8: **end for** **Output:** The updated weight parameter vector \mathbf{w}

Discriminative models for Parsing

- In the original perceptron learning algorithm:
 - The incremental weight updating suffers from overfitting
 - The algorithm is not capable of dealing with training data that is linearly inseparable
- A variant of the original perceptron learning algorithm is the voted perceptron algorithm.
- In the voted perceptron algorithm instead of storing and updating parameter values inside one weight vector, its learning process keeps track of all intermediate weight vectors and these intermediate vectors are used in the classification phase to vote for the answer.

Discriminative models for Parsing

- The voted perceptron keeps a count c_i to record the number of times a particular weight parameter vector (w_i, c_i) survives in the training.
- For a training example, if its selected top candidate is different from the truth, a new count c_{i+1} , being initialized to 1 is used.
- An updated weight vector (w_{i+1}, c_{i+1}) is produced and the original c_i and weight vector (w_i, c_i) are stored.

Discriminative models for Parsing

Algorithm 3–2. The Voted Perceptron Algorithm

Training Phase

Input: Training data $\langle (x_1, y_1), \dots, (x_m, y_m) \rangle$,
number of iterations T

Initialization: $k = 0$, $\mathbf{w}_0 = \mathbf{0}$, $c_1 = 0$

Algorithm:

for $t = 1, \dots, T$ **do**

for $i = 1, \dots, m$ **do**

Calculate y'_i , where $y'_i = \operatorname{argmax}_{y \in \text{GEN}(x)} \Phi(x_i, y) \cdot \mathbf{w}_k$

Discriminative models for Parsing

if $y'_i = y_i$ **then**

$$c_k = c_k + 1$$

else

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \Phi(x_i, y_i) - \Phi(x_i, y'_i)$$

$$c_{k+1} = 1$$

$$k = k + 1$$

end if

end for

end for

Output: A list of weight vectors $\langle (\mathbf{w}_1, c_1), \dots, (\mathbf{w}_k, c_k) \rangle$

Discriminative models for Parsing

Prediction Phase

Input: The list of weight vectors $\langle (\mathbf{w}_1, c_1), \dots, (\mathbf{w}_k, c_k) \rangle$, an unsegmented sentence x

Calculate:

$$y^* = \operatorname{argmax}_{y \in GEN(x)} \left(\sum_{i=1}^k c_i \Phi(x, y) \cdot \mathbf{w}_i \right)$$

Output: The voted top ranked candidate y^*

Discriminative models for Parsing

- The averaged perceptron algorithm is an approximation to the voted perceptron.
- It maintains the stability of the voted perceptron but reduces the space and time complexity.
- Instead of using w the averaged weight parameter vector γ over the m training examples is used for future predictions on unseen data.

$$\gamma = \frac{1}{mT} \sum_{i=1 \dots m, t=1 \dots T} w^{i,t}$$

Discriminative models for Parsing

- In calculating γ an accumulating parameter σ is used and updated using w for each example.
- After the last iteration σ/mT produces the final parameter vector γ .

Algorithm 3–3. The Averaged Perceptron Learning Algorithm

Inputs: Training Data $\langle (x_1, y_1), \dots, (x_m, y_m) \rangle$;

number of iterations T

Initialization: Set $w = \mathbf{0}$, $\gamma = \mathbf{0}$, $\sigma = 0$

Algorithm:

for $t = 1, \dots, T$ **do**

for $i = 1, \dots, m$ **do**

Discriminative models for Parsing

Calculate y'_i , where $y'_i = \operatorname{argmax}_{y \in GEN(x)} \Phi(x_i, y) \cdot \mathbf{w}$

if $y'_i \neq y_i$ **then**

$$\mathbf{w} = \mathbf{w} + \Phi(x_i, y_i) - \Phi(x_i, y'_i)$$

end if

$$\sigma = \sigma + \mathbf{w}$$

end for

end for

Output: The averaged weight parameter vector $\gamma = \sigma / (mT)$

Multilingual Issues: What is a Token

- Tokenization, Case and Encoding
- Word Segmentation
- Morphology

Tokenization, Case and Encoding

- The definition of a word token is well defined given a treebank or parser but variable across different treebanks or parsers.
- For example the possessive marker(Rama's) and copula verbs(There's) are treated differently in different treebanks.
- In some languages there are issues with upper case and lower case.
- For example the word Boeing might seem like singing if the first letter is not an upper case.
- Low count tokens can be replaced with patterns that retain case information. Example Patagonia can be replaced by Xxx.

Tokenization, Case and Encoding

- For languages that are not encoded in ASCII, the different encodings need to be managed.
- There are also issues with the sentence terminator(.), which in some corpora will be in ASCII but in others it will be in UTF-8(Unicode Transformation Format).
- Some languages like Chinese are encoded in different formats depending on the place where the text originates.

Word Segmentation

- The written form of many languages like Chinese lack marks identifying words.
- Word segmentation is the process of demarcating blocks in a character sequence such that the produced output is composed of separate tokens and is meaningful.
- Only if we are able to identify each word in the sentence POS tag can be assigned and syntax analysis can be done.
- Chinese word segmentation has a large community of researchers and has resulted in three different types.

Word Segmentation

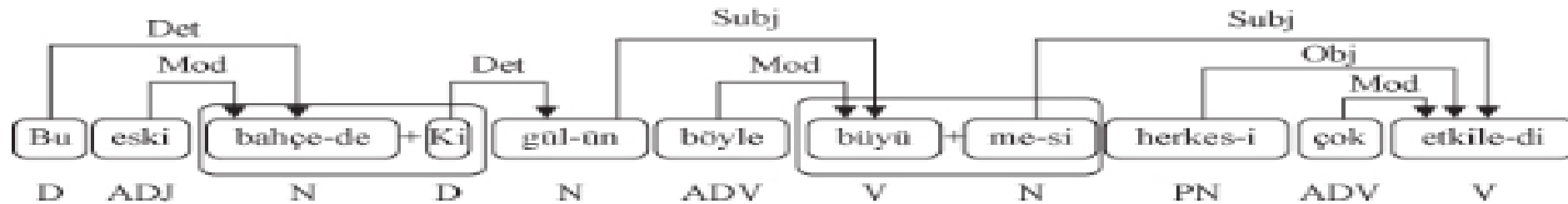
- One approach to Chinese parsing is to parse the character sequence directly.
- The non-terminals in the tree that span a group of characters can be said to specify the word boundaries.
- The best word segmentation model creates a pipeline where the parser is unable to choose between different plausible segmentations.
- The parser for CFGs can parse an input word lattice.(represented by a finite state automata)

Morphology

- In many languages the notion of splitting up tokens using spaces is problematic.
- Each word can contain several components called morphemes such that the meaning of the word can be thought of as the combination of the meanings of the morphemes.
- A word is a representation of a stem combined with several morphemes.

Morphology

- An example from Turkish treebank shows that the syntactic dependencies need to be aware of the morphemes within words.



- In the above example morpheme boundaries within a word are shown using the + symbol.
- Morphemes and not words are used as heads and dependents.
- Agglutinative languages like Turkish have the property of entire clauses being combined with morphemes to create complex words.

Morphology

- In languages like Czech and Russian different morphemes are used to mark grammatical case, gender and so on.
- In such languages each inflected word can be ambiguously segmented into morphemes with different analyses.
- Now the parser has to deal with morphological ambiguity also.
- The disambiguation problem of morphology can be reduced to a POS tagging task.
-

Morphology

- For example a POS of V--M-3---- can indicate:
 - Each word can have morphemes that inflect the word along 10 different dimensions.
 - In this case the stem is a verb with masculine gender and in third person.
- The POS tagger has to produce this complex tag which can be done by proper training.
- In such languages each word is tagged with a POS tag that encodes a lot of information about the morphemes.
- This tag set can be used as features for a statistical parser for a highly inflected language.

Morphology

- In discriminative models for statistical parsing we can include the morphological information because discriminative models allow the inclusion of a large number of overlapping features.
- The morphological information associated with the words can be used to build better statistical parsers by simply throwing them into the mix.