Chapter 8

Advanced Parsing

8.1 Introduction

As we have seen, parsing builds trees over sentences, according to a phrase structure grammar. However, as the coverage of the grammar increases and the length of the input sentence grows, the number of parse trees grows rapidly. In fact, it grows at an astronomical rate.

Let’s explore this issue with the help of a simple example. The word *fish* is both a noun and a verb. We can make up the nonsense sentence *fish fish fish*, meaning *fish like to fish for other fish*. (Try this with *police* if you prefer something more sensible.) Here is a toy grammar for the ‘fish’ sentences.

```python
>>> from nltk_lite.parse import cfg, chart
>>> grammar = cfg.parse_grammar(""
... S -> NP V NP
... NP -> NP Sbar
... Sbar -> NP V | V NP
... NP -> 'fish'
... V -> 'fish'
... "")
```

Now we can try parsing a longer sentence, *fish fish fish fish fish*, which amongst other things, means *fish that are fished by other fish are in the habit of fishing fish themselves*. We use the NLTK chart parser, which is presented later on in this chapter. This sentence has four readings.

```python
>>> tokens = ['fish'] * 5
>>> cp = chart.ChartParse(grammar, chart.TD_STRATEGY)
>>> for tree in cp.get_parse_list(tokens):
...     print tree
(S:  
 (NP: (NP: 'fish') (Sbar: (V: 'fish') (NP: 'fish'))))
 (V: 'fish')
 (NP: 'fish'))
(S:  
 (NP: (NP: 'fish') (Sbar: (NP: 'fish') (V: 'fish')))  
 (V: 'fish')
 (NP: 'fish'))
(S:  
 (NP: 'fish')
 (V: 'fish'))
```
8.1. Introduction

As the length of this sentence goes up (3, 5, 7, ...) we get the following numbers of parse trees: 1; 4; 20; 112; 672; 4,224; 27,456; 183,040; 1,244,672; 8,599,552; 60,196,864; 426,008,576. The last of these — a figure of the order of $10^8$ — is for a sentence of length 23, the average length of sentences in the WSJ section of Penn Treebank. (This growth is super-exponential (equal to $2^n \cdot C(n+1)$, where $C(n)$ is the $n$th Catalan number, $(2n)!/(n!(n+1)!)$).) No practical NLP system could construct $10^8$ trees for a typical sentence, much less choose the appropriate one in the context. It’s clear that humans don’t do this either! Note that the problem is not with our choice of example. [ChurchPatil1982] point out that the syntactic ambiguity of PP attachment in sentences like (56) also grows in proportion to the Catalan numbers.

(56) Put the block in the box on the table.

Moreover, as soon as we try to construct a broad-coverage grammar, we are forced to make lexical entries highly ambiguous for their part of speech. In a toy grammar, a is only a determiner, dog is only a noun, and runs is only a verb. However, in a broad-coverage grammar, a is also a noun (e.g. part a), dog is also a verb (meaning to follow closely), and runs is also a noun (e.g. ski runs). In fact, all words can be referred to by name: e.g. the verb ’ate’ is spelled with three letters; in speech we do not need to supply quotation marks. Furthermore, it is possible to verb most nouns. Thus a parser for a broad-coverage grammar will be overwhelmed with ambiguity. Even complete gibberish will often have a reading, e.g. the a are of I. As [Abney, 1996] has pointed out, this is not word salad but a grammatical noun phrase, in which are is a noun meaning a hundredth of a hectare (or 100 sq m), and a and I are nouns designating coordinates:

![Figure 8.1: The a are of I](image)

Given this unlikely phrase, a broad-coverage parser should find this surprising reading. Similarly, sentences which seem to be unambiguous, such as John saw Mary, turn out to have other readings we would not have anticipated (as Abney explains). This ambiguity is unavoidable, and leads to horrendous inefficiency in parsing seemingly inoccuous sentences.

Let’s look more closely at the issue of efficiency. The top-down recursive-descent parser presented in Chapter 7 can be very inefficient, since it often builds and discards the same sub-structure many times over. (You should try the recursive-descent parser demo if you haven’t already: from nltk_lite. draw import srparser; srparser.demo()).
In this chapter, we will present two, independent, methods for dealing with ambiguity:

**Dynamic Programming and Chart Parsing:** These techniques allow us to derive the parses of an ambiguous sentence more efficiently.

**Probabilistic Parsing:** These techniques allow us to *rank* the parses of an ambiguous sentence on the basis of evidence from corpora.

### 8.2 Dynamic programming

The simple parsers discussed in Chapter 7 have significant limitations. The bottom-up shift-reduce parser can only find one parse, and it often fails to find a parse even if one exists. As just pointed out, the top-down recursive-descent parser can be very inefficient, and if the grammar contains left-recursive rules, it can enter into an infinite loop. In order to address these problems of completeness
8.2. Dynamic programming

and efficiency, we will explore a technique called **dynamic programming**, which stores intermediate results, and re-uses them when appropriate.

Dynamic programming is a general technique for designing algorithms which is widely used in natural language processing. The term ‘programming’ is used in a different sense to what you might expect, to mean planning or scheduling. Dynamic programming is used when a problem contains overlapping sub-problems. Instead of computing solutions to these sub-problems repeatedly, we simply store them in a lookup table.

In the remainder of this section we will introduce dynamic programming, but in a rather different context to syntactic parsing.

### 8.2.1 Sanscrit Meter

Pingala was an Indian author who lived around the 5th century B.C., and wrote a treatise on Sanscrit prosody called the *Chandas Shastra*. Virahanka extended this work around the 6th century A.D., studying the number of ways of combining short and long syllables to create a meter of length \( n \). He found, for example, that there are five ways to construct a meter of length 4: \( V_4 = \{LL, SSL, SLS, LSS, SSSS\} \). In general, we can split \( V_n \) into two subsets, those starting with \( L: \{LL, LSS\} \), and those starting with \( S: \{SSL, SLS, SSSS\} \). This is the clue for decomposing the problem:

\[
V_4 = \\
\text{LL, LSS; or L prefixed to each item of } V_2 = \{L, SS\} \\
\text{SSL, SLS, SSSS; or S prefixed to each item of } V_3 = \{SL, LS, SSS\}
\]

With this observation, we can write a little recursive function to compute these meters:

```python
>>> def virahanka1(n):
...     if n == 0:
...         return []
...     elif n == 1:
...         return ['S']
...     else:
...         s = ['S' + prosody for prosody in virahanka1(n-1)]
...         l = ['L' + prosody for prosody in virahanka1(n-2)]
...         return s + l
>>> virahanka1(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

Notice that, in order to compute \( V_4 \) we first compute \( V_3 \) and \( V_2 \). But to compute \( V_3 \), we need to first compute \( V_2 \) and \( V_1 \). This **call structure** is depicted in the following tree:

(57)

As you can see, \( V_2 \) is computed twice. This might not seem like a significant problem, but it turns out to be rather wasteful as \( n \) gets large: to compute \( V_{10} \) using this recursive technique, we would computes...
V_2^34\) times; for \(V_{20}\) we would compute \(V_2\) \(4,181\) times; for \(V_{30}\) we would compute \(V_2\) \(514,229\) times; and for \(V_{40}\) we would compute \(V_2\) \(63,245,986\) times! A much better alternative is to store the value of \(V_2\) in a table and look it up whenever we need it. The same goes for other values, such as \(V_3\) and so on. Here is a dynamic programming approach which computes the same result as the earlier program, only much more efficiently. It uses some auxiliary storage, a table called \texttt{lookup}:

```python
>>> lookup = [None] * 100
>>> lookup[0] = ['']
>>> lookup[1] = ['S']
>>> def virahanka2(n):
...     for i in range(n-1):
...         s = ['S' + prosody for prosody in lookup[i]]
...         l = ['L' + prosody for prosody in lookup[i+1]]
...         lookup[i+2] = s + l
...     return lookup[n]
```

This is the classic \texttt{bottom-up} approach to dynamic programming, where we fill up a table with solutions to all smaller sub-problems, then simply read off the result we are interested in. Notice that each sub-problem is only ever solved once.

However, this method is still wasteful for some applications, because it may compute solutions to sub-problems that are never used in solving the main problem. This wasted computation can be avoided using the \texttt{top-down} approach to dynamic programming:

```python
>>> lookup = [None] * 100
>>> lookup[0] = ['']
>>> lookup[1] = ['S']
>>> def virahanka3(n):
...     if not lookup[n]:
...         s = ['S' + prosody for prosody in virahanka3(n-1)]
...         l = ['L' + prosody for prosody in virahanka3(n-2)]
...         lookup[n] = s + l
...     return lookup[n]
```

Unlike the bottom-up approach, this approach is recursive. It avoids the huge wastage of our first version by checking whether it has previously stored the result. If not, it computes the result recursively and stores it in the table. The last step is to return the stored result.

This concludes our introduction to dynamic programming. In the next section we will see this method applied to syntactic parsing.

### 8.3 Chart Parsing

#### 8.3.1 Well-formed Substring Tables

We start off by defining a simple grammar.

```python
>>> from nltk_lite.parse import cfg
>>> grammar = cfg.parse_grammar(""
... S -> NP VP
... PP -> P NP
... NP -> Det N | NP PP
... VP -> V NP | VP PP
... Det -> 'the'
"")
```
As you can see, this grammar allows the VP opened the box on the floor to be analysed in two ways, depending on where the PP is attached.

Dynamic programming allows us to build the PP on the floor just once; we enter the information into a table, and then look it up when we need to use it as a subconstituent of either the object NP or the higher VP. The table that holds this information is often called a well-formed substring table (or WFST for short). We will show how to construct the WFST bottom-up so as to systematically record what syntactic constituents have been found.

Let’s set our input to be the sentence the kids opened the box on the floor. It is helpful to think of the input as being indexed like a Python list. We have illustrated this in Figure 8.14.

This allows us to say that, for instance, the word opened can be found at position 2, 3 in the input. This of course is what we get when we split an input string into a list of tokens.

```python
>>> sent = "the kids opened the box on the floor"
>>> tokens = sent.split()
>>> tokens[2:3]
[‘opened’]
```
We can also just use the start position of a token to refer to it. So we can also get hold of *opened* as `tokens[2]`. In a WFST, we record the position of the words by filling in cells in a triangular matrix. We will take the vertical axis of the matrix as giving the start position of a substring, and the horizontal axis as giving the end position. In such a matrix, therefore, we might expect to find the word *opened* in the cell with coordinates (2, 3). To make our life easier, however, we will look up the lexical category of each word, and place that category in the matrix instead. So cell (2, 3) will contain the entry `V`. More generally, if our input string is $a_1 a_2 \cdots a_n$, and our grammar contains a rule of the form $A \rightarrow a_i$, then we add $A$ to the cell $(i-1, i)$.

So, for every word in `tokens`, we can look up in our grammar productions what category it belongs to. (We make the simplifying assumption for now that there is no lexical ambiguity.)

```python
>>> grammar.productions(rhs=tokens[2])
[V -> 'opened']
```

For our WFST, we create an $n \times n$ matrix (where $n$ is the number of tokens plus 1) as a list of lists in Python, and we initialize the matrix with the lexical categories of each token in the input.

```python
>>> def init_wfst(tokens, grammar):
...     numtokens = len(tokens)
...     wfst = [["" for i in range(numtokens+1)] for j in range(numtokens+1)]
...     prod_rhs = grammar.productions(rhs=tokens[i])
...     wfst[i][i+1] = prod_rhs[0].lhs()
...     return wfst
```

We use the following utility function to pretty-print the WFST for us.

```python
>>> def display(wfst, tokens):
...     numfields = len(wfst)
...     hline = " " + "=" * 5 * len(tokens)
...     print hline
...     print \"| \" + \"=".join([\"%4d\" % i) for i in range(1, numfields)])
...     print hline
...     for i in range(numfields-1):
...         rownum = i
...         print \"%d\" % rownum,
...         for j in range(1, numfields):
...             print \"%4s\" % wfst[i][j],
...         print \"|\"
...     print hline
```

We create the initialized WFST and display it:

```python
>>> wfst0 = init_wfst(tokens, grammar)
>>> display(wfst0, tokens)
```

![Figure 8.2: Slice Points in the Input String](image)

Bird, Klein & Loper

February 1, 2007
As promised, there is a $V$ in cell (2, 3). The same information can be represented in a directed acyclic graph, as shown in Figure 8.15. This graph is usually called a chart.

![Figure 8.3: A Graph Representation of the Initialized WFST](image)

Returning to our tabular representation, given that we have $\text{DET}$ in cell (0, 1), and $\text{N}$ in cell (1, 2), what should we put into cell (0, 2)? In other words, what syntactic category derives the string the kids? We have already established that $\text{DET}$ derives the and $\text{N}$ derives kids, so we need to find a rule of the form $A \rightarrow \text{DET} \text{N}$, that is, a rule whose righthand side matches the categories in the cells we have already found. To help us easily retrieve production rules by their righthand sides, we create an index for the grammar.

```python
>>> index = {}
>>> for prod in grammar.productions():
...     index[prod.rhs()] = prod.lhs()
>>> r = grammar.productions()[2].rhs()
>>> r
(<Det>, <N>)
>>> index[r]
<NP>
```

This is an example of a space-time trade-off: we do a reverse lookup on the grammar, instead of having to check through entire list of productions each time we want to look up via the right-hand side. In the case at hand, we know that we can enter $\text{N}$ in cell (0,2). Figure 8.16 is the corresponding graph representation, where we add a new edge labeled $\text{N}$ to cover the input from 0 to 2.

![Figure 8.4: Adding an NP Edge to the Chart](image)

More generally, given an input $a_1 a_2 \cdots a_n$, we need to decide what to enter into cell $(i, j)$, where $0 \leq i < j \leq n$. We can enter $A$ in $(i, j)$ if we find nonterminals $B$, $C$, and an integer $k$ such that:
8. Advanced Parsing

Introduction to Natural Language Processing (DRAFT)

- $i < k < j$;
- $B$ is in cell $(i, k)$;
- $C$ is in cell $(k, j)$; and
- $A \rightarrow BC$ is a production of the grammar.

The following function uses this inference step to complete the WFST.

```python
>>> def complete_wfst(wfst, tokens, index, trace=False):
...     numtokens = len(tokens)
...     for span in range(2, numtokens+1):
...         for start in range(numtokens+1-span):
...             end = start + span
...             for mid in range(start+1, end):
...                 nt1 = wfst[start][mid]
...                 nt2 = wfst[mid][end]
...                 if (nt1,nt2) in index:
...                     nt3 = index[(nt1,nt2)]
...                     if trace:
...                         print "[%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s]" % \
...                             (start, nt1, mid, nt2, end, start, nt3, end)
...                     wfst[start][end] = index[(nt1,nt2)]
...     return wfst
...
```

```python
>>> wfst1 = complete_wfst(wfst0, tokens, index)
>>> display(wfst1, tokens)
```

This produces the following table:

```
1 2 3 4 5 6 7 8
-------------------------------
0| Det NP S S |
1| N |
2| V VP VP |
3| Det NP NP |
4| N |
5| P PP |
6| Det NP |
7| N |
-------------------------------
```

The tabular algorithm above gives an intuitive idea of why parsing context-free grammar is proportional to $n^3$: there is a triply nested loop, and the size of each loop is based on the length of the input.

By setting `trace` to `True` when calling the function `complete_wfst()`, we get additional output that is intended to be reminiscent of a chart representation:

```
```
For example, this says that since we found Det at \texttt{wfst[0][1]} and N at \texttt{wfst[1][2]}, we can add NP to \texttt{wfst[0][2]}.

We conclude that there is a parse for the whole input string once we have constructed an S node that covers the whole input, from position 0 to position 8; i.e., we can conclude that $S \Rightarrow^{*} a_1 a_2 \cdots a_n$.

A WFST is a data structure that can be used by a variety of parsing algorithms. The particular method for constructing a WFST that we have just presented is based on the CYK algorithm [ref?]. As you can see, the WFST is not itself a parse tree, so the technique is strictly speaking recognizing:dt that a sentence is admitted by a grammar, rather than parsing it. One significant limitation of CYK is that it requires every non-lexical grammar production to be binary. A grammar where this holds is said to be in Chomsky Normal Form. Although it is possible to convert an arbitrary CFG into Chomsky Normal Form (cf. [JurafskyMartin]), we would prefer to use an approach without such a requirement.

### 8.3.2 Exercises

1. Read about string edit distance and the Levenshtein Algorithm. Try the implementation provided in \texttt{nltk_lite.utilities.edit_dist}. How is this using dynamic programming? Does it use the bottom-up or top-down approach?

2. Modify the functions \texttt{init_wfst()} and \texttt{complete_wfst()} so that the contents of each cell in the WFST is a set of non-terminal symbols rather than a single non-terminal. \textit{Hint}. Something about avoiding deep copies.

3. Modify the functions \texttt{init_wfst()} and \texttt{complete_wfst()} so that when a non-terminal symbol is added to a cell in the WFST, it includes a record of the cells from which it was derived. Implement a function which will convert a WFST in this form to a parse tree.

### 8.3.3 Active Charts

One important aspect of the tabular approach to parsing can be seen more clearly if we look at the graph representation: given our grammar, there are two different ways to derive a top-level VP for the input, as shown in Figures 8.17 and 8.18.

![Figure 8.5: VP → V NP](image)

For the sake of simplicity, we did not try to show this structural ambiguity in our matrix representation. Nevertheless, it is possible to extend the approach so that each cell contains a set of non-terminals rather
than a single one. In our graph representation, we simply merge the two sets of edges in Figures 8.17 and 8.18 to yield Figure 8.19 (which also contains the final S edge).

However, we cannot necessarily read off from the WFST what the justification was for adding a specific edge. For example, in 8.18, the longest VP edge might be interpreted as having arisen by virtue of a rule VP → V NP PP. Unlike phrase structure trees, a WFST does not encode a relation of immediate dominance. In order to make such information available, we can label edges not just with a non-terminal category, but with the whole production which justified the addition of the edge. This is illustrated in Figure 8.20.

In general, a parser hypothesizes constituents based on the grammar and its current knowledge about the tokens it has seen and the constituents it has already found. Any constituent that is consistent with the current knowledge can be hypothesized; but many of these hypothesized constituents may not be used in complete parses. We can view a WFST as recording these hypotheses.

All of the edges that we've seen so far represent complete constituents. However, it can also be helpful to hypothesize incomplete constituents. For example, much of the work done by a parser in processing the production VP → V NP is also useful when processing VP → V NP PP. Thus, we might want to record the hypothesis that “the V constituent likes forms the beginning of a VP.”

We can record hypotheses of this form by adding a dot to the edge’s right hand side. The children to the left of the dot specify what children the constituent starts with; and the children to the right of the dot specify what children still need to be found in order to form a complete constituent. For example, the edge in the following chart records the hypothesis that “a VP starts with the V likes, but still needs an NP to become complete”:

These dotted edges are used to record all of the hypotheses that a chart parser makes about constituents in a sentence. Formally, we can define a dotted edge as follows:

A dotted edge \[ A \rightarrow c_1 \ldots c_d \cdot c_{d+1} \ldots c_n, (i,j) \] records the hypothesis that a constituent of type \( A \) starts with children \( c_1 \ldots c_d \) covering words \( w_i \ldots w_j \), but still needs children \( c_{d+1} \ldots c_n \) to be complete (where both \( c_1 \ldots c_d \) and \( c_{d+1} \ldots c_n \) may be empty.)

If \( d = n \) (i.e., if \( c_{d+1} \ldots c_n \) is empty) then the edge represents a complete constituent, and is called a complete edge. Otherwise, the edge represents an incomplete constituent, and is called an incomplete
8.3. Chart Parsing

**edge.** In the following chart, $[\text{VP} \rightarrow \text{V NP } \cdot, (1, 3)]$ is a complete edge, and $[\text{VP} \rightarrow \cdot \text{ NP}, (1, 2)]$ is an incomplete edge.

If $n = 0$ (i.e., if $c_1 \ldots c_n$ is empty), then the edge is called a **self-loop edge**. In the following chart, $[\text{VP} \rightarrow \cdot \text{ V NP}, (1, 1)]$ is a self-loop edge.

If a complete edge spans the entire sentence, and has the grammar’s start symbol as its left-hand side, then the edge is called a **parse edge**, and it encodes one or more parse trees for the sentence. In the following chart, $[S \rightarrow \text{ NP VP } \cdot, (0, 3)]$ is a parse edge.

### 8.3.4 The Chart Parser

To parse a sentence, a chart parser first creates an empty chart spanning the sentence. It then finds edges that are licensed by its knowledge about the sentence, and adds them to the chart one at a time until one or more parse edges are found. The edges that it adds can be licensed in one of three ways:

1. The *sentence* can license an edge. In particular, each word $w_i$ in the sentence licenses the complete edge $[w_i \rightarrow \cdot, (i, i+1)]$. 
2. The grammar can license an edge. In particular, each grammar production $A \to \alpha$ licenses the self-loop edge $[A \to \bullet \alpha, (i, i)]$ for every $i$, $0 \leq i < n$.

3. The current chart contents can license an edge.

However, it is not wise to add all licensed edges to the chart, since many of them will not be used in any complete parse. For example, even though the edge in the following chart is licensed (by the grammar), it will never be used in a complete parse:

Chart parsers therefore use a set of rules to heuristically decide when an edge should be added to a chart. This set of rules, along with a specification of when they should be applied, forms a strategy.

### 8.3.5 The Fundamental Rule

One rule is particularly important, since it is used by every chart parser: the Fundamental Rule. This rule is used to combine an incomplete edge that’s expecting a nonterminal $B$ with a complete edge immediately following it whose left hand side is $B$. Formally, it states that if the chart contains the edges:

1. $[A \to \alpha \bullet B \beta, (i, j)]$
2. $[B \to \gamma \bullet, (j, k)]$

Then the parser should add the edge:

3. $[A \to \alpha B \bullet \beta, (i, k)]$
8.3.6 Bottom Up Parsing

To create a bottom-up parser, we need to add two rules: the **Bottom-Up Initialization Rule**; and the **Bottom-Up Predict Rule**.

The Bottom-Up Initialization Rule says to add all edges licensed by the sentence. In particular, it states that for every word $w_i$, the parser should add the edge:

1. $[w_i \rightarrow \bullet, (i, i+1)]$

The Bottom-Up Predict Rule says that if the chart contains a complete edge, then the parser should add a self-loop edge at the complete edge’s left boundary for each grammar production whose right-hand side begins with the completed edge’s left-hand side. In other words, it states that if the chart contains the complete edge:

1. $[A \rightarrow \alpha \bullet, (i, j)]$

And the grammar contains the production:

2. $B \rightarrow A \beta$

Then the parser should add the self-loop edge:

3. $[B \rightarrow \bullet \beta, (i, i)]$

Using these three rules, we can parse a sentence as follows:

1. Create an empty chart spanning the sentence.

2. Apply the Bottom-Up Initialization Rule to each word.

3. Until no more edges are added: a) Apply the Bottom-Up Predict Rule everywhere it applies. #) Apply the Fundamental Rule everywhere it applies.

4. Return all of the parse trees corresponding to the parse edges in the chart.

For example, Figure 8.21 shows the order in which edges get added to the chart when applying bottom-up parsing to a simple example sentence:
Figure 8.9: Bottom-Up Chart Parsing
8.3.7 Top-Down Parsing

To create a bottom-up parser, we need to use the Fundamental Rule plus three other rules: the Top-Down Initialization Rule, the Top-Down Expand Rule, and the Top-Down Match Rule.

The top-down initialization rule captures the fact that root of any parse must be the start symbol. It states that for every grammar production:

1. \( S \rightarrow \alpha \)

   The parser should add the self-loop edge:
   
   \[ S \rightarrow \bullet \alpha \]

2. \([S \rightarrow \bullet \alpha, (0:0)]\)

   The top-down expand rule says that if the chart contains an incomplete edge whose dot is followed by a nonterminal \( B \), then the parser should add any self-loop edges licensed by the grammar whose left-hand side is \( B \). In particular, if the chart contains the incomplete edge:

   \[ A \rightarrow \alpha \bullet B \beta, (i, j) \]

   Then for each grammar production:

2. \( B \rightarrow \gamma \)

   The parser should add the edge:

   \[ B \rightarrow \bullet \gamma \]

3. \([B \rightarrow \bullet \gamma, (j, j)]\)

   The top-down match rule says that if the chart contains an incomplete edge whose dot is followed by a terminal \( w \), then the parser should add an edge if the terminal corresponds to the text. In particular, if the chart contains the incomplete edge:

   \[ A \rightarrow \alpha \bullet w[j] \beta \]

1. \([A \rightarrow \alpha \bullet w[j] \beta, (i, j)]\)
Then the parser should add the complete edge:

\[ w[j] \rightarrow \bullet \]

2. \[ w_j \rightarrow \bullet , (j, j+1) \]

Using these four rules, we can parse a sentence as follows:

1. Create an empty chart spanning the sentence.
2. Apply the Top-Down Initialization Rule to each word.
3. Until no more edges are added: a) Apply the Top-Down Expand Rule everywhere it applies. b) Apply the Top-Down Match Rule everywhere it applies. c) Apply the Fundamental Rule everywhere it applies.
4. Return all of the parse trees corresponding to the parse edges in the chart.

### 8.3.8 Chart Parsing in NLTK-Lite

`nltk_lite.parse.chart` defines a simple yet flexible chart parser, `ChartParse`. A new chart parser is constructed from a grammar and a list of chart rules (also known as a strategy). These rules will be applied, on order, until no new edges are added to the chart. In particular, `ChartParse` uses the following algorithm:

Until no new edges are added:

- For each chart rule \( R \):
  - Apply \( R \) to any applicable edges in the chart.

Return any complete parses in the chart.

`nltk_lite.parse.chart` defines two pre-made strategies: `TD_STRATEGY`, a basic top-down strategy; and `BU_STRATEGY`, a basic bottom-up strategy. When constructing a chart parser, you can use either of these strategies, or create your own.

The following example illustrates the use of the chart parser. We start by defining a simple grammar:

```python
>>> grammar = cfg.parse_grammar('''
... S -> NP VP
... VP -> V NP | VP PP
... V -> "saw" | "ate"
... NP -> "John" | "Mary" | "Bob" | Det N | NP PP
... Det -> "a" | "an" | "the" | "my"
... N -> "dog" | "cat" | "cookie"
... PP -> P NP
... P -> "on" | "by" | "with"
... ''')
```

Next we tokenize a sentence. We make sure it is a list (not an iterator), since we wish to use the same tokenized sentence several times.
8.3. Chart Parsing

>>> sent = list(tokenize.whitespace('John saw a cat with my cookie'))
>>> parser = ChartParse(grammar, BU_STRATEGY)
>>> for tree in parser.get_parse_list(sent):
    ...    print tree
(S:
  (NP: 'John')
  (VP:
    (VP: (V: 'saw') (NP: (Det: 'a') (N: 'cat')))
    (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie')))))
(S:
  (NP: 'John')
  (VP:
    (NP: (Det: 'a') (N: 'cat'))
    (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie')))))

The `trace` parameter can be specified when creating a parser, to turn on tracing (higher trace levels produce more verbose output). The following examples show the trace output for parsing a sentence with the bottom-up strategy:

# Parse the sentence, bottom-up, with tracing turned on.
>>> parser = ChartParse(grammar, BU_STRATEGY, trace=2)
>>> parser.get_parse(sent)
Bottom Up Init Rule:
| [-----] . . . . . . | [0:1] 'John'
| [-----] . . . . . . | [1:2] 'saw'
| . [-----] . . . . . | [2:3] 'a'
| . . [-----] . . . . | [3:4] 'cat'
| . . . [-----] . . . | [4:5] 'with'
| . . . . [-----] . . | [5:6] 'my'
| . . . . . [-----] . | [6:7] 'cookie'
Bottom Up Predict Rule:
|> . . . . . . . | [0:0] NP -> * 'John'
|> . . . . . . . | [1:1] V -> * 'saw'
|. . > . . . . . | [2:2] Det -> * 'a'
|. . . > . . . . | [3:3] N -> * 'cat'
|. . . . > . . . | [4:4] P -> * 'with'
|. . . . . > . . | [5:5] Det -> * 'my'
|. . . . . . > . | [6:6] N -> * 'cookie'
Fundamental Rule:
| [-----] . . . . . . | [0:1] NP -> 'John' *
| [-----] . . . . . . | [1:2] V -> 'saw' *
| . [-----] . . . . . | [2:3] Det -> 'a' *
| . . [-----] . . . . | [3:4] N -> 'cat' *
| . . . [-----] . . . | [4:5] P -> 'with' *
| . . . . [-----] . . | [5:6] Det -> 'my' *
| . . . . . [-----] | [6:7] N -> 'cookie' *
Bottom Up Predict Rule:
|> . . . . . . . | [0:0] S -> * NP VP
|> . . . . . . . | [0:0] NP -> * NP PP
|. . > . . . . . | [1:1] VP -> * V NP
8. Advanced Parsing

Introduction to Natural Language Processing (DRAFT)

| . . > . . . . | [2:2] NP -> * Det N
| . . . > . . . | [4:4] PP -> * P NP
| . . . > . . . | [5:5] NP -> * Det N

Fundamental Rule:

| [------> . . . . . . | [0:1] S -> NP * VP
| [------> . . . . . . | [0:1] NP -> NP * PP
| . . [------> . . . . | [1:2] VP -> V * NP
| . . [------> . . . . | [2:3] NP -> Det * N
| . [------> . . . . | [2:4] NP -> Det N *
| . . . [------> . . . | [4:5] PP -> P * NP
| . . [------> . . . . | [5:6] NP -> Det N *
| . [-----------------] . . | [1:4] VP -> V NP *
| . . . [-----------------] | [4:7] PP -> P NP *

Bottom Up Predict Rule:

| . . > . . . . . | [2:2] S -> * NP VP
| . . > . . . . . | [2:2] NP -> * NP PP
| . . . > . . . | [5:5] S -> * NP VP
| . . . > . . . | [5:5] NP -> * NP PP
| . > . . . . . | [1:1] VP -> * VP PP

Fundamental Rule:

| . [-------------------> . . . | [2:4] S -> NP * VP
| . [-------------------> . . . | [2:4] NP -> NP * PP
| . [-------------------> . . . | [5:7] S -> NP * VP
| . [-------------------> . . . | [5:7] NP -> NP * PP
| . [-------------------> . . . | [1:4] VP -> VP * PP
| . [-------------------> . . . | [2:7] NP -> VP PP *
| . [-------------------> . . . | [1:7] VP -> VP PP *
| . [-------------------> . . . | [2:7] NP -> NP PP *
| . [-------------------> . . . | [1:7] VP -> VP PP *

(S: (NP: ‘John’) (VP: (VP: (V: ‘saw’) (NP: (Det: ‘a’) (N: ‘cat’)))
(PP: (P: ‘with’) (NP: (Det: ‘my’) (N: ‘cookie’))))

A more interactive interface to the chart-parser can be found in the graphical demo:

```python
>>> from nltk_lite.draw.chart import demo
>>> demo()
```

8.3.9 Exercises

1. Use the graphical chart-parser interface to experiment with different rule invocation strategies. Come up with your own strategy which you can execute manually using the graphical interface. Describe the steps, and report any efficiency improvements it has (e.g. in terms of the size of the resulting chart). Do these improvements depend on the structure of the grammar? What do you think of the prospects for significant performance boosts from cleverer rule invocation strategies?
8.4 Probabilistic Parsing

As we pointed out in the introduction to this chapter, dealing with ambiguity is a key challenge to broad coverage parsers. We have shown how chart parsing can help improve the efficiency of computing multiple parses of the same sentences. But the sheer number of parses can be just overwhelming. We will show how probabilistic parsing helps to manage a large space of parses. However, before we deal with these parsing issues, we must first back up and introduce weighted grammars.

8.4.1 Weighted Grammars

We begin by considering the verb give. This verb requires both a direct object (the thing being given) and an indirect object (the recipient). These complements can be given in either order, as illustrated in example (59). In the “prepositional dative” form, the indirect object appears last, and inside a prepositional phrase, while in the “double object” form, the indirect object comes first:

(59a) Kim gave a bone to the dog

(59b) Kim gave the dog a bone

Using the Penn Treebank sample, we can examine all instances of prepositional dative and double object constructions involving give, as shown below:

```python
>>> from nltk_lite.corpora import treebank
>>> from string import join

>>> give = lambda t: t.node == 'VP' and len(t) > 2 and t[1].node == 'NP'
...    and (t[2].node == 'PP-DTV' or t[2].node == 'NP')
...    and ('give' in t[0].leaves() or 'gave' in t[0].leaves())

>>> for tree in treebank.parsed():
...    for t in tree.subtrees(give):
...        print "%s [\%s: %s] [\%s: %s] [\%s: %s]\n" %
...            (join(t[0].leaves()),
...             t[1].node, join(t[1].leaves()),
...             t[2].node, join(t[2].leaves()))

gave [NP: the chefs] [NP: a standing ovation]
give [NP: advertisers] [NP: discounts for * maintaining or increasing ad spending]
give [NP: it] [PP-DTV: to the politicians]
give [NP: them] [NP: similar help]
give [NP: them] [NP: *T*-1]
give [NP: only French history questions] [PP-DTV: to students in a European history]
give [NP: federal judges] [NP: a raise]
give [NP: consumers] [NP: the straight scoop on the U.S. waste crisis]
give [NP: Mitsui] [NP: access to a high-tech medical product]
give [NP: Mitsubishi] [NP: a window on the U.S. glass industry]
give [NP: much thought] [PP-DTV: to the rates 0 she was receiving *T*-2 , nor to the fees 0 she was paying *T*-3]
give [NP: your Foster Savings Institution] [NP: the gift of hope and freedom from the regulators who *T*-206 want *-1 to close its doors -- for good]
give [NP: market operators] [NP: the authority * to suspend trading in futures at any time]
give [NP: quick approval] [PP-DTV: to $ 3.18 billion *U* in supplemental appropriative funding for law enforcement and anti-drug programs in fiscal 1990]
give [NP: the Transportation Department] [NP: up to 50 days 0 * to review any purchase *T*-1]
```
give [NP: the president] [NP: such power]  
give [NP: me] [NP: the heebie-jeebies]  
give [NP: holders] [NP: the right *RNR*-1, but not the obligation *RNR*-1, * to buy -LRB- a call -RRB- or sell -LRB- a put -RRB- a specified amount of an underlying investment by a certain date at a preset price, known * as the strike price]  
gave [NP: Mr. Thomas] [NP: only a "' qualified '" rating, rather than "' well qualified]  
give [NP: the president] [NP: line-item veto power]

We can observe a strong tendency for the shortest complement to appear first. However, this does not account for a form like give [NP: federal judges] [NP: a raise], where animacy may be playing a role. In fact there turn out to be a large number of contributing factors, as surveyed by Bresnan and Hay (2006).

How can such tendencies be expressed in a conventional context free grammar? It turns out that they cannot. However, we can address the problem by adding weights, or probabilities, to the productions of a grammar.

A probabilistic context free grammar (or PCFG) is a context free grammar that associates a probability with each of its productions. It generates the same set of parses for a text that the corresponding context free grammar does, and assigns a probability to each parse. The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it.

Probabilistic context free grammars are implemented by the nltk_lite.parse.pcfg.WeightedGrammar class. Like CFGs, each PCFG consists of a start state and a list of productions. But the productions are represented by pcfg.WeightedProduction, a subclass of cfg.Production that associates a probability with a context free grammar production.

### 8.4.2 PCFG Productions

Each PCFG production specifies that a nonterminal (the left-hand side) can be expanded to a sequence of terminals and nonterminals (the right-hand side). In addition, each production has a probability associated with it. Productions are created using the nltk_lite.parse.pcfg.WeightedProduction constructor, which takes a probability, a nonterminal left-hand side, and zero or more terminals and nonterminals for the right-hand side.

```python
>>> from nltk_lite.parse import cfg  
>>> S, VP, V, NP = cfg.nonterminals(‘S, VP, V, NP’)  

>>> from nltk_lite.parse import pcfg  
>>> prod1 = pcfg.WeightedProduction(VP, [V, NP], prob=0.23)  
>>> prod1  
   VP -> V NP (p=0.23) 

>>> prod2 = pcfg.WeightedProduction(V, ['saw'], prob=0.12)  
>>> prod2  
   V -> 'saw' (p=0.12) 

>>> prod3 = pcfg.WeightedProduction(NP, ['cookie'], prob=0.04)  
>>> prod3  
   NP -> 'cookie' (p=0.04) 
```

The probability associated with a production is returned by the prob method:
As with CFG productions, the left-hand side of a PCFG production is returned by the `lhs` method; and the right-hand side is returned by the `rhs` method:

```python
>>> prod1.lhs()
<V>
>>> prod1.rhs()
(<V>, <NP>)
```

### 8.4.3 PCFGs

PCFGs are created using the `pcfg.WeightedGrammar` constructor, which takes a start symbol and a list of productions:

```python
prods = [pcfg.WeightedProduction(S, [NP, VP], prob=1.0),
... pcfg.WeightedProduction(VP, ['saw', NP], prob=0.4),
... pcfg.WeightedProduction(VP, ['ate'], prob=0.3),
... pcfg.WeightedProduction(VP, ['gave', NP, NP], prob=0.3),
... pcfg.WeightedProduction(NP, ['the', 'cookie'], prob=0.8),
... pcfg.WeightedProduction(NP, ['Jack'], prob=0.2)]
```

```python
>>> grammar = pcfg.WeightedGrammar(S, prods)
>>> print grammar
Grammar with 6 productions (start state = S)
  S -> NP VP (p=1.0)
  VP -> 'saw' NP (p=0.4)
  VP -> 'ate' (p=0.3)
  VP -> 'gave' NP NP (p=0.3)
  NP -> 'the' 'cookie' (p=0.8)
  NP -> 'Jack' (p=0.2)
```

In order to ensure that the trees generated by the grammar form a proper probability distribution, PCFG grammars impose the constraint that all productions with a given left-hand side must have probabilities that sum to one:

\[
\text{for all } \text{lhs}: \quad \sum_{\text{rhs}} P(\text{lhs} \rightarrow \text{rhs}) = 1
\]

The example grammar given above obeys this constraint: for `S`, there is only one production, with a probability of 1.0; for `VP`, 0.4+0.3+0.3=1.0; and for `NP`, 0.8+0.2=1.0.

As with CFGs, the start state of a PCFG is returned by the `start` method; and the productions are returned by the `productions` method:

```python
>>> grammar.start()
<S>
>>> from pprint import pprint
>>> pprint(grammar.productions())
(S -> NP VP (p=1.0),
  VP -> 'saw' NP (p=0.4),
  VP -> 'ate' (p=0.3),
  VP -> 'gave' NP NP (p=0.3),
  NP -> 'the' 'cookie' (p=0.8),
  NP -> 'Jack' (p=0.2))
```
8.5 Probabilistic Parsers

8.5.1 The Probabilistic Parser Interface

The parse trees returned by `parse` and `get_parse_list` include probabilities:

```python
>>> from nltk_lite.parse import ViterbiParse
>>> viterbi_parser = ViterbiParse(grammar)
>>> sent = list(tokenize.whitespace('Jack saw the cookie'))
>>> viterbi_parser.get_parse(sent)
(S: (NP: 'Jack') (p=0.2) (VP: 'saw' (NP: 'the' 'cookie') (p=0.8)) (p=0.32)) (p=0.064)
```

```python
>>> viterbi_parser.get_parse_list(sent)
[(S: (NP: 'Jack') (p=0.2) (VP: 'saw' (NP: 'the' 'cookie') (p=0.8)) (p=0.32)) (p=0.064)]
```

8.5.2 Probabilistic Parser Implementations

The next two sections introduce two probabilistic parsing algorithms for PCFGs. The first is a Viterbi-style algorithm that uses dynamic programming to find the single most likely parse for a given text. Whenever it finds multiple possible parses for a subtree, it discards all but the most likely parse. The second is a bottom-up chart parser that maintains a queue of edges, and adds them to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, allowing the parser to expand more likely edges before less likely ones. Different queue orderings are used to implement a variety of different search strategies. These algorithms are implemented in the `nltk_lite.parse.viterbi` and `nltk_lite.parse.pchart` modules.

8.5.3 A Viterbi-Style PCFG Parser

The `ViterbiParse` PCFG parser is a bottom-up parser that uses dynamic programming to find the single most likely parse for a text. It parses texts by iteratively filling in a `most likely constituents table`. This table records the most likely tree structure for each span and node value. In particular, it has an entry for every start index, end index, and node value, recording the most likely subtree that spans from the start index to the end index, and has the given node value. For example, after parsing the sentence “I saw John with my cookie” with a simple grammar, the most likely constituents table might be as follows:

<table>
<thead>
<tr>
<th>Span</th>
<th>Node</th>
<th>Tree</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0:1]</td>
<td>NP</td>
<td>(NP: I)</td>
<td>0.3</td>
</tr>
<tr>
<td>[2:3]</td>
<td>NP</td>
<td>(NP: John)</td>
<td>0.3</td>
</tr>
<tr>
<td>[4:6]</td>
<td>NP</td>
<td>(NP: my cookie)</td>
<td>0.2</td>
</tr>
<tr>
<td>[3:6]</td>
<td>PP</td>
<td>(PP: with (NP: my cookie))</td>
<td>0.1</td>
</tr>
<tr>
<td>[2:6]</td>
<td>NP</td>
<td>(NP: (NP: John) (PP: with (NP: my cookie)))</td>
<td>0.01</td>
</tr>
<tr>
<td>[1:3]</td>
<td>VP</td>
<td>(VP: saw (NP: John))</td>
<td>0.03</td>
</tr>
<tr>
<td>[1:6]</td>
<td>VP</td>
<td>(VP: saw (NP: (NP: John) (PP: with (NP: my cookie))))</td>
<td>0.001</td>
</tr>
<tr>
<td>[0:6]</td>
<td>S</td>
<td>(S: (NP: I) (VP: saw (NP: (NP: John) (PP: with (NP: my cookie)))))</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

Table 8.2:
Once the table has been completely filled in, the parser simply returns the entry for the most likely constituent that spans the entire text, and whose node value is the start symbol. For this example, it would return the entry with a span of [0:6] and a node value of “S”.

Note that we only record the most likely constituent for any given span and node value. For example, in the table above, there are actually two possible constituents that cover the span [1:6] and have “VP” node values.

1. “saw John, who has my cookie”:
   \[(VP: \text{saw} (NP: (NP: John) (PP: with (NP: my cookie))))\]

2. “used my cookie to see John”:
   \[(VP: \text{saw} (NP: John) (PP: with (NP: my cookie)))\]

Since the grammar we are using to parse the text indicates that the first of these tree structures has a higher probability, the parser discards the second one.

**Filling the Most Likely Constituents Table:** Because the grammar used by ViterbiParse is a PCFG, the probability of each constituent can be calculated from the probabilities of its children. Since a constituent’s children can never cover a larger span than the constituent itself, each entry of the most likely constituents table depends only on entries for constituents with shorter spans (or equal spans, in the case of unary and epsilon productions).

ViterbiParse takes advantage of this fact, and fills in the most likely constituent table incrementally. It starts by filling in the entries for all constituents that span a single element of text. After it has filled in all the table entries for constituents that span one element of text, it fills in the entries for constituents that span two elements of text. It continues filling in the entries for constituents spanning larger and larger portions of the text, until the entire table has been filled.

To find the most likely constituent with a given span and node value, ViterbiParse considers all productions that could produce that node value. For each production, it checks the most likely constituents table for sequences of children that collectively cover the span and that have the node values specified by the production’s right hand side. If the tree formed by applying the production to the children has a higher probability than the current table entry, then it updates the most likely constituents table with the new tree.

**Handling Unary Productions and Epsilon Productions:** A minor difficulty is introduced by unary productions and epsilon productions: an entry of the most likely constituents table might depend on another entry with the same span. For example, if the grammar contains the production \(V \rightarrow VP\), then the table entries for \(VP\) depend on the entries for \(V\) with the same span. This can be a problem if the constituents are checked in the wrong order. For example, if the parser tries to find the most likely constituent for a \(VP\) spanning [1:3] before it finds the most likely constituents for \(V\) spanning [1:3], then it can’t apply the \(V \rightarrow VP\) production.

To solve this problem, ViterbiParse repeatedly checks each span until it finds no new table entries. Note that cyclic grammar productions (e.g. \(V \rightarrow V\)) will not cause this procedure to enter an infinite loop. Since all production grammar probabilities are less than or equal to 1, any constituent generated by a cycle in the grammar will have a probability that is less than or equal to the original constituent; so ViterbiParse will discard it.
8.5.4 Using ViterbiParser

Viterbi parsers are created using the ViterbiParser constructor:

```python
>>> from nltk_lite.parse.viterbi import *
>>> ViterbiParser(grammar)
<ViterbiParser for <Grammar with 6 productions>>
```

Note that since ViterbiParser only finds the single most likely parse, that get_parse_list will never return more than one parse.

```python
>>> viterbi_parser1 = ViterbiParser(pcfg.toy1)
>>> sent1 = list(tokenize.whitespace('I saw John with my cookie'))
>>> tree1 = viterbi_parser1.parse(sent1)
>>> print tree1
(S:
  (NP: 'I')
  (VP:
    (V: 'saw')
    (NP:
      (NP: 'John')
      (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie')))))) (p=5.2040625e-05)
```

```python
>>> viterbi_parser2 = ViterbiParser(pcfg.toy2)
>>> sent2 = list(tokenize.whitespace('the boy saw Jack with Bob under the table with a telescope'))
>>> trees = viterbi_parser2.get_parse_list(sent2)
>>> for tree in trees:
...    print tree
(S:
  (NP: (Det: 'the') (N: 'boy'))
  (VP:
    (V: 'saw')
    (NP:
      (NP: (Name: 'Jack'))
      (PP: (P: 'with')
        (NP:
          (NP: (Name: 'Bob'))
          (PP: (P: 'under')
            (NP: (Det: 'the') (N: 'table')))
          (PP: (P: 'with')
            (NP: (Det: 'a') (N: 'telescope')))))) (p=7.53678903935e-11)
```

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays the constituents that are considered, and indicates which ones are added to the most likely constituent table. It also indicates the likelihood for each constituent.

```python
>>> viterbi_parser1.trace(3)
>>> tree = viterbi_parser1.parse(sent1)
Inserting tokens into the most likely constituents table...
```
8.6. A Bottom-Up PCFG Chart Parser

8.6.1 Introduction

The Viterbi-style algorithm described in the previous section finds the single most likely parse for a given text. But for many applications, it is useful to produce several alternative parses. This is often the case when probabilistic parsers are combined with other probabilistic systems. In particular, the most probable parse may be assigned a low probability by other systems; and a parse that is given a low probability by the parser might have a better overall probability.

For example, a probabilistic parser might decide that the most likely parse for “I saw John with the cookie” is is the structure with the interpretation “I used my cookie to see John”; but that parse would be assigned a low probability by a semantic system. Combining the probability estimates from the parser and the semantic system, the parse with the interpretation “I saw John, who had my cookie” would be given a higher overall probability.

This section describes BottomUpChartParser, a parser for PCFGs that can find multiple parses for a text. It assumes that you have already read the chart parsing tutorial, and are familiar with the data structures and productions used for chart parsing.

The level of tracing output can also be set with an optional argument to the ViterbiParse constructor. By default, no tracing output is generated. Tracing output can be turned off by calling trace with a value of 0.
8.6.2  The Basic Algorithm

`BottomUpChartParser` is a bottom-up parser for PCFGs that uses a `Chart` to record partial results. It maintains a queue of edges, and adds them to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, allowing the parser to insert more likely edges before exploring less likely ones. For each edge that the parser adds to the chart, it may become possible to insert new edges into the chart; these are added to the queue. `BottomUpChartParser` continues adding the edges in the queue to the chart until enough complete parses have been found, or until the edge queue is empty.

8.6.3  Probabilistic Edges

An `Edge` associates a dotted production and a location with a (partial) parse tree. A `probabilistic edge` can be formed by using a `ProbabilisticTree` to encode an edge’s parse tree. The probability of this tree is the product of the probability of the production that generated it and the probabilities of its children. For example, the probability associated with an edge `Edge: S \rightarrow NP \cdot VP, (0:2)` is the probability of its NP child times the probability of the PCFG production `S \rightarrow NP \cdot VP`. Note that an edge’s tree only includes children for elements to the left of the edge’s dot. Thus, the edge’s probability does *not* include any probabilities for the elements to the right of the edge’s dot.

8.6.4  The Edge Queue

The edge queue is a sorted list of edges that can be added to the chart. It is initialized with a single edge for each token in the text. These `token edges` have the form `[Edge: token \rightarrow \cdot]` where `token` is the word.

As each edge from the queue is added to the chart, it may become possible to insert new edges into the chart; these new edges are added to the queue. There are two ways that it can become possible to insert new edges into the chart:

1. The `bottom-up initialization production` can be used to add a self-loop edge whenever an edge whose dot is in position 0 is added to the chart.

2. The `fundamental production` can be used to combine a new edge with edges already present in the chart.

The edge queue is implemented using a `list`. For efficiency reasons, `BottomUpChartParser` uses `pop` to remove edges from the queue. Thus, the front of the queue is the `end` of the list. This needs to be kept in mind when implementing sorting orders for the queue: edges that should be tried first should be placed at the end of the list.

8.6.5 Sorting The Edge Queue

By changing the sorting order used by the queue, we can control the strategy that the parser uses to search for parses of a text. Since there are a wide variety of reasonable search strategies, `BottomUpChartParser` does not define the sorting order for the queue. Instead, `BottomUpPCFGChartParser` is defined as an abstract class; and subclasses are used to implement a variety of different queue orderings. Each subclass is required to define the `sort_queue` method, which sorts a given queue. The remainder of this section describes four different subclasses of `BottomUpChartParser` that are defined in the `nltk_lite.parse.pchart` module.
**InsideParse:**

The simplest way to order the queue is to sort the edges by the probabilities of their trees. This ordering concentrates the efforts of the parser on edges that are more likely to be correct descriptions of the texts that they span. This approach is implemented by the `InsideParse` class.

The probability of an edge’s tree provides an upper bound on the probability of any parse produced using that edge. The probabilistic “cost” of using an edge to form a parse is one minus its tree’s probability. Thus, inserting the edges with the most likely trees first results in a lowest-cost-first search strategy. Lowest-cost-first search is an optimal search strategy: the first solution it finds is guaranteed to be the best solution.

However, lowest-cost-first search can be rather inefficient. Since a tree’s probability is the product of the probabilities of all the productions used to generate it, smaller trees tend to have higher probabilities than larger ones. Thus, lowest-cost-first search tends to insert edges with small trees before moving on to edges with larger ones. But any complete parse of the text will necessarily have a large tree; so complete parses will tend to be inserted after nearly all other edges.

The basic problem with lowest-cost-first search is that it ignores the probability that an edge’s tree is part of a complete parse. It will try parses that are locally coherent, even if they are unlikely to form part of a complete parse. Unfortunately, it can be quite difficult to calculate the probability that a tree is part of a complete parse. However, we can use a variety of techniques to approximate that probability.

Since `InsideParse` is a subclass of `BottomUpChartParse`, it only needs to define a `sort_queue` method. Thus, the implementation of `InsideParse` class is quite simple:

```python
class InsideParse(BottomUpChartParse):
    def sort_queue(self, queue, chart):
        # Sort the edges by the probabilities of their trees.
        queue.sort(lambda e1, e2: cmp(e1.tree().prob(), e2.tree().prob()))
```

**LongestParse:** `LongestParse` sorts its queue in descending order of the edges’ lengths. These lengths (properly normalized) provide a crude approximations to the probabilities that trees are part of complete parses. Thus, `LongestParse` employs a best-first search strategy, where it inserts the edges that are closest to producing complete parses before trying any other edges. Best-first search is not an optimal search strategy: the first solution it finds is not guaranteed to be the best solution. However, it will usually find a complete parse much more quickly than lowest-cost-first search.

Since `LongestParse` is a subclass of `BottomUpChartParse`, its implementation simply defines a `sort_queue` method:

```python
class LongestParse(BottomUpChartParse):
    def sort_queue(self, queue, chart):
        # Sort the edges by the lengths of their trees.
        queue.sort(lambda e1, e2: cmp(len(e1.loc()), len(e2.loc())))
```

**BeamParse:** When large grammars are used to parse a text, the edge queue can grow quite long. The edges at the end of a large well-sorted queue are unlikely to be used. Therefore, it is reasonable to remove (or prune) these edges from the queue.

`BeamParse` provides a simple implementation of a pruning PCFG parser. It uses the same sorting order as `InsideParse`. But whenever the edge queue grows beyond a pre-defined maximum length, `BeamParse` truncates it. The resulting search strategy, lowest-cost-first search with pruning, is a type of beam search. (A beam search is a search strategy that only keeps the best partial results.) The queue’s predefined maximum length is called the beam size (or simply the beam). The parser’s beam size is set by the first argument to its constructor.
Beam search reduces the space requirements for lowest-cost-first search, by discarding edges that are not likely to be used. But beam search also loses many of lowest-cost-first search’s more useful properties. Beam search is not optimal: it is not guaranteed to find the best parse first. In fact, since it might prune a necessary edge, beam search is not even complete: it is not guaranteed to return a parse if one exists.

The implementation for BeamParse defines two methods. First, it overrides the constructor, since it needs to record the beam size. And second, it defines the sort_queue method, which sorts the queue and discards any excess edges:

```python
class BeamParse(BottomUpChartParse):
    def __init__(self, beam_size, grammar, trace=0):
        BottomUpChartParse.__init__(self, grammar, trace)
        self._beam_size = beam_size

    def sort_queue(self, queue, chart):
        # Sort the queue.
        queue.sort(lambda e1, e2: cmp(e1.tree().prob(), e2.tree().prob()))
        # Truncate the queue, if necessary.
        if len(queue) > self._beam_size:
            queue[:] = queue[len(queue) - self._beam_size:]
```

Note that when truncating the queue, sort_queue uses the expression queue[: ] to change the contents of the queue variable. In particular, compare it to the following code, which reassigns the local variable queue, but does not modify the contents of the given list:

```python
# WRONG: This does not change the contents of the edge queue.
if len(queue) > self._beam_size:
    queue = queue[len(queue) - self._beam_size:]

# WRONG: The sort method returns None.
return queue.sort(lambda e1, e2: cmp(e1.tree().prob(), e2.tree().prob()))
```

### 8.6.6 Using BottomUpChartParser

These parsers are created using the BottomUpChartParse subclasses’s constructors. These include: InsideParse, LongestParse, BeamParser, and RandomParse.

See the reference documentation for the BottomUpChartParse module for a complete list of subclasses. Unless a subclass overrides the constructor, it takes a single PCFG:

```python
>>> from nltk_lite.parse.pchart import *
>>> inside_parser = InsideParse(pcfg.toy1)
>>> longest_parser = LongestParse(pcfg.toy1)
>>> beam_parser = BeamParse(20, pcfg.toy1)

>>> print inside_parser.parse(sent1)
(S:
 (NP: ’I’)
 (VP:
   (V: ’saw’)
   (NP:
     (NP: ’John’)
     (PP: (P: ’with’) (NP: (Det: ’my’) (N: ’cookie’)))))) (p=5.2040625e-05)
```
8.6. A Bottom-Up PCFG Chart Parser

```python
>>> for tree in inside_parser.get_parse_list(sent1):
...     print tree
(S:
  (NP: 'I')
  (VP:
    (V: 'saw')
    (NP:
      (NP: 'John')
      (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie'))))))
(S: (NP: 'I') (VP: (V: 'saw') (NP: 'John')) (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie'))))

Warning

BottomUpChartParser is an abstract class; you should not directly instantiate it. If you try to use it to parse a text, it will raise an exception, since sort_queue will be undefined.

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays edges as they are added to the chart, and shows the probability for each edges’ tree.

```python
>>> inside_parser.trace(3)
>>> trees = inside_parser.get_parse_list(sent1)

| . . . . . [-] | [5:6] 'cookie' prob=1.0
| . . . [-] . | [4:5] 'my' prob=1.0
| . . [-] . . | [3:4] 'with' prob=1.0
| . . [-] . . | [2:3] 'John' prob=1.0
| . [-] . . . | [1:2] 'saw' prob=1.0
| [-] . . . . | [0:1] 'I' prob=1.0
| . [-] . . . | [1:2] V -> 'saw' * prob=0.65
| . > . . . . | [1:1] VP -> * V NP prob=0.7
| . > . . . . | [1:1] V -> * 'saw' prob=0.65
| . . [-] . . | [3:4] P -> 'with' * prob=0.61
| . . > . . . | [3:3] PP -> * P NP prob=1.0
| . . [-] . . | [3:4] PP -> P * NP prob=0.61
| . . > . . . | [3:3] P -> * 'with' prob=0.61
| . . . [-] . | [5:6] N -> 'cookie' * prob=0.5
| . . . . . . | [5:5] N -> * 'cookie' prob=0.5
| . [-] . . . | [1:2] VP -> V * NP prob=0.455
| . > . . . . | [1:1] VP -> * V prob=0.2
| . . . [-] . | [4:5] Det -> 'my' * prob=0.2
| . . . > . . | [4:4] NP -> * Det N prob=0.5
| . . . > . . | [4:4] Det -> * 'my' prob=0.2
| [-] . . . . | [0:1] NP -> 'I' * prob=0.15
| > . . . . . | [0:0] S -> * NP VP prob=1.0
| > . . . . . | [0:0] NP -> * NP PP prob=0.25
| [-] . . . . | [0:1] S -> NP * VP prob=0.15
| > . . . . . | [0:0] NP -> * 'I' prob=0.15
| . [-] . . . | [1:2] VP -> V * prob=0.13
8.7 Grammar Induction

As we have seen, PCFG productions are just like CFG productions, adorned with probabilities. So far, we have simply specified these probabilities in the grammar. However, it is more usual to estimate these probabilities from training data, namely a collection of parse trees or treebank.

The simplest method uses Maximum Likelihood Estimation, so called because probabilities are chosen in order to maximize the likelihood of the training data. The probability of a production \( VP \rightarrow V NP PP \) is \( p(V, NP, PP \mid VP) \). We calculate this as follows:

\[
P(V, NP, PP \mid VP) = \frac{\text{count}(VP \rightarrow V NP PP)}{\text{count}(VP \rightarrow \ldots)}
\]

Here is a simple program that induces a grammar from the first three parse trees in the Penn Treebank corpus:

```python
>>> from nltk_lite.corpora import treebank
>>> from itertools import islice
>>> productions = []
>>> for tree in islice(treebank.parsed(), 3):
...     productions += tree.productions()
>>> grammar = pcfg.induce(S, productions)
```
8.8 Further Reading

>>> for production in grammar.productions()[:10]:
...     print production
PP -> IN NP (p=1.0)
NNP -> ‘Nov.’ (p=0.0714285714286)
NNP -> ‘Agnew’ (p=0.0714285714286)
JJ -> ‘industrial’ (p=0.142857142857)
NP -> CD NNS (p=0.133333333333)
, -> ‘,’ (p=1.0)
CC -> ‘and’ (p=1.0)
NNP -> ‘Pierre’ (p=0.0714285714286)
NP -> NNP NNP NNP NNP (p=0.0666666666667)
NNP -> ‘Rudolph’ (p=0.0714285714286)

Note
Grammar induction usually involves normalizing the grammar in various ways. The nltk_lite.parse.treetransforms module supports binarization (Chomsky Normal Form), parent annotation, Markov order-N smoothing, and unary collapsing. This information can be accessed by importing treetransforms from nltk_lite.parse, then calling help(treetransforms).

8.8 Further Reading

About this document...
This chapter is a draft from Introduction to Natural Language Processing, by Steven Bird, Ewan Klein and Edward Loper, Copyright © 2007 the authors. It is distributed with the Natural Language Toolkit [http://nltk.sourceforge.net], Version 0.7.1, under the terms of the Creative Commons Attribution-ShareAlike License [http://creativecommons.org/licenses/by-sa/2.5/].


[JurafskyMartin] Daniel Jurafsky and James H. Martin Speech and Language Processing Prentice-Hall 2nd Edition