10. Probabilistic Parsing

10.1 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.

```
>>> 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*.

```
>>> 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')
  (NP: (NP: 'fish') (Sbar: (V: 'fish') (NP: 'fish'))))
(S:
  (NP: 'fish')
  (V: 'fish')
  (NP: (NP: 'fish') (Sbar: (NP: 'fish') (V: 'fish'))))
```

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.C(n+1), where C(n) is the nth 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. Its clear that humans don't do this either!

Note that the problem is not with our choice of example. 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:

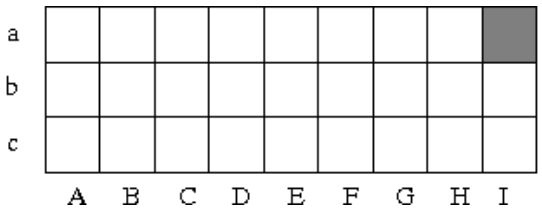

Figure 1: The a are of I

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. As we will see in this chapter, probabilistic parsing solves these twin problems of ambiguity and efficiency. However, before we deal with these parsing problems, we must first back up and introduce weighted grammars.

10.2 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 (1). 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:

(1a) Kim gave a bone to the dog

(1b) 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:

```
>>> 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]" %\
...               (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]
gave [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]
gave [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 th
  she was paying *T*-3]
give [NP: your Foster Savings Institution] [NP: the gift of hope and freedom from t
  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 a
gave [NP: quick approval] [PP-DTV: to $ 3.18 billion *U* in supplemental appropriat
  law enforcement and anti-drug programs in fiscal 1990]
give [NP: the Transportation Department] [NP: up to 50 days 0 * to review any purch
  15 % or more of the stock in an airline *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 h
  a call -RRB- or sell -LRB- a put -RRB- a specified amount of an underlying invest
  a certain date at a preset price , known * as the strike price]
gave [NP: Mr. Thomas] [NP: only a `` qualified '' rating , rather than `` well qual
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.Grammar` class. Like CFGs, each PCFG consists of a start state and a list of productions. But the productions

are represented by **pcfg.Production**, a subclass of **cfg.Production** that associates a probability with a context free grammar production.

10.2.1 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.Production** constructor, which takes a probability, a nonterminal left-hand side, and zero or more terminals and nonterminals for the right-hand side.

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

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

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

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

The probability associated with a production is returned by the **prob** method:

```
>>> print prod1.prob(), prod2.prob(), prod3.prob()
0.23 0.12 0.04
```

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:

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

10.2.2 PCFGs

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

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

```
>>> grammar = pcfg.Grammar(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:

for all *lhs*: $SIGMA_{rhs}\ \mathrm{P}(lhs \rightarrow rhs) = 1$

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:

```
>>> 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))
```

10.3 Probabilistic Parsers

10.3.1 The Probabilistic Parser Interface

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

```
>>> from nltk_lite.parse import ViterbiParse
>>> from nltk_lite import tokenize
>>> 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=0.06

>>> viterbi_parser.get_parse_list(sent)
[(S: (NP: 'Jack') (p=0.2) (VP: 'saw' (NP: 'the' 'cookie') (p=0.8)) (p=0.32)) (p=0.0
```

10.3.2 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.

10.3.3 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:

Most Likely Constituents Table			
Span	**Node**	**Tree**	**Prob**
[0:1]	NP	(NP: I)	0.3
[2:3]	NP	(NP: John)	0.3
[4:6]	NP	(NP: my cookie)	0.2
[3:6]	PP	(PP: with (NP: my cookie))	0.1
[2:6]	NP	(NP: (NP: John) (PP: with (NP: my cookie)))	0.01
[1:3]	VP	(VP: saw (NP: John)))	0.03
[1:6]	VP	(VP: saw (NP: (NP: John) (PP: with (NP: my cookie))))	0.001
[0:6]	S	(S: (NP: I) (VP: saw (NP: (NP: John) (PP: with (NP: my cookie)))))	0.0001

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: saw (NP: (NP: John) (PP: with (NP: my cookie))))

2. "used my cookie to see John":

   (VP: 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 in 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 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.

10.3.4 Using `ViterbiParse`

Viterbi parsers are created using the **`ViterbiParse`** constructor:

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

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

```
>>> viterbi_parser1 = ViterbiParse(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)

>>> viterbi_parser2 = ViterbiParse(pcfg.toy2)
>>> sent2 = list(tokenize.whitespace('the boy saw Jack with Bob under the table wit
>>> 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:
            (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.

```
>>> viterbi_parser1.trace(3)
>>> tree = viterbi_parser1.parse(sent1)
Inserting tokens into the most likely constituents table...
   Insert: |=.....| I
   Insert: |.=....| saw
   Insert: |..=...| John
   Insert: |...=..| with
   Insert: |....=.| my
   Insert: |.....=| cookie
Finding the most likely constituents spanning 1 text elements...
   Insert: |=.....| NP -> 'I' (p=0.15)              0.1500000000
   Insert: |.=....| V -> 'saw' (p=0.65)             0.6500000000
   Insert: |.=....| VP -> V (p=0.2)                 0.1300000000
   Insert: |..=...| NP -> 'John' (p=0.1)            0.1000000000
   Insert: |...=..| P -> 'with' (p=0.61)            0.6100000000
   Insert: |....=.| Det -> 'my' (p=0.2)             0.2000000000
   Insert: |.....=| N -> 'cookie' (p=0.5)           0.5000000000
Finding the most likely constituents spanning 2 text elements...
   Insert: |==....| S -> NP VP (p=1.0)              0.0195000000
   Insert: |.==...| VP -> V NP (p=0.7)              0.0455000000
```

```
   Insert: |....==| NP -> Det N (p=0.5)              0.0500000000
Finding the most likely constituents spanning 3 text elements...
   Insert: |===...| S -> NP VP (p=1.0)               0.0068250000
   Insert: |...===| PP -> P NP (p=1.0)               0.0305000000
Finding the most likely constituents spanning 4 text elements...
   Insert: |..====| NP -> NP PP (p=0.25)             0.0007625000
Finding the most likely constituents spanning 5 text elements...
   Insert: |.=====| VP -> VP PP (p=0.1)              0.0001387750
   Insert: |.=====| VP -> V NP (p=0.7)               0.0003469375
  Discard: |.=====| VP -> VP PP (p=0.1)              0.0001387750
Finding the most likely constituents spanning 6 text elements...
   Insert: |======| S -> NP VP (p=1.0)               0.0000520406
```

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`**.

10.4 A Bottom-Up PCFG Chart Parser

10.4.1 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.

10.4.2 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.

10.4.3 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 → NP • VP]@[0:2]** is the probability of its NP child times the probability of the PCFG production **S → NP 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.

10.4.4 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 → •] 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.

10.4.5 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:

```
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:

```
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:

```
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:

```
# 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()))
```

10.4.6 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:

```
>>> 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)

>>> 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')))))) (p=5.2040625e-05)
(S:
  (NP: 'I')
  (VP:
    (VP: (V: 'saw') (NP: 'John'))
    (PP: (P: 'with') (NP: (Det: 'my') (N: 'cookie'))))) (p=2.081625e-05)
```

Warning

`BottomUpChartParse` 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.

```
>>> 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
  |. > . . . . .| [1:1] VP -> * VP PP              prob=0.1
  |. . . . [-> .| [4:5] NP -> Det * N              prob=0.1
  |. . [-] . . .| [2:3] NP -> 'John' *             prob=0.1
  |. . > . . . .| [2:2] S  -> * NP VP              prob=1.0
  |. . > . . . .| [2:2] NP -> * NP PP              prob=0.25
  |. . [-> . . .| [2:3] S  -> NP * VP              prob=0.1
  |. . > . . . .| [2:2] NP -> * 'John'             prob=0.1
  |. . . . [---]| [4:6] NP -> Det N *             prob=0.05
  |. . . . > . .| [4:4] S  -> * NP VP              prob=1.0
  |. . . . > . .| [4:4] NP -> * NP PP              prob=0.25
  |. . . . [--->| [4:6] S  -> NP * VP              prob=0.05
  |. [---] . . .| [1:3] VP -> V NP *               prob=0.0455
  |[-> . . . . .| [0:1] NP -> NP * PP              prob=0.0375
  |. . . [-----]| [3:6] PP -> P NP *               prob=0.0305
  |. . [-> . . .| [2:3] NP -> NP * PP              prob=0.025
```

```
|[---] . . . .| [0:2] S  -> NP VP *          prob=0.0195
|. [-> . . . .| [1:2] VP -> VP * PP          prob=0.013
|. . . . [--->| [4:6] NP -> NP * PP          prob=0.0125
|[-----] . . .| [0:3] S  -> NP VP *          prob=0.006825
|. [---> . . .| [1:3] VP -> VP * PP          prob=0.00455
|. . [-------]| [2:6] NP -> NP PP *          prob=0.0007625
|. . [------->| [2:6] S  -> NP * VP          prob=0.0007625
|. [---------]| [1:6] VP -> V NP *           prob=0.0003469375
|. . [------->| [2:6] NP -> NP * PP          prob=0.000190625
|. [---------]| [1:6] VP -> VP PP *          prob=0.000138775
|[===========]| [0:6] S  -> NP VP *          prob=5.2040625e-05
|. [--------->| [1:6] VP -> VP * PP          prob=3.469375e-05
|[===========]| [0:6] S  -> NP VP *          prob=2.081625e-05
|. [--------->| [1:6] VP -> VP * PP          prob=1.38775e-05
```

10.5 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`** → **`V NP PP`** is *p(V,NP,PP | VP)*. We calculate this as follows:

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

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

```
>>> 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)
>>> 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)`.

10.6 Further Reading

Steven Abney (1996). Statistical Methods and Linguistics. In: Judith Klavans and Philip Resnik (eds.), The Balancing Act: Combining Symbolic and Statistical Approaches to Language. MIT Press. `http://www.vinartus.net/spa/95c.pdf`

Christopher Manning and Hinrich Schutze (1999). *Foundations of Statistical Natural Language Processing*. MIT Press. (esp chapter 12).

Joan Bresnan and Jennifer Hay (2006). *Gradient Grammar: An Effect of Animacy on the Syntax of give in Varieties of English* http://www-lfg.stanford.edu/bresnan/anim-spokensyntax-final.pdf