Syntax Parsing

1 Syntax Parsing

Syntax parsing is the process of analyzing the grammatical structure of a sentence in order to understand how words are organized and related to each other. The main motivation behind parsing is that every sentence contains a hidden structural description. For example, in the sentence *"He wanted to go for drive in monsoon"*, the meaning becomes clearer when we understand how phrases such as *"to go"* and *"for drive"* are grouped. Parsing helps uncover this internal structure so that machines, as well as linguists, can interpret the sentence correctly.

Another important purpose of parsing is to understand how the position of words affects meaning. Simply identifying parts of speech (POS tagging) is not always sufficient; it is also necessary to determine how words function within the sentence. Consider the sentence: *"The cat who lives dangerously had nine lives."* In this example, the word *"lives"* appears twice. In the first occurrence, it functions as a verb (meaning "exists"), while in the second occurrence, it functions as a noun (meaning "number of lives"). Parsing distinguishes such differences by analyzing syntactic structure rather than relying only on word form.

In applications such as text summarization, parsing plays a key role in identifying important phrases and constituents. When summarizing a long document, several similar ideas are compressed into a few lines. To achieve this effectively, the system must understand sentence structure so that it can extract the main clauses and preserve essential information. Identifying paraphrases—where a few words are changed without altering the meaning—also depends on syntactic understanding.

Treebanks represent a data-driven approach to syntax. A treebank is a collection of sentences annotated with their syntactic structures. These annotated examples are used to train parsing models. Instead of manually writing all grammar rules, we can automatically learn structural patterns from real linguistic data.

Finally, Context-Free Grammar (CFG) provides the fundamental framework for syntactic analysis. To analyze a sentence, grammatical rules must be defined to explain how the output structure is generated. The grammar specifies how words combine to form phrases and how phrases combine to form complete sentences. In this way, grammar provides insight into how sentence structure is systematically formed and interpreted.

2 A Simple Context-Free Grammar (CFG) for English

A simple Context-Free Grammar (CFG) for English defines a set of production rules that describe how sentences are formed. Typically, a sentence (S) is rewritten as a noun phrase (NP) followed by a verb phrase (VP):

$$S \rightarrow NP\ VP$$

The noun phrase may consist of a proper noun such as *"John"*, a noun such as *"pocket"*, or a determiner followed by a noun ($D\ N$), and it may also include a prepositional phrase (PP). The verb phrase can be a verb followed by a noun phrase ($V\ NP$) or a verb phrase followed by a prepositional phrase ($VP\ PP$).

Additional lexical rules define terminal productions such as:

$$V \rightarrow \text{“bought”}, \quad D \rightarrow \text{“a”}, \quad N \rightarrow \text{“shirt”}, \quad P \rightarrow \text{“with”}.$$

In Natural Language Processing (NLP), words are treated as **terminals**, while syntactic categories such as NP, VP, and PP are **non-terminals**. Using these CFG rules, we can derive a sentence such as *"John bought a shirt with pockets"* by repeatedly applying production rules starting from the start symbol S.

3 Ambiguity and Multiple Derivations

CFGs often lead to structural ambiguity, meaning that a sentence may have more than one valid parse tree. For example, in the sentence *"John bought a shirt with pockets"*, the prepositional phrase *"with pockets"* can attach either to the noun *"shirt"* (describing the shirt) or to the verb phrase (describing the act of buying). These two attachments produce different syntactic structures and interpretations.

Due to recursive grammar rules, the number of possible parse trees can grow rapidly. In fact, the number of possible binary tree structures for n elements is related to the Catalan numbers:

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

which grow exponentially. Therefore, enumerating all possible parses is computationally expensive. Efficient parsing algorithms aim to compute the correct parse in polynomial time, typically $O(n^3)$, rather than exploring all possible structures.

4 Treebank: A Data-Driven Approach to Syntax

A treebank is a collection of sentences annotated with their syntactic structure. It represents a data-driven approach to syntax analysis and addresses the knowledge acquisition problem. Each sentence in a treebank is manually analyzed and reviewed by human experts to ensure correctness. Annotation guidelines are carefully designed to maintain consistency across annotations.

To measure annotation consistency, a portion of the data (for example, 10%) is annotated by multiple annotators. Their annotations are compared to compute **inter-annotator agreement**. Treebanks do not explicitly provide formal grammar rules; instead, the grammar is implicit in the annotated structures. From these examples, grammar rules can be automatically learned.

5 Why Treebanks Are Useful

Treebanks support two major tasks: learning grammar and training parsers. Instead of manually specifying grammar rules, machine learning models can infer structural patterns from annotated sentences. These learned patterns can then be applied to analyze new, unseen sentences. The typical workflow involves using a treebank to train a syntactic parser, which then performs syntax analysis on new input.

6 Methods to Construct a Treebank

There are two primary methods for constructing a treebank:

1. **Dependency Graphs:** These focus on word-to-word relationships and are common in languages such as Czech and Turkish.
2. **Phrase Structure Trees:** These represent hierarchical phrase-based structures and are commonly used for English. They are particularly useful for modeling long-distance dependencies and nested constructions.

7 Components of Building Parsers

Building a parser involves three major components:

- **Representation:** The formal structure used to model syntax (e.g., phrase structure trees or dependency trees).
- **Training:** Learning parameters or grammar rules from annotated data.
- **Decoding:** Selecting the most appropriate parse for a given sentence from all possible structures.

8 Dependency Syntax Structure

Dependency syntax represents sentence structure using direct word-to-word relationships. In this framework, each word (except the root) depends on another word called its *head*. The structure forms a tree with a single root, typically the main verb of the sentence. Each word has exactly one head, producing a one-to-one mapping between words and their heads.

Graphically, words are represented as nodes, and dependencies are directed edges from the head to the dependent. These edges are labeled with grammatical roles such as subject (SBJ), object (OBJ), or modifier (MOD). This representation emphasizes functional relationships and is computationally efficient for syntactic analysis in NLP systems.

9 Projective and Non-Projective Dependency Trees

In dependency syntax, a **projective dependency tree** is one in which the dependency arcs do not cross when drawn above the sentence. Formally, if a word w_i depends on a head w_j, then all the words occurring between w_i and w_j in the linear order must also be dominated by w_j. This ensures that each subtree forms a continuous span in the sentence. Projectivity is common in relatively fixed word-order languages such as English.

A **non-projective dependency tree**, on the other hand, contains crossing arcs. This occurs when a dependent is separated from its head by words that are not dominated by that head. Non-projectivity is frequent in free word-order languages such as Czech, Hungarian, and Turkish.

10 Example of a Projective Dependency Tree

Sentence: *I prefer the morning flight through Denver.*

Dependency Table

Index	Word	POS	Head	Relation
1	I	PRP	2	nsubj
2	prefer	VBP	0	root
3	the	DT	5	det
4	morning	NN	5	nmod
5	flight	NN	2	dobj
6	through	IN	7	case
7	Denver	NNP	5	nmod

Table 1: Projective Dependency Representation

Graph Representation

```
        prefer (ROOT)
        /      \
    I(nsubj)   flight(dobj)
                /      |      \
         the(det) morning(nmod) Denver(nmod)
                                   |
                              through(case)
```

No arcs cross in this structure; therefore, it is projective.

11 Example of a Non-Projective Dependency Tree

Sentence: *Chris saw a dog yesterday which I liked.*

Dependency Table

Index	Word	POS	Head	Relation
1	Chris	NNP	2	nsubj
2	saw	VBD	0	root
3	a	DT	4	det
4	dog	NN	2	dobj
5	yesterday	NN	2	nmod
6	which	WDT	8	dobj
7	I	PRP	8	nsubj
8	liked	VBD	4	relcl

Table 2: Non-Projective Dependency Representation

Graph Representation

```
        saw (ROOT)
       /    |      \
 Chris   dog(dobj)  yesterday(nmod)
           |   \
         a(det) liked(relcl)
                  /   \
               I      which
```

When drawn linearly, the arc between *dog* and *liked* crosses other arcs, making this structure non-projective.

12 Common Dependency Labels

- **nsubj** – Nominal subject (doer of the action)
- **dobj** – Direct object (receiver of the action)
- **det** – Determiner (a, an, the)
- **nmod** – Noun modifier
- **case** – Prepositional marker
- **relcl** – Relative clause modifier

13 CoNLL-2007 Shared Task Dataset

Two important statistics related to non-projectivity are:

- %dep – Percentage of crossing dependency arcs
- %sent – Percentage of sentences containing at least one non-projective dependency

Code	Language	%dep	%sent
Ar	Arabic	0.4%	10.1%
Ba	Basque	2.9%	26.2%
Ca	Catalan	0.1%	2.9%
Ch	Chinese	0.0%	0.0%
Cz	Czech	1.9%	23.2%
Bn	Bulgarian	0.3%	6.7%
Gr	Greek	1.1%	20.3%
Hu	Hungarian	2.9%	26.4%
It	Italian	0.5%	7.4%
Tr	Turkish	5.5%	33.7%

Table 3: Percentage of Non-Projectivity in CoNLL-2007 Dataset

Non-Projectivity Statistics

14 Observations

Languages with relatively fixed word order (e.g., Chinese, Catalan) show very low non-projectivity. Free word-order and morphologically rich languages (e.g., Turkish, Hungarian, Basque) show higher levels of crossing dependencies.

15 Challenges of Non-Projectivity

1. Harder to parse computationally.
2. Cannot be fully represented by standard Context-Free Grammar (CFG).
3. Complicates grammar learning.
4. Often leads to lower parsing accuracy in practice.

16 Phrase Structure Grammar and Syntactic Representation

Phrase Structure Grammar explains how sentences are built hierarchically from smaller units called constituents. A sentence is not merely a linear sequence of words; rather, it has an internal structure where words combine into phrases, and phrases combine into larger phrases until a complete sentence is formed. This hierarchical organization allows us to understand grammatical relations and meaning.

In its simplest form, a sentence can be represented as:

$$S \rightarrow NP\ VP$$

This rule states that a sentence (S) typically consists of a noun phrase (NP) followed by a verb phrase (VP). However, each of these phrases can contain smaller phrases and words arranged in a tree-like structure.

16.1 Constituents

A constituent is a group of words that functions as a single unit within a sentence. Constituents may be single words or multi-word phrases. For example:

the morning flight

This string behaves as a single noun phrase (NP). Even though it contains multiple words, together they refer to one entity. The determiner "the", the modifier "morning", and the noun "flight" combine hierarchically to form an NP.

16.2 Example 1: Mr. Baker seems especially sensitive

Consider the sentence:

Mr. Baker seems especially sensitive.

The phrase structure representation is:

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

The entire structure is labeled S, indicating a sentence. The NP-SBJ represents the subject noun phrase consisting of the proper nouns “Mr.” and “Baker”. The VP contains the verb “seems” and a predicate adjective phrase (ADJP-PRD). Within the adjective phrase, “especially” functions as an adverb modifying the adjective “sensitive”.

The corresponding predicate–argument structure is:

$$seem(Mr.\ Baker,\ especially(sensitive))$$

This representation makes explicit that the verb “seem” takes two arguments: the subject (Mr. Baker) and the property attributed to him (especially sensitive).

16.3 Example 2: Chris threw the ball

Now consider:

Chris threw the ball.

Phrase structure representation:

```
(S
   (NP-SBJ (NNP Chris))
   (VP (VBD threw)
       (NP (DT the) (NN ball))))
```

The NP-SBJ contains the proper noun “Chris”. The VP consists of the past tense verb “threw” and an object noun phrase “the ball”. The determiner “the” and noun “ball” combine to form the object NP.

Predicate–argument structure:

$$throw(Chris,\ the\ ball)$$

This shows that the verb “throw” selects two arguments: the subject and the object.

16.4 Example 3: Passive Construction

Consider the passive sentence:

The ball was thrown by Chris.

Phrase structure representation (simplified):

```
(S
   (NP-SBJ-1 (DT The) (NN ball))
   (VP (VBD was)
       (VP (VBN thrown)
           (NP *-1)
           (PP (IN by)
               (NP (NNP Chris)))))
```

Here, “The ball” appears as the surface subject but originates as the object of “thrown”. The trace (*-1) marks the original object position. The prepositional phrase “by Chris” introduces the logical agent.

The underlying predicate–argument structure remains:

$$throw(Chris,\ the\ ball)$$

Thus, although the syntactic form changes (active to passive), the semantic relation remains the same.

16.5 Example 4: Control Structure

Consider:

They persuaded Mr. Trotter to take it back.

Phrase structure representation:

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

The main clause consists of the subject “They”, the verb “persuaded”, and the object “Mr. Trotter”. The embedded infinitival clause “to take it back” contains a null subject represented as (-NONE- *). This empty element refers back to “Mr. Trotter”.

Predicate–argument structure:

$$persuade(They,\ Mr.\ Trotter,\ take\_back(Mr.\ Trotter,\ it))$$

This makes explicit that Mr. Trotter is understood as the subject of the embedded verb “take”.

16.6 Example 5: WH-Movement

Consider the question:

What is Tim eating?

Phrase structure representation:

```
(SBARQ
   (WHNP-1 (WP What))
   (SQ
      (VBZ is)
      (NP-SBJ (NNP Tim))
      (VP (VBG eating)
          (NP *T*-1))))
```

The WH-word “What” has moved to the front of the sentence. The trace (*T*-1) marks its original position as the object of “eating”.

Predicate–argument structure:

$$eat(Tim,\ what)$$

This example illustrates movement and trace theory in syntactic analysis.

16.7 Example 6: Long-Distance Dependency

Consider:

Who was believed to have been shot?

Simplified structure:

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

The WH-word "Who" originates as the object of "shot" inside a deeply embedded clause. It moves to the sentence-initial position, leaving traces. This is called a long-distance dependency because the moved element and its original position are separated by multiple clause boundaries.

Predicate–argument structure:

$$\textit{believe}(\textit{someone},\ \textit{shot}(\textit{someone},\ \textit{who}))$$

This final example demonstrates how phrase structure, movement, traces, and predicate–argument structure together reveal the underlying logical interpretation of complex sentences.

17 chart parser

a pilot likes flying planes

Grammar with Scores

$$\begin{aligned}
S &\rightarrow NP\ VP \quad [2.0]\\
VP &\rightarrow VBG\ NNS \quad [1.6]\\
VP &\rightarrow VBZ\ VP \quad [1.7]\\
VP &\rightarrow VBZ\ NP \quad [1.8]\\
NP &\rightarrow DT\ NN \quad [1.5]\\
NP &\rightarrow JJ\ NNS \quad [1.4]\\
DT &\rightarrow a \quad [0.5]\\
NN &\rightarrow pilot \quad [0.6]\\
VBZ &\rightarrow likes \quad [0.7]\\
VBG &\rightarrow flying \quad [0.6]\\
JJ &\rightarrow flying \quad [0.5]\\
NNS &\rightarrow planes \quad [0.6]
\end{aligned}$$

a	pilot	likes	flying	planes
DT 0.5	NP 2.6			S 9.8
	NN 0.6			
		VBZ 0.7		VP 5.2
			VBG 0.6 JJ 0.5	VP 2.8 NP 2.5
				NNS 0.6

Span 1

$$x_{11} = DT = 0.5$$

$$x_{22} = NN = 0.6$$

$$x_{33} = VBZ = 0.7$$

$$x_{44} = VBG = 0.6, \quad JJ = 0.5$$

$$x_{55} = NNS = 0.6$$

Span 2

$x_{12} \Rightarrow x_{11}\ x_{22}$

$$DT\ NN \Rightarrow NP$$

$$0.5 + 0.6 + 1.5 = 2.6$$

$$x_{12} = NP\ (2.6)$$

$x_{23} \Rightarrow x_{22}\ x_{33}$

NN VBZ
Invalid branch

$x_{34} \Rightarrow x_{33}\ x_{44}$

VBZ VBG
Invalid branch

$x_{45} \Rightarrow x_{44}\ x_{55}$

$$VBG\ NNS \Rightarrow VP$$

$$0.6 + 0.6 + 1.6 = 2.8$$

$$JJ\ NNS \Rightarrow NP$$

$$0.5 + 0.6 + 1.4 = 2.5$$

$$x_{45} = VP\ (2.8)$$

Span 3

x_{13}

$$x_{13} \Rightarrow x_{11}\ x_{23}$$

Invalid branch

$$x_{13} \Rightarrow x_{12}\ x_{33}$$

Invalid branch

x_{24}

$$x_{24} \Rightarrow x_{22}\ x_{34}$$

Invalid branch

$$x_{24} \Rightarrow x_{23}\ x_{44}$$

Invalid branch

x_{35}

$$x_{35} \Rightarrow x_{33}\ x_{45}$$

$$VBZ\ VP \Rightarrow VP$$

$$0.7 + 2.8 + 1.7 = 5.2$$

$$x_{35} = VP\ (5.2)$$

Span 4

x_{14}

$$x_{14} \Rightarrow x_{11}\ x_{24}$$

Invalid branch

$$x_{14} \Rightarrow x_{12}\ x_{34}$$

Invalid branch

$$x_{14} \Rightarrow x_{13}\ x_{44}$$

Invalid branch

x_{25}

$$x_{25} \Rightarrow x_{22}\ x_{35}$$

Invalid branch

$$x_{25} \Rightarrow x_{23}\ x_{45}$$

Invalid branch

$$x_{25} \Rightarrow x_{24}\ x_{55}$$

Invalid branch

Span 5

x_{15}

$$x_{15} \Rightarrow x_{11}\ x_{25}$$

Invalid branch

$$x_{15} \Rightarrow x_{12}\ x_{35}$$

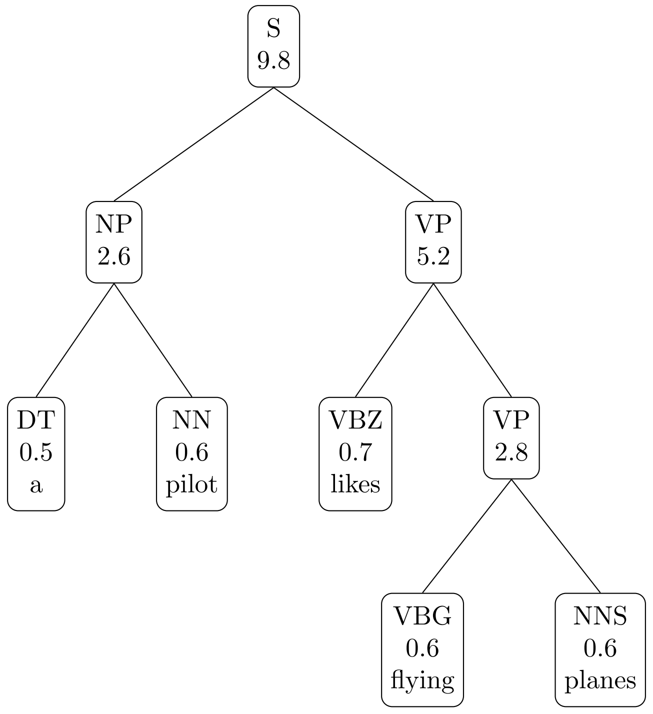
$$NP\ VP \Rightarrow S$$

$$2.6 + 5.2 + 2.0 = 9.8$$

$$x_{15} = S\ (9.8)$$

Final Score = 9.8

Final Parse Tree



18 Transition-Based Dependency Parsing

18.1 Basic Idea

Transition-based parsing derives a single syntactic representation (a dependency graph) through a deterministic sequence of elementary parsing actions. Instead of constructing the full tree at once, the parser incrementally builds the dependency structure by manipulating three components: a stack, a buffer, and a set of dependency arcs.

A parser configuration is defined as a triple:

$$c = (S, B, A)$$

where:

- S is a stack containing partially processed words,
- B is a buffer containing remaining input words,
- A is a set of labeled dependency arcs (w_i, d, w_j).

Initially:

$$([], [w_1, \ldots, w_n], \emptyset)$$

Termination condition:

$$(S, [], A)$$

This means parsing finishes when the buffer is empty and all dependencies are constructed.

18.2 Transition System

A transition system is defined as:

$$S = (C, T, c_s, C_t)$$

where:

- C is the set of configurations,
- T is the set of transitions,
- c_s is initialization,
- C_t is the set of terminal configurations.

Each transition transforms one configuration into another.

18.3 Arc-Eager Transitions

Arc-Eager parsing defines four transitions:

1. SHIFT

Precondition: Buffer is not empty.
Operation: Removes the first word from the buffer and pushes it onto the stack.
Meaning: We delay making a dependency decision and simply move the word for future attachment.

2. LEFT-ARC(d)

Precondition: Stack and buffer not empty; stack top has no head.
Operation: Adds arc:

$$(w_j \rightarrow w_i)$$

where w_j is first word in buffer (head) and w_i is top of stack (dependent). Then pop stack.
Meaning: The buffer word is the head of the stack word.

3. RIGHT-ARC(d)

Precondition: Stack and buffer not empty.
Operation: Adds arc:

$$(w_i \rightarrow w_j)$$

where stack top is head and buffer word is dependent. Then shift buffer word to stack.
Meaning: The stack word governs the buffer word.

4. REDUCE

Precondition: Stack top already has a head.
Operation: Pops stack top.
Meaning: Word is complete; all its dependencies have been assigned.

18.4 Parsing Example

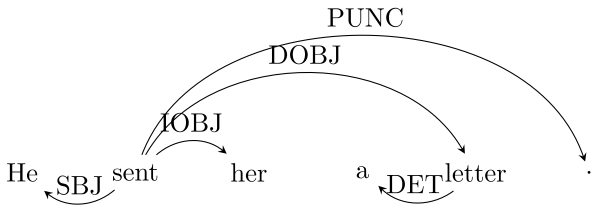
Sentence:

He sent her a letter .

Final dependency relations:

- sent → He (SBJ)
- sent → her (IOBJ)
- letter → a (DET)
- sent → letter (DOBJ)
- sent → . (PUNC)

18.5 Dependency Graph



18.6 Step-by-Step Transitions

Initial configuration:

$$([], [He, sent, her, a, letter, .], \emptyset)$$

Step 1: SHIFT Move "He" to stack.

$$([He], [sent, her, a, letter, .], \emptyset)$$

We shift because no dependency decision can yet be made.

Step 2: LEFT-ARC(SBJ) Add arc:

$$sent \rightarrow He$$

$$([], [sent, her, a, letter, .], \{sent \rightarrow He\})$$

Meaning: "sent" is head of "He".

Step 3: SHIFT

$$([sent], [her, a, letter, .], A)$$

Step 4: RIGHT-ARC(IOBJ)

$$sent \rightarrow her$$

$$([sent, her], [a, letter, .], A)$$

Meaning: "her" is indirect object of "sent".

Step 5: SHIFT

$$([sent, her, a], [letter, .], A)$$

Step 6: LEFT-ARC(DET)

$$letter \to a$$

$$([sent, her], [letter, .], A)$$

Meaning: ”a” is determiner of ”letter”.

Step 7: REDUCE Remove ”her” (already has head).

$$([sent], [letter, .], A)$$

Step 8: RIGHT-ARC(DOBJ)

$$sent \to letter$$

$$([sent, letter], [.], A)$$

Meaning: ”letter” is direct object of ”sent”.

Step 9: REDUCE

$$([sent], [.], A)$$

Step 10: RIGHT-ARC(PUNC)

$$sent \to .$$

$$([sent, .], [], A)$$

Step 11: REDUCE Final configuration:

$$([sent], [], A)$$

Parsing complete.

18.7 Understanding Arc Transitions Semantically

- **SBJ** arc: marks syntactic subject.
- **IOBJ** arc: indirect object (recipient).
- **DET** arc: determiner modifying noun.
- **DOBJ** arc: direct object.
- **PUNC** arc: punctuation attachment.

Each arc represents a head-dependent grammatical relation.

18.8 Conclusion

Transition-based parsing incrementally constructs dependency graphs through deterministic actions. The stack stores processed words, the buffer stores remaining words, and arcs accumulate syntactic relations. Arc-eager parsing builds dependencies as early as possible, enabling efficient linear-time parsing while maintaining structural correctness.

19 MST-Based Dependency Parsing

19.1 Basic Idea

Maximum Spanning Tree (MST) based dependency parsing views parsing as a global graph optimization problem. Instead of incrementally building a tree using transitions, we construct a fully connected directed graph over all words in the sentence and then select the highest scoring directed spanning tree.

The core idea is:

Starting from all possible connections, find the maximum spanning tree.

This ensures that the final dependency structure maximizes the total score over all chosen dependency arcs.

19.2 Graph Theory Foundations

A graph is defined as:

$$G = (V, A)$$

where:

- V is the set of vertices,
- A is the set of arcs (edges).

In dependency parsing, we use **directed graphs (digraphs)**. An arc (i, j) means there is a directed edge from vertex i to vertex j.

A **multi-digraph** allows multiple arcs between two vertices:

$$(i, j, k) \in A$$

indicates the k^{th} arc from i to j.

19.3 Directed Spanning Tree

A directed spanning tree of a graph $G = (V, A)$ is a subgraph:

$$G' = (V, A')$$

such that:

- $V' = V$
- $A' \subseteq A$
- $|A'| = |V| - 1$
- G' is acyclic

Thus, a spanning tree connects all vertices with exactly $|V| - 1$ arcs and contains no cycles.

19.4 Weighted Directed Graph

Each arc has a weight:

$$w_{ijk} \geq 0$$

The weight of a spanning tree G' is:

$$w(G') = \sum_{(i,j,k) \in G'} w_{ijk}$$

19.5 Maximum Spanning Tree Problem

Let $T(G)$ be the set of all spanning trees of G.
The MST is:

$$G^* = \arg\max_{G' \in T(G)} w(G')$$

This means we choose the spanning tree with maximum total weight.

19.6 Building the Graph for a Sentence

For a sentence $x = x_1, \ldots, x_n$, define:

$$V_x = \{x_0 = root, x_1, \ldots, x_n\}$$

$$E_x = \{(i, j) \mid i \neq j,\ i \in [0, n],\ j \in [1, n]\}$$

This means:

- Every word can potentially be head of every other word.
- Root connects to all words.

19.7 Example: John saw Mary

We build a fully connected directed graph.

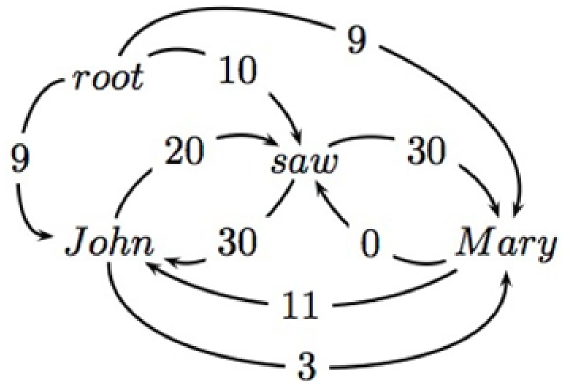


Figure 1: Caption

Each arc has a learned weight representing how likely that dependency is.

19.8 Chu–Liu–Edmonds Algorithm

To find MST in directed graphs, we use the Chu–Liu–Edmonds algorithm.

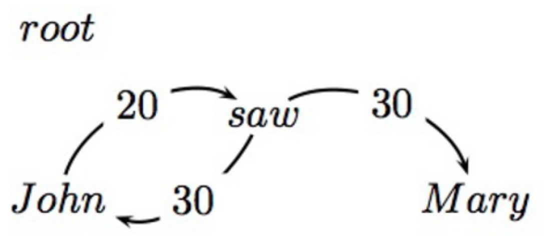
Step 1: Select Best Incoming Edge

For every vertex (except root), select the highest scoring incoming edge.
For example:

- John chooses saw $\rightarrow$ John (30)

- Mary chooses saw → Mary (30)
- saw chooses root → saw (10)

If this forms a tree (no cycle), we are done.



Step 2: Detect Cycle

If a cycle exists, identify it.
Suppose John ↔ saw form a cycle.

Step 3: Contract Cycle

Collapse the cycle into a single super-node w_{js}.
Graph becomes smaller.

19.9 Recalculate Arc Weights

Two types of edges must be recalculated:

Outgoing Edges from Cycle

Outgoing arc weight equals:

$$\text{max(all outgoing arcs from nodes inside cycle)}$$

Example:

$$\max(John \to Mary = 3,\ saw \to Mary = 30) = 30$$

Incoming Edges to Cycle

Incoming arc weight equals weight of best tree including that incoming arc.
Example:

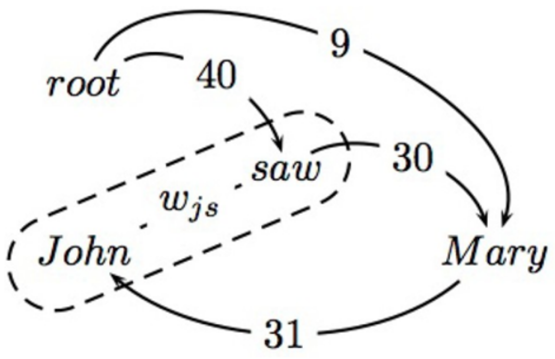
$$root \to saw \to John = 40$$

$$root \to John \to saw = 29$$

We choose the maximum.

19.10 Recursive Call

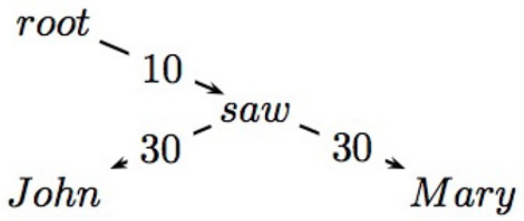
After contraction and weight recalculation:
If this graph is a tree, we have found MST for contracted graph.



19.11 Expand the Cycle

Now we expand the cycle and reconstruct final tree.

Final MST:



Thus the dependency tree is:

- root → saw
- saw → John
- saw → Mary

19.12 Conceptual Understanding

Transition-based parsing is local and incremental.

MST-based parsing is global and optimizes total score at once.

Arc-Eager builds tree step by step.

MST selects best global structure from all possible trees.

19.13 Summary

MST-based dependency parsing:

- Constructs fully connected weighted directed graph.
- Defines tree score as sum of arc weights.
- Uses Chu–Liu–Edmonds algorithm.
- Handles cycles via contraction.
- Guarantees globally optimal tree.

20 Models for Ambiguity Parsing

20.1 Ambiguity in Grammar

For an ambiguous grammar, a sentence may have more than one valid derivation (parse tree). This creates structural ambiguity. To resolve ambiguity, we associate scores with parse trees and select the most probable one. One way to achieve this is by modifying a standard Context-Free Grammar (CFG) into a Probabilistic Context-Free Grammar (PCFG).

20.2 Probabilistic Context-Free Grammar (PCFG)

A PCFG is defined as:

$$G = (T, N, S, R, P)$$

where:

- T : Set of terminal symbols
- N : Set of non-terminal symbols
- S : Start symbol
- R : Set of production rules
- P : Probability function over rules

Each production rule has the form:

$$X \to \gamma, \quad X \in N, \quad \gamma \in (T \cup N)^*$$

Each rule is assigned a probability such that:

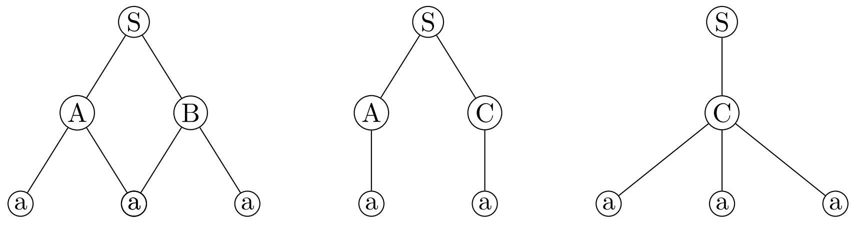
$$\forall X \in N, \quad \sum_{\gamma} P(X \to \gamma) = 1$$

Thus, probabilities of all rules expanding the same non-terminal must sum to 1.

20.3 Estimating Rule Probabilities from a Treebank

Rule probabilities can be estimated from a treebank (annotated parse trees).

Suppose we have three trees:



Assume:

$$t_1 = 10, \quad t_2 = 20, \quad t_3 = 50$$

Then:

$$P(S \to AB) = \frac{10}{10 + 20 + 50} = 0.125$$

$$P(S \to AC) = \frac{20}{80} = 0.25$$

$$P(S \to C) = \frac{50}{80} = 0.625$$

Similarly:

$$P(A \to aa) = \frac{10}{10 + 20} = 0.33$$

$$P(A \to a) = \frac{20}{30} = 0.67$$

$$P(B \to aa) = \frac{20}{20 + 50} = 0.285$$

$$P(C \to aaa) = \frac{50}{70} = 0.714$$

20.4 Limitations of PCFG

In a PCFG, a non-terminal can be expanded independently of context. This independence assumption is often unrealistic and may lead to incorrect modeling of syntactic dependencies.

20.5 Improvements to PCFG

Three main approaches improve PCFG performance:

1. **Parent Annotation**: Non-terminals are annotated with parent labels to incorporate context.
2. **Automated Non-terminal Splitting (Split-Merge)**:
    - Unsupervised method
    - Uses EM algorithm
    - Learns refined non-terminal categories
3. **Lexicalization of Non-terminals**:
    - Incorporates head words
    - Models word-level dependencies

20.6 Alternative: Max-Rule Parsing

Instead of choosing the most probable parse tree, we maximize the expected number of correct constituents.

In CKY parsing, the scoring function for span $X[i, j]$ is replaced with inside-outside probability products.

This approach is known as **Max-Rule Parsing**.

21 Generative Model Parsing

A parse tree is constructed as a sequence of decisions:

$$D = d_1, d_2, \ldots, d_n$$

Each derivation represents a sequence of decisions.

The probability of a parse is:

$$P(x, y) = P(d_1, \ldots, d_n)$$

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

The sequence $d_1, \ldots, d_{i-1}$ is called the **history**.
Simplification is done by grouping histories into equivalence classes using a function ϕ.

22 Discriminative Models

Discriminative models directly model:

$$\text{Input: } x$$

$$\text{Output: } y$$

Each pair (x, y) is mapped to a feature vector:

$$\phi(x, y) \in R^d$$

Each dimension represents a feature capturing partial information.
A weight vector:

$$w \in R^d$$

assigns importance to each feature.

22.1 Scoring Function

$$Score(x, y) = \phi(x, y) \cdot w$$

Higher score indicates a more plausible parse.
Prediction:

$$y^* = \arg \max_{y \in GEN(x)} \phi(x, y) \cdot w$$

23 Perceptron Learning Algorithm

Perceptron is a single-layer online learning algorithm.
Prediction:

$$y' = \arg \max_{y \in GEN(x)} \phi(x, y) \cdot w$$

Update rule:

$$w = w + \phi(x, y) - \phi(x, y')$$

Increase weights for correct features and decrease for incorrect ones.

24 Voted Perceptron

Stores all intermediate weight vectors.

Each weight vector w_i has a count c_i.

Prediction:

$$y^* = \arg\max \sum_i c_i(\phi(x, y) \cdot w_i)$$

Advantages:

- More stable
- Reduces overfitting

25 Averaged Perceptron

Instead of voting, compute average weight:

$$w_{avg} = \frac{1}{mT} \sum w_i$$

Maintains:

- Current weight vector w
- Accumulated vector v

Final average:

$$w_{avg} = \frac{v}{mT}$$

Reduces space and time complexity.

26 Lazy Update

Used when feature space is very large.

- Only update features appearing in current example.
- Avoid updating all dimensions.
- Maintain update timestamp.

This significantly reduces time complexity.