Chapter 12

Parsing Algorithms

12.1 Introduction

In this chapter, we explore the *parsing* problem, which encompasses several questions, including:

- Does $L(G)$ contain w?
- What is the highest-weight derivation of w?
- What is the set of all derivations of w?

12.2 Chomsky normal form

Let's assume that G has a particularly simple form. We say that a CFG is in *Chomsky normal form* if each of its productions has one of the following forms:

$$X \rightarrow YZ$$
$$X \rightarrow a$$

It can be shown (see below) that any context-free grammar not generating a language containing ϵ can be converted into Chomsky normal form, and still generate the same language.

Our grammar from above can be massaged to be in Chomsky normal form:

$$\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow \text{DT NN} \\
\text{NP} &\rightarrow \text{time} \mid \text{fruit} \\
\text{NP} &\rightarrow \text{NN NNS} \\
\text{VP} &\rightarrow \text{VBP NP} \\
\text{VP} &\rightarrow \text{flies} \\
\text{VP} &\rightarrow \text{VP PP} \\
\text{PP} &\rightarrow \text{IN NP} \\
\text{DT} &\rightarrow \text{a} \mid \text{an} \\
\text{NN} &\rightarrow \text{time} \mid \text{fruit} \mid \text{arrow} \mid \text{banana} \\
\text{NNS} &\rightarrow \text{flies} \\
\text{VBP} &\rightarrow \text{like} \\
\text{IN} &\rightarrow \text{like}
\end{aligned} \tag{12.1}$$

12.3 The CKY algorithm

The *CKY algorithm* is named after three people who independently invented it: Cocke, Kasami, and Younger, although it has been rediscovered more times than that.

In its most basic form, the algorithm just decides whether $w \in L(G)$. It builds a data structure known as a *chart*; it is an $n \times n$ array. The element *chart*$[i, j]$ is a set of nonterminal symbols. If $X \in$ *chart*$[i, j]$, then that means we have discovered that $X \Rightarrow^* w_{i+1} \cdots w_j$.

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: $w \in L(G)$ iff $S \in$ *chart*$[0, n]$

initialize *chart*$[i, j] \leftarrow \emptyset$ **for all** $0 \leq i < j \leq n$
for all $i \leftarrow 1, \ldots, n$ **and** $(X \rightarrow w_i) \in R$ **do**
  chart$[i-1, i] \leftarrow$ *chart*$[i-1, i] \cup \{X\}$
end for
for $\ell \leftarrow 2, \ldots, n$ **do**
  for $i \leftarrow 0, \ldots, n - \ell$ **do**
    $j \leftarrow i + \ell$
    for $k \leftarrow i+1, \ldots, j-1$ **do**
      for all $(X \rightarrow Y Z) \in R$ **do**
        if $Y \in$ *chart*$[i, k]$ **and** $Z \in$ *chart*$[k, j]$ **then**
          chart$[i, j] \leftarrow$ *chart*$[i, j] \cup \{X\}$
        end if
      end for
    end for
  end for
end for

Question 7. What is the time and space complexity of this algorithm?

Question 8. Using the grammar (12.1), run the CKY algorithm on the string:

$${}_0\ \text{time}\ {}_1\ \text{flies}\ {}_2\ \text{like}\ {}_3\ \text{an}\ {}_4\ \text{arrow}\ {}_5$$

0,1	0,2	0,3	0,4	0,5
	1,2	1,3	1,4	1,5
		2,3	2,4	2,5
			3,4	3,5
				4,5

12.4 Viterbi CKY

But it is much more useful to find the highest-weight parse. Suppose that our grammar has the following probabilities:

$$
\begin{array}{rlrl}
\mathrm{S} & \xrightarrow{1} \mathrm{NP\ VP} & \mathrm{DT} & \xrightarrow{0.5} \mathrm{a} \\
\mathrm{NP} & \xrightarrow{0.5} \mathrm{DT\ NN} & \mathrm{DT} & \xrightarrow{0.5} \mathrm{an} \\
\mathrm{NP} & \xrightarrow{0.2} \mathrm{time} & \mathrm{NN} & \xrightarrow{0.25} \mathrm{time} \\
\mathrm{NP} & \xrightarrow{0.2} \mathrm{fruit} & \mathrm{NN} & \xrightarrow{0.25} \mathrm{fruit} \\
\mathrm{NP} & \xrightarrow{0.1} \mathrm{NN\ NNS} & \mathrm{NN} & \xrightarrow{0.25} \mathrm{arrow} \\
\mathrm{VP} & \xrightarrow{0.6} \mathrm{VBP\ NP} & \mathrm{NN} & \xrightarrow{0.25} \mathrm{banana} \\
\mathrm{VP} & \xrightarrow{0.3} \mathrm{flies} & \mathrm{NNS} & \xrightarrow{1} \mathrm{flies} \\
\mathrm{VP} & \xrightarrow{0.1} \mathrm{VP\ PP} & \mathrm{VBP} & \xrightarrow{1} \mathrm{like} \\
\mathrm{PP} & \xrightarrow{1} \mathrm{IN\ NP} & \mathrm{IN} & \xrightarrow{1} \mathrm{like}
\end{array}
\tag{12.2}
$$

Then we use a modification of CKY that is analogous to the Viterbi algorithm. First, we modify the algorithm to find the maximum weight:

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: $\mathit{best}[0, n][S]$ is the maximum weight of a parse of w

initialize $\mathit{best}[i, j][X] \leftarrow 0$ **for all** $0 \leq i < j \leq n, X \in N$
for all $i \leftarrow 1, \ldots, n$ **and** $(X \xrightarrow{p} w_i) \in R$ **do**
    $\mathit{best}[i-1, i][X] \leftarrow \max\{\mathit{best}[i-1, i][X], p\}$
end for
for $\ell \leftarrow 2, \ldots, n$ **do**
    for $i \leftarrow 0, \ldots, n-\ell$ **do**
        $j \leftarrow i + \ell$
        for $k \leftarrow i+1, \ldots, j-1$ **do**
            for all $(X \xrightarrow{p} Y Z) \in R$ **do**
                $p' \leftarrow p \times \mathit{best}[i, k][Y] \times \mathit{best}[k, j][Z]$
                $\mathit{best}[i, j][X] \leftarrow \max\{\mathit{best}[i, j][X], p'\}$
            end for
        end for
    end for
end for

Question 9. Do you see how to modify the algorithm to compute the *total* weight of all parses of w?

A slight further modification lets us find the maximum-weight parse itself. Just as in the Viterbi algorithm for FSAs, whenever we update $\mathit{best}[i, j][X]$ to a new best weight, we also need to store a *back-pointer* that records how we obtained that weight. We will represent back-pointers like this: $X_{i,j} \rightarrow Y_{i,k}Z_{k,j}$ means that we built an X spanning i, j from a Y spanning i, k and a Z spanning k, j.

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: G' generates the best parse of w
Ensure: *best*$[0, n][S]$ is its weight

```
for all 0 ≤ i < j ≤ n, X ∈ N do
    initialize best[i, j][X] ← 0
    initialize back[i, j][X] ← nil
end for
for all i ← 1, ..., n and (X -p-> w_i) ∈ R do
    if p > best[i − 1, i][X] then
        best[i − 1, i][X] ← p
        back[i − 1, i][X] ← (X_{i−1,i} → w_i)
    end if
end for
for ℓ ← 2, ..., n do
    for i ← 0, ..., n − ℓ do
        j ← i + ℓ
        for k ← i + 1, ..., j − 1 do
            for all (X -p-> Y Z) ∈ R do
                p' ← p × best[i, k][Y] × best[k, j][Z]
                if p' > best[i, j][X] then
                    best[i, j][X] ← p'
                    back[i, j][X] ← (X_{i,j} → Y_{i,k} Z_{k,j})
                end if
            end for
        end for
    end for
end for
G' = {back[i, j][X] | 0 ≤ i < j ≤ n, X ∈ N}
```

G' is then a grammar that generates at most one tree, the best tree for w.

Question 10. Using the grammar (12.2), run the Viterbi CKY algorithm on the same string:

$$_0 \text{ time } _1 \text{ flies } _2 \text{ like } _3 \text{ an } _4 \text{ arrow } _5$$

0,1	0,2	0,3	0,4	0,5
	1,2	1,3	1,4	1,5
		2,3	2,4	2,5
			3,4	3,5
				4,5

12.5 Parsing general CFGs

Previously, we learned about PCFGs, and how to find the best PCFG derivation of a string using the Viterbi algorithm. Now we will extend those algorithms to the general CFG case.

12.5.1 Binarization

It turns out that any CFG (whose language does not contain ϵ) can be converted into an equivalent grammar in Chomsky normal form.

To guarantee that $k \leq 2$, we must eliminate all rules with right-hand side longer than 2. We will see below that the grammars we extract from training data may already have this property. But if not, we need to *binarize* the grammar. For example, suppose we have the production

$$\text{NP} \rightarrow \text{DT JJS NN NN PP} \tag{12.3}$$

which is too long to be in Chomsky normal form. There are many ways to break this down into smaller rules, but here is one way. We create a bunch of new nonterminal symbols NP(β) where β is a string of nonterminal symbols; this stands for a partial NP whose sisters to the *left* are β. Then we replace rule (12.3) with:

$$\text{NP} \rightarrow \text{DT NP(DT)} \tag{12.4}$$
$$\text{NP(DT)} \rightarrow \text{JJS NP(DT,JJS)} \tag{12.5}$$
$$\text{NP(DT,JJS)} \rightarrow \text{NN NP(DT,JJS,NN)} \tag{12.6}$$
$$\text{NP(DT,JJS,NN)} \rightarrow \text{NN NP(DT,JJS,NN,NN)} \tag{12.7}$$
$$\text{NP(DT,JJS,NN,NN)} \rightarrow \text{PP} \tag{12.8}$$

Note that the annotations contain enough information to reverse the binarization. So the binarized grammar is equivalent to the unbinarized grammar, but has $k \leq 2$.

12.5.2 Parsing with unary rules

But we are not done yet. CKY does not just require $k \leq 2$, but also forbids rules of any of the following forms:

$$A \rightarrow ab \tag{12.9}$$
$$A \rightarrow aB \tag{12.10}$$
$$A \rightarrow Ab \tag{12.11}$$
$$A \rightarrow \epsilon \tag{12.12}$$
$$A \rightarrow B \tag{12.13}$$

The first three cases are very easy to eliminate, but we never see them in grammars induced from the Penn Treebank. *Nullary rules* (12.12) are not hard to eliminate (Hopcroft and Ullman, 1979), but the weighted case can be nasty (Stolcke, 1995). Fortunately, nullary rules aren't very common in practice, so we won't bother with them here.

Unary rules (12.13) are quite common and annoying. Like nullary rules, they are not hard to eliminate from a CFG (Hopcroft and Ullman, 1979), but in practice, most people don't try to; instead, they extend the CKY algorithm to handle them directly. The extension shown below is not the most efficient, but fits most naturally with the way we have implemented CKY.

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: $w \in L(G)$ iff $S \in \textit{chart}[0, n]$
initialize $\textit{chart}[i, j] \leftarrow \emptyset$ **for all** $0 \leq i < j \leq n$
for all $i \leftarrow 1, \ldots, n$ **and** $(X \rightarrow w_i) \in R$ **do**
    $\textit{chart}[i-1, i] \leftarrow \textit{chart}[i-1, i] \cup \{X\}$
end for
for $\ell \leftarrow 2, \ldots, n$ **do**
    for $i \leftarrow 0, \ldots, n - \ell$ **do**
        $j \leftarrow i + \ell$
        for $k \leftarrow i+1, \ldots, j-1$ **do**
            for all $(X \rightarrow YZ) \in R$ **do**
                if $Y \in \textit{chart}[i, k]$ **and** $Z \in \textit{chart}[k, j]$ **then**
                    $\textit{chart}[i, j] \leftarrow \textit{chart}[i, j] \cup \{X\}$
                end if

13: **end for**
14: **end for**
15: *again* $\leftarrow$ **true**
16: **while** *again* **do**
17: *again* $\leftarrow$ **false**
18: **for all** $(X \rightarrow Y) \in R$ **do**
19: **if** $X \notin \mathit{chart}[i,j]$ **and** $Y \in \mathit{chart}[i,j]$ **then**
20: $\mathit{chart}[i,j] \leftarrow \mathit{chart}[i,j] \cup \{X\}$
21: *again* $\leftarrow$ **true**
22: **end if**
23: **end for**
24: **end while**
25: **end for**
26: **end for**

The new part is lines 15–24 and is analogous to the binary rule case.

Question Why is the **while** loop on line 16 necessary? What is its maximum number of iterations?

Here's how to modify the Viterbi CKY algorithm to allow unary rules.

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: G' generates the best parse of w
Ensure: $\mathit{best}[0,n][S]$ is its weight

for all $0 \leq i < j \leq n, X \in N$ **do**
initialize $\mathit{best}[i,j][X] \leftarrow 0$
initialize $\mathit{back}[i,j][X] \leftarrow$ nil
end for
for all $i \leftarrow 1, \ldots, n$ **and** $(X \xrightarrow{p} w_i) \in R$ **do**
if $p > \mathit{best}[i-1,i][X]$ **then**
$\mathit{best}[i-1,i][X] \leftarrow p$
$\mathit{back}[i-1,i][X] \leftarrow (X_{i-1,i} \rightarrow w_i)$
end if
end for
for $\ell \leftarrow 2, \ldots, n$ **do**
for $i \leftarrow 0, \ldots, n-\ell$ **do**
$j \leftarrow i + \ell$
for $k \leftarrow i+1, \ldots, j-1$ **do**
for all $(X \xrightarrow{p} YZ) \in R$ **do**
$p' \leftarrow p \times \mathit{best}[i,k][Y] \times \mathit{best}[k,j][Z]$
if $p' > \mathit{best}[i,j][X]$ **then**

$best[i,j][X] \leftarrow p'$
$back[i,j][X] \leftarrow (X_{i,j} \rightarrow Y_{i,k} Z_{k,j})$
end if
end for
end for
$again \leftarrow$ **true**
while *again* **do**
$again \leftarrow$ **false**
for all $(X \xrightarrow{p} Y) \in R$ **do**
$p' \leftarrow p \times best[i,j][Y]$
if $p' > best[i,j][X]$ **then**
$best[i,j][X] = p'$
$back[i,j][X] \leftarrow (X_{i,j} \rightarrow Y_{i,j})$
$again \leftarrow$ **true**
end if
end for
end while
end for
end for
$G' \leftarrow \text{extract}(S, 0, n)$

If the grammar has unary cycles in it, that is, it is possible to derive $X \Rightarrow \ldots \Rightarrow^* X$, then certain complications can arise from the fact that a string may have an infinite number of derivations. In particular, if the weight of the cycle is greater than 1, then the Viterbi CKY algorithm will break. Even if all rule weights are less than 1, some algorithms require modification; for example, if we want to find the total weight of all the derivations of a string, we have to perform an infinite summation (Stolcke, 1995). Therefore, it is fairly common to implement hacks of various kinds to break the cycles. For example, we could modify the grammar so that it goes round the cycle at most five times.

Question 11. Why doesn't the Viterbi CKY algorithm break on unary cycles if we assume that all rule weights are less than 1?

Bibliography

Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley.

Stolcke, Andreas (1995). "An Efficient Probabilistic Context-Free Parsing Algorithm that Computes Prefix Probabilities". In: *Computational Linguistics* 21, pp. 165–201.