Two-level probabilistic grammars for natural language parsing

Infante Lopez, G.G.

Citation for published version (APA):
Appendix A

Parsing PCW-Grammars

A.1 Introduction

In this appendix we focus on two aspects of our PCW-grammar parsing algorithm. One aspect, the most theoretical one, is related to the study of the capacity and computational complexity of our implementation for handling PCW-grammars. Since various meta-derivations can produce the same w-rule, it is important to distinguish between the most probable derivation tree and the most probable w-tree. As described in Chapter 4, different meta-derivations can yield the same w-rule and consequently the same w-tree. A parser returning the most probable derivation tree considers the probability of a w-rule as the probability value of the most probable meta-derivation. In contrast, a parsing algorithm searching for the most probable tree considers the probability of a w-rule as the sum of all probabilities assigned to the meta-derivations producing it. In this appendix we investigate the differences between these two approaches, focusing on necessary and sufficient conditions for both approaches to return the same tree.

The second aspect we focus on, is related to technical issues of our PCW-grammar parsing algorithm implementation. In some of the experiments we performed, the parser had to handle grammars containing a number of rules close to one million. The parsing algorithm is an optimization algorithm, it searches for the best solution among a set of possible solutions. At each step in the optimization process, the algorithm builds possible solutions retrieving new rules from the grammar. When working with large grammars, as we did, the complexity of the parsing algorithm becomes unmanageable if the retrieval step takes more than constant time. In this appendix, we briefly describe the approach we followed to minimize the computational costs of this step.

The rest of the appendix is organized as follows. Section A.2 discusses the theoretical issues related to the parsing algorithm, Section A.3 discusses the Java implementation, and Section A.4 concludes the appendix.
Appendix A. Parsing PCW-Grammars

A.2 Theoretical Issues

Recall from Chapter 4 that a PCW-grammar is a 6-tuple \( (V, NT, T, S, \xrightarrow{m}, \xrightarrow{s}) \) such that:

- \( V \) is a set of symbols called **variables**. Elements in \( V \) are denoted with over-lined capital letters, e.g., \( \overline{A}, \overline{B}, \overline{C} \).

- \( NT \) is a set of symbols called **non-terminals**: elements in \( NT \) are denoted with upper-case letters, e.g., \( X, Y, Z \).

- \( T \) is a set of symbols called **terminals**, denoted with lower-case letters, e.g., \( a, b, c \), such that \( V, T \) and \( NT \) are pairwise disjoint.

- \( S \) is an element of \( NT \) called **start symbol**.

- \( \xrightarrow{m} \) is a finite binary relation defined on \( (V \cup NT \cup T)^* \) such that if \( x \xrightarrow{m} y \), then \( x \in V \). The elements of \( \xrightarrow{m} \) are called **meta-rules**.

- \( \xrightarrow{s} \) is a finite binary relation on \( (V \cup NT \cup T)^* \) such that if \( u \xrightarrow{s} v \) then \( u \in NT, v \neq \epsilon \) and \( v \) does not have any variable appearing more than once. The elements of \( \xrightarrow{s} \) are called **pseudo-rules**.

Meta-rules and pseudo-rules have probabilities associated to them, see Example A.2.1 for an example of a w-grammar.

**A.2.1. Example.** Let \( W = (V, NT, T, S, \xrightarrow{m}, \xrightarrow{s}) \) be a W-grammar, where \( V = \{A, C\}, NT = \{B, S\}, T = \{a, c\}, \xrightarrow{m} \) and \( \xrightarrow{s} \) as described in Table A.1.

<table>
<thead>
<tr>
<th>pseudo-rules (( \xrightarrow{s} ))</th>
<th>meta-rules (( \xrightarrow{m} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \xrightarrow{s,0.5} A )</td>
<td>( A \xrightarrow{m,0.5} aC )</td>
</tr>
<tr>
<td>( S \xrightarrow{s,0.5} B )</td>
<td>( A \xrightarrow{m,0.5} \overline{C}a )</td>
</tr>
<tr>
<td>( B \xrightarrow{s,0.75} aa )</td>
<td>( \overline{C} \xrightarrow{m,1} a )</td>
</tr>
<tr>
<td>( B \xrightarrow{s,0.25} cc )</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: A W-grammar that has a best derivation tree that does not correspond to the most probable tree.

As described in Chapter 4, there are two types of derivations depending on the type of the rules used to produce them. **Meta-derivations** are derivations in which only meta-rules are used, while **w-derivations** are derivations in which only w-rules are used. Since w-rules are built by meta-deriving all variables in a pseudo-rule, there might be
w-rules that are the product of different meta-derivations. We can think of a w-rule as a way to pack all meta-derivations that yield the w-rule, because the probability assigned to the w-rule is the sum of all the possible meta-derivations it covers.

Since a w-rule covers many meta-derivations, the underlying PCFG cannot be used for parsing PCW grammars. A parser for PCFGs can be used for parsing PCW-grammars if and only if all possible w-rules cover one and only one meta-derivation. If this is not the case, the w-tree resulting from parsing with the underlying PCFG plus hiding its meta-derivations might not be the w-tree with the highest probability.

In order to better understand this phenomenon, we use the grammar in Example A.2.1. This grammar produces the two w-trees pictured in Figure A.1.a and Figure A.1.b, both of them yielding “aa”. Clearly, the most probable w-tree is the tree in Figure A.1.a, given that it is the one with the highest probability.

While trees in (a) and (b) are trees belonging to the forest of the PCW-grammar, trees in (c), (d) and (e) are trees that belong to the forest of the underlying PCFG. The procedure for hiding meta-rules maps trees (c) and (d) to tree (a), and (e) to (b).

<table>
<thead>
<tr>
<th>w-trees</th>
<th>underlying PCFG trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>p = 0.5</td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>p = 0.25</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>p = 0.25</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>p = 0.375</td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>p = 0.375</td>
<td></td>
</tr>
<tr>
<td>(e)</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: Trees (a) and (b) belong to the forest of the W-grammar in Example A.2.1, while trees in (c), (d) and (e) are trees in the forest of the PCFG underlying the same W-grammar.

The PCFG parser using the PCFG underlying searches for the best parser among the trees that belong to the forest generated by the PCFG underlying, i.e., the parser searches for the best among the trees in the right-hand side of Figure A.1. Once the best tree is found, it is mapped to a w-tree by hiding all meta-derivations. In this example, the most probable tree in the forest generated by the PCFG underlying is the tree in part (e), which is mapped to the w-tree in part (b). Clearly, the w-tree with the highest probability is the tree in part (a) and not the one in part (b). The algorithm failed in returning the most probable tree. In other words, the PCW parser defined as
the procedure of, first, searching for the most probable tree in the forest generated by
the PCFG underlying and, second, hiding all of its meta-derivations, might not return
the most probable w-tree.

Clearly, if the hiding procedure maps one tree in the forest generated by the PCFG
underlying to one tree in the forest generated by the W-grammar, then a parser for W-
grammar is equivalent to the process of using a PCFG parser plus post tree-processing.

There are two configurations for which the mapping between the two forests is not
a one-to-one map. The first one occurs when there is at least one meta-variable that
can be instantiated with a value that can be meta-derived in two different ways. The
second one occurs when there is a pseudo-rule that has only one terminal in its body,
and that body can be generated with another w-rule. For the grammar in Example A.1,
the mapping between two forests is not a one-to-one mapping because the variable \( \overline{A} \)
in pseudo-rule \( S \xrightarrow{a,0.5} \overline{A} \) can be instantiated with two different meta-derivations.

Note that in all our experiments, meta-rules come from probabilistic determinis-
tic automata. Since they are deterministic, all possible variable instantiations have a
unique way to derive them. Also, not all the pseudo-rules we used in our grammars
have a variable in their body. Consequently, since, for all the grammars we developed
in this thesis, there is a one-to-one mapping between the forest generated by the w-
grammar and the forest generated by the PCFG underlying, we decided to implement
our parser as a Cocke-Younger-Kasami (CYK) parsing algorithm plus a procedure that
hides all meta-derivations from the tree returned by the CYK algorithm.

The parser we implemented can be used as a parser that returns the most probable
tree, because we know that the most probable tree corresponds to the most probable
derivation tree. The problem of knowing when these two trees are the same is not
a trivial one. As we discussed before, the two trees are the same tree if there are
no variables that can be instantiated in the same way with two different derivations.
Since variables are instantiated through a context free like system, the problem of
knowing whether there are two ways to derive the same string becomes equivalent to
the problem of knowing whether a context free grammar is unambiguous. It is well-
known that the latter is an undecidable problem, which implies that it is is undecidable
whether the most probable tree is the same as the most probable derivation tree for a
given grammar.

A.3 Practical Issues

Conceptually, our parsing algorithm consists of two different modules. One module,
a CYK parser, searches for the most probable derivation in the underlying PCFG (see
Chapter 4 for the definition of the underlying PCFG). The second module, the function
devoted to hiding meta-derivations, is in charge of transforming the most probable
derivation tree into a w-tree.

The rules handled by the version of the CYK algorithm we implemented have an
number associated to them. This integer, called level of visibility, is a generalization of
the concept of meta-rules and pseudo-rules. The tree returned by the CYK algorithm
is transformed to many different trees, depending on the visibility level of the rules to
be hidden.

A.3.1 Levels of Visibility

Trees to be transformed can be thought of as trees in which, for each node, there is an
integer marking the node's level of visibility. In order to transform a tree by hiding a
level of visibility, we implemented a function that takes two arguments, one argument
is the tree to be transformed and the second argument is the level of visibility to be
hidden. The algorithm traverses the tree in a bottom up fashion and, for each node
having the visibility level to be replaced, it replaces the node itself with the sub-trees
hanging from that node in the original tree. Figure A.2 shows an example of hiding
operations for different levels of visibility. Since the hide operation can be applied to
a tree which was already transformed, a tree has many possible sets of visibility levels
to hide. For example, the tree in Figure A.2.d is the result of hiding level of visibility 1
from the tree in Figure A.2.c, which is in turn the result of hiding visibility level 2 from
the tree in Figure A.2.a. Note that the order in which the visibility levels are applied
does not matter.

The PCW parser is a particular case of the parser we implemented. In order to
obtain a PCW-parser, we marked meta-rules with visibility level 1 and pseudo-rules
with visibility level 0. In order to obtain a w-tree from the tree output by the CYK
component, we hide nodes whose level of visibility equals 1.

A.3.2 Optimization Aspects

The core of our algorithm is a Probabilistic CYK parsing algorithm capable only of
 parsing grammars in Chomsky Normal Form (CNF). Probabilistic CYK parsing was
first described by Ney (1991), but the version we discuss here is adapted from Collins

The CYK algorithm assumes the following input, output and data structures.

- **Input**

  - A CNF PCFG. Assume that the $|N|$ non-terminals have indices $1, 2, \ldots, |N|$, and that the start symbol $S$ has index 1.
Appendix A. Parsing PCW-Grammars

- \( n \) words \( w_1, \ldots, w_n \).

- **Data structures.** A dynamic programming array \( \pi[i, j, a] \) holds the maximum probability for a constituent with non-terminal index \( a \) spanning words \( i \ldots j \).

- **Output.** The maximum probability parse will be \( \pi[1, n, 1] \): the parse tree whose root is \( S \) and which spans the entire string of words \( w_1, \ldots, w_n \).

The CYK algorithm fills out the probability array by induction. Figure A.3 gives the pseudo-code for this probabilistic CYK algorithms as it appears in (Jurafsky and Martin, 2000).

Note that steps 10, 11 and 12 are actually building all possible rules that can be built using the non-terminals of the grammar. In order to minimize the number of iterations, we iterate only on those rules that actually belong to the grammar and that can help building the solution. In order to achieve this, we implement our grammars as dictionaries indexed on bodies of rules. This approach is easy to implement because bodies of rules are of length one or two. Unfortunately, this modification does not reduce the worst case complexity, because in that case the grammar contains all the possible rules that can be built with its set of non-terminals.

In order to parse with a grammar that is not in CNF, our parsing algorithm first transforms the given grammar into an equivalent grammar in CNF. Clearly, since the
function CYK(words, grammar) returns The most probable parse and its probability.

1: Create and clear $\pi[num\_words, num\_words, num\_nonterminals]$ {Base case}
2: for $i \leftarrow 1$ to $num\_words$ do
3:   for $A \leftarrow w_i$ to $num\_nonTerminals$ do
4:     if $A \rightarrow w_i$ is in the grammar then
5:       $\pi[i, i,] \leftarrow P(A \rightarrow w_i)$  
   {recursive case}
6: for span $\leftarrow 2$ to $num\_words$ do
7:   for begin $\leftarrow 1$ to $num\_words - span + 1$ do
8:     end $\leftarrow begin + span - 1$
9:   for $m = begin$ to $end - 1$ do
10:      for $A = 1$ to $num\_nonterminals$ do
11:         for $B = 1$ to $num\_nonterminals$ do
12:            for $C = 1$ to $num\_nonterminals$ do
13:               prob $= \pi[begin, m, B] \times \pi[m + 1, end, C] \times P(A \rightarrow BC)$
14:               if prob $> \pi[begin, end, A]$ then
15:                 $\pi[begin, end, A] = prob$
16:                 back[begin, end, A] = \{m, B, C\}
17: return buildtree(back[1, num\_words, 1], $\pi[1, num\_words, 1]$)

Figure A.3: The probabilistic CYK algorithm for finding the maximum probability parse of a string of $num\_words$ words given a PCFG grammar with $num\_rules$ rules in Chomsky Normal Form. $back$ is an array of back-pointers used to recover the best parse.

algorithm parses with a CNF, it will return a tree generated by the CNF grammar and not by the original grammar. Since we are interested in the tree generated by the grammar before it was transformed to CNF, we use a new level of visibility $j$ in all the rules that were added during the transformation into CNF process. In order to obtain the tree in the original grammar, we hide the level of visibility $j$.

Summing up, our parsing algorithm consists of the following items:

1. A translation module: An algorithm that transforms any grammar into CNF.
3. A post processing module: An algorithm that hides levels of visibility in a tree.

We compute now the computational complexity of our parsing algorithm. For this purpose we only take into consideration items (2) and (3). Since item (1) is done one
time for each grammar, we only consider it indirectly: we take into consideration how
the transformation to CNF affects the size of the grammar.

We want to compute the computational complexity of the three items whenever a
grammar $G$ is used to parse a sentence $s$.

By (Ney, 1991), the computational complexity of the CYK algorithm for parsing a
sentence $s$ using grammar $G$ is $2nQ + (n^3/6)R$, where $n$ is the length of the sentence,
$Q$ the number of preterminal rules in the grammar, and $R$ the number of rules in the
grammar. Note that the number of rules and preterminal rules refer to the transformed
grammar.

According to (Hopcroft and Ullman, 1979), if a grammar $G$ with $R$ rules is trans-
formed to CNF, the resulting grammar contains $O(R^2)$ rules. The complexity of the
post processing time depends on the number of rules in the CNF tree. Since a CNF tree
yielding a sentence of length $n$ has $\sum_{i=1}^{n-1} i$ rules, the post processing step takes time
$\sum_{i=1}^{n-1} i$. Finally, the complexity for the whole algorithm becomes $2nQ + (n^3/6)R^2$.

A.4 Conclusions

In this appendix we have dealt with two particular aspects of our implementation. First,
we showed that it may happen that our implementation does not always return the most
probable tree. We also showed that there are some grammars for which our grammar
does return the most probable tree. For all grammars used in this thesis, however,
the parser does return the most probable tree. We also showed that the problem of
determining whether the parser returns the most probable tree or the most probable
derivation tree is undecidable.

The second aspect we focused on was related to the actual implementation of the
parser. We decomposed our implementation into three different modules, we gave
the computational complexity of each of them, and showed that the computational
complexity of the whole algorithm is $O(n^3R^2)$, where $n$ is the length of the sentence
and $R$ the size of the grammar.