Stochastic Lexicalized Context-Free Grammar

Yves Schabes and Richard C. Waters

Mitsubishi Electric Research Laboratories
201 Broadway, Cambridge, MA 02139
email: {schabes|dick}@merl.com

Abstract

Stochastic lexicalized context-free grammar (SLCFG) is an attractive compromise between the parsing efficiency of stochastic context-free grammar (SCFG) and the lexical sensitivity of stochastic lexicalized tree-adjoining grammar (SLTAG). SLCFG is a restricted form of SLTAG that can only generate context-free languages and can be parsed in cubic time. However, SLCFG retains the lexical sensitivity of SLTAG and is therefore a much better basis for capturing distributional information about words than SCFG.

1 Motivation

The application of stochastic techniques to syntax modeling has recently regained popularity. Most of the work in this area has tended to emphasize one or the other of the following two goals. The first goal is to capture as much distributional information about words as possible. The second goal is to capture as many of the hierarchical constraints inherent in natural languages as possible. Unfortunately, these two goals have been more or less incompatible to date.

Early stochastic proposals such as Markov Models, N-gram models [2, 14] and Hidden Markov Models [7] are very effective at capturing simple distributional information about adjacent words. However, they cannot capture long range distributional information nor the hierarchical constraints inherent in natural languages.

Stochastic context-free grammar (SCFG) [1, 3, 5] extends context-free grammar (CFG) by associating each rule with a probability that controls its use. Each rule is associated with a single probability that is the same for all the sites where the rule can be applied.

SCFG captures hierarchical information just as well as CFG; however, it does not do a good job of capturing distributional information about words. There are at least two reasons for this. First, many rules do not contain any words and therefore the associated probabilities do not have any direct link to words. Second, distributional phenomena that involve the application of two or more rules do not have a direct link to any of the stochastic parameters of SCFG, because the probabilities apply only to single rules.

It has been observed in practice that SCFG performs worse than non-hierarchical approaches. This has lead many researchers to believe that simple distributional information about adjacent words is the most important single source of information. In the absence of a formalism that adequately combines this information with other kinds of information, the emphasis in research has been on simple non-hierarchical statistical models of words, such as word N-gram models.

Recently, it has been suggested that stochastic lexicalized tree-adjoining grammar (SLTAG) [8, 9] may be able to capture both distributional and hierarchical information. An SLTAG grammar consists of a set of trees each of which contains one or more lexical items. These elementary trees can be viewed as the elementary clauses (including their transformational variants) in which the lexical items participate. The elementary trees are combined by substitution and adjunction. Each possible way of combining two trees is associated with a probability.

Since it is based on tree-adjoining grammar (TAG), SLTAG can capture some kinds of hier-

archical information that cannot be captured by SCFG. However, the key point of comparison between SLTAG and SCFG is that since SLTAG is lexicalized and uses separate probabilities governing each possible combination of trees, each probability is directly linked to a pair of words. This makes it possible to represent a great deal of distributional information about words.

Unfortunately, the statistical algorithms for SLTAG [9] require much more computational resources than the ones for SCFG. For instance, the algorithms for estimating the stochastic parameters and determining the probability of a string require in the worst case $O(n^6)$-time for SLTAG [9] but only $O(n^3)$-time for SCFG [3].

Stochastic lexicalized context-free grammar (SLCFG) is a restricted form of SLTAG that retains most of the advantages of SLTAG without requiring any greater computational resources than SCFG. SLTAG restricts the elementary trees that are possible and the way adjunction can be performed. These restrictions limit SLCFG to producing only context-free languages and allow SLCFG to be parsed in $O(n^3)$-time in the worst case. However, SLCFG retains most of the key features of SLTAG enumerated above. In particular, the probabilities in SLCFG are directly linked to pairs of words.

SLCFG is a stochastic extension of lexicalized context-free grammar (LCFG) [12, 13]. The following sections, introduce LCFG, define the stochastic extension to SLCFG, present an algorithm that can determine the probability of a string generated by an SLCFG in $O(n^3)$-time, and discuss the algorithms needed to train the parameters of an SLCFG.

2 LCFG

Lexicalized context-free grammar (LCFG) [12, 13] is a tree generating system that is a restricted form of lexicalized tree-adjoining grammar (LTAG) [4]. The grammar consists of two sets of trees: initial trees, which are combined by substitution and auxiliary trees, which are combined by adjunction. An LCFG is lexicalized because every initial and auxiliary tree is required to contain a terminal symbol on its frontier.

Definition 1 An *LCFG* is a five-tuple (Σ, NT, I, A, S), where Σ is a set of terminal symbols, NT is a set of non-terminal symbols, I and A are finite sets of finite trees labeled by terminal and non-terminal symbols, and S is a distinguished non-terminal start symbol. The set $I \cup A$ is referred to as the elementary trees.

The interior nodes in each elementary tree are labeled by non-terminal symbols. The nodes on the frontier of each elementary tree are labeled with terminal symbols, non-terminal symbols, and the empty string (ε). At least one frontier node is labeled with a terminal symbol. With the possible exception of one (see below), the non-terminal symbols on the frontier are marked for substitution. (By convention, substitutability is indicated in diagrams by using a down arrow ($\downarrow$).)

The difference between auxiliary trees and initial trees is that each auxiliary tree has exactly one non-terminal frontier node that is marked as the foot. The foot must have the same label as the root. (By convention, the foot of an auxiliary tree is indicated in diagrams by using an asterisk ($*$).) The path from the root of an auxiliary tree to the foot is called the *spine.*

Auxiliary trees in which every non-empty frontier node is to the left of the foot are called *left* auxiliary trees. Similarly, auxiliary trees in which every non-empty frontier node is to the right of the foot are called *right* auxiliary trees. Other auxiliary trees are called *wrapping* auxiliary trees.[1]

LCFG does not allow adjunction to apply to foot nodes or nodes marked for substitution. LCFG allows the adjunction of a left auxiliary tree and a right auxiliary tree on the same node. However, LCFG does not allow the adjunction of either two left or two right auxiliary trees on the same node.

Crucially, LCFG does not allow wrapping auxiliary trees. It does not allow elementary wrapping auxiliary trees, and it does not allow the adjunction of two auxiliary trees, if the result would be a wrapping auxiliary tree.

Figure 1, shows seven elementary trees that might appear in an LCFG for English. The trees containing 'boy', 'saw', and 'left' are initial trees. The remainder are auxiliary trees.

[1] In [13] these three kinds of auxiliary trees are referred to differently as right recursive, left recursive, and centrally recursive, respectively.

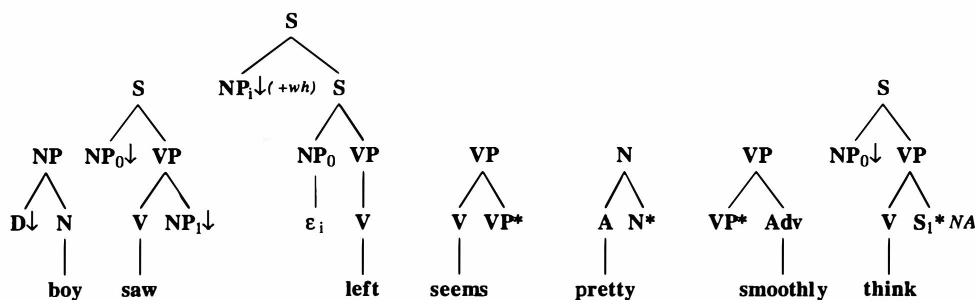

Figure 1: Example LCFG trees.

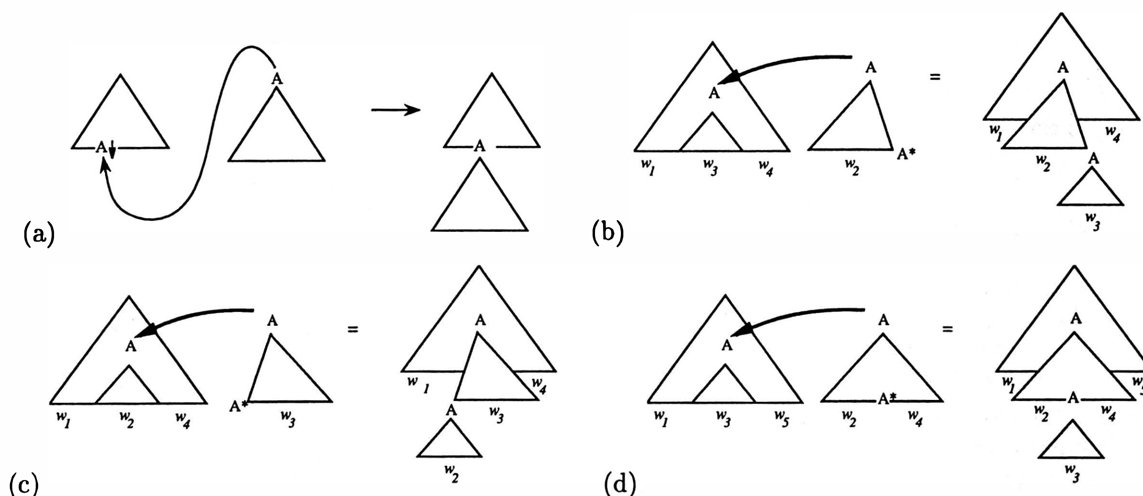

Figure 2: Tree combination: (a) substitution, (b) left adjunction, (c) right adjunction, and (d) wrapping adjunction, which is not allowed by SLCFG.

An LCFG derivation must start with an initial tree rooted in S. After that, the tree can be repeatedly extended using substitution and adjunction. A derivation is complete when every frontier node is labeled with a terminal symbol.

As illustrated in Figure 2a, substitution replaces a node marked for substitution with a copy of an initial tree.

Adjunction inserts a copy of an auxiliary tree T into another tree at an interior node η that has the same label as the root (and therefore foot) of T. In particular, η is replaced by a copy of T and the foot of the copy of T is replaced by the subtree rooted at η. The adjunction of a left auxiliary tree is referred to as left adjunction (see Figure 2b). The adjunction of a right auxiliary tree is referred to as right adjunction (see Figure 2c).

LCFG's prohibition on wrapping auxiliary trees can be rephrased solely in terms of elementary trees. To start with, there must be no elementary wrapping auxiliary trees. In addition, an elementary left (right) auxiliary tree cannot be adjoined on any node that is on the spine of an elementary right (left) auxiliary tree. Further, no adjunction whatever is permitted on a node η that is to the right (left) of the spine of an elementary left (right) auxiliary tree T. (Note that for T to be a left (right) auxiliary tree, every frontier node subsumed by η must be labeled with ε.)

Tree adjoining grammar formalisms typically forbid adjunction on foot nodes and substitution nodes. In addition, they typically forbid multiple adjunctions on a node. However, in the case of LCFG, it is convenient to relax this latter restriction slightly by allowing right and left adjunction on a node, but at most once each. (Due to the other restrictions placed on LCFG, this relaxation

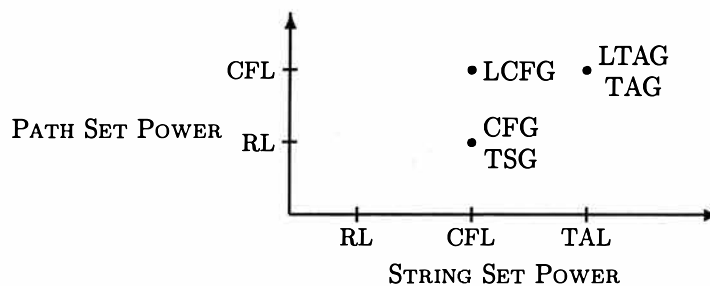

Figure 3: The tree and string complexity of LCFG and several other formalisms

increases the trees that can be generated without increasing the ambiguity of derivations.)

2.1 Comparisons

The only important difference between LCFG and LTAG is that LTAG allows both elementary and derived wrapping auxiliary trees. The importance of this is that wrapping adjunction (see Figure 2d) encodes string wrapping and is therefore context sensitive in nature. In contrast, left and right adjunction (see Figures 2b & 2c) merely support string concatenation. As a result, while LTAG is context sensitive in nature, LCFG is limited to generating only context-free languages.

To see that LCFG can only generate context-free languages, consider that any LCFG G can be converted into a CFG generating the same strings in two steps as follows. First, G is converted into a tree substitution grammar (TSG) G' that generates the same strings. Then, this TSG is converted into a CFG G''.

A TSG is the same as an LCFG (or LTAG) except that there cannot be any auxiliary trees. To create G' first make every initial tree of G be an initial tree of G'. Next, make every auxiliary tree T of G be an initial tree of G'. When doing this, relabel the foot of T with ε (turning T into an initial tree). In addition, let A be the label of the root of T. If T is a left auxiliary tree, rename the root to A_L; otherwise rename it to A_R.

To complete the creation of G' alter every node η in every initial tree in G' as follows: Let A be the label of η. If left adjunction is possible at η, add a new first child of η labeled A_L, mark it for substitution, and add a tree corresponding to $A_L \rightarrow \varepsilon$ if one does not already exist. Right adjunction is handled analogously by adding a new last child of η labeled A_R and insuring the existance of a tree corresponding to $A_R \rightarrow \varepsilon$.

The TSG G' generates the same strings as G, because all cases of adjunction have been changed into equivalent substitutions. Note that the transformation would not work if LCFG allowed wrapping auxiliary trees. The TSG G' can be converted into a CFG G'' by flattening each tree in G' into a context-free rule that expands the root of the tree into the frontier in one step.

Although the string sets generated by LCFG are the same as those generated by CFG, LCFG is capable of generating more complex sets of trees than CFG. In particular, it is interesting to look at the path sets of the trees generated. (The path set of a grammar is the set of all paths from root to frontier in the trees generated by the grammar. The path set is a set of strings over $\Sigma \cup NT \cup \{\varepsilon\}$.)

The path sets for CFG (and TSG) are regular languages [15]. In contrast, just as for LTAG and TAG, the path sets for LCFG are context-free languages. To see this, consider that adjunction makes it possible to embed a sequence of nodes (the spine of the auxiliary tree) in place of a node on a path. Therefore, from the perspective of the path set, auxiliary trees are analogous to context-free productions.

Figure 3 summarizes the relationship between LCFG and several other grammar formalisms. The horizontal axis shows the complexity of strings that can be generated by the formalisms, i.e., regular languages (RL), context-free languages (CFL), and tree adjoining languages (TAL). The vertical axis shows the complexity of the path sets that can be generated.

CFG (and TSG) create context-free languages, but the path sets they create are regular languages. LTAG and TAG generate tree adjoining languages and have path sets that are context-free languages. LCFG is intermediate in nature. It can only generate context-free languages, but has path sets that are also context-free languages.

2.2 LCFG lexicalizes CFG

As shown in [12, 13] LCFG lexicalizes CFG without changing the trees derived. Further, a constructive procedure exists for converting any CFG G into an equivalent LCFG G'.

The fact that LCFG lexicalizes CFG is significant, because every other method for lexicalizing CFGs without changing the trees derived requires context-sensitive operations [4] and therefore dramatically increases worst case processing time.

As shown in [12, 13] (and in Section 4) LCFG can be parsed in the worst case just as quickly as CFG. Since LCFG is lexicalized, it is expected that it can be parsed much faster than CFG in the typical case.

3 Stochastic LCFG

The definition of stochastic lexicalized context-free grammar (SLCFG) is the same as the definition of LCFG except that probabilities are added that control the combination of trees by adjunction and substitution.

Definition 2 An *SLCFG* is an 11-tuple (Σ,NT, I,A,S,P_I,P_S,P_L,P_{NL},P_R,P_{NR}), where (Σ,NT,I, A,S) is an LCFG and P_I, P_S, P_L, P_{NL}, P_R, and P_{NR} are statistical parameters as defined below.

For every root ρ of an initial tree, $P_I(\rho)$ is the probability that a derivation starts with the tree rooted at ρ. It is required that:

$$\sum_{\rho} P_I(\rho) = 1$$

Note that $P_I(\rho) \neq 0$ if and only if ρ is labeled S.

For every root ρ of an initial tree and every node η that is marked for substitution, $P_S(\rho, \eta)$ is the probability of substituting the tree rooted at ρ for η. For each η it is required that:

$$\sum_{\rho} P_S(\rho, \eta) = 1$$

For every node η in every elementary tree, $P_{NL}(\eta)$ is the probability that left adjunction will not occur on η. For every root ρ of a left auxiliary tree, $P_L(\rho, \eta)$ is the probability of adjoining the tree rooted at ρ on η. For each η it is required that:

$$P_{NL}(\eta) + \sum_{\rho} P_L(\rho, \eta) = 1$$

$P_{NL}(\eta) = 0$ if and only if left adjunction on η is obligatory.

The parameters $P_{NR}(\eta)$ and $P_R(\rho, \eta)$ control right adjunction in an exactly analogous way.

An SLCFG derivation is described by the initial tree it starts with, together with the sequence of substitution and adjunction operations that take place. The probability of a derivation is defined as the product of: the probability P_I of starting with the given tree, the probabilities P_S, P_L, and P_R of the operations that occurred, and the probabilities P_{NL} and P_{NR} of adjunction not occurring at the places where it did not occur.

The probability of a string is the sum of the probabilities of all the different ways of deriving it. A most likely derivation of a string is a derivation that has as large a probability as any other derivation for the string. The probability of a tree generated by an SLCFG for a string is the sum of the probabilities of every way of deriving the tree. (Unlike in SCFG, in SLCFG there can be more than one way to derive a given tree.) A most likely tree generated for a string is a tree whose probability is as large as any other tree generated for the string. (Note that a most likely derivation need not generate a most likely tree.)

4 Parsing SLCFG

Since SLCFG is a restricted case of SLTAG, the $O(n^6)$-time SLTAG parser [9] can be used for parsing SLCFG. Further, it can be straightforwardly modified to require at most $O(n^4)$-time when applied to SLCFG. However, this does not take full advantage of the context-freeness of SLCFG.

This section demonstrates that SLCFG can be parsed in $O(n^3)$-time by exhibiting a CKY-style

bottom-up algorithm for computing the probability assigned to a string by an SLCFG. This algorithm can be trivially modified to extract a most probable derivation of the given string. More efficient SLCFG processors can be based on the Earley style LCFG recognizer presented in [12].

4.1 Terminology

Suppose that G is an SLCFG and that $a_1 \cdots a_n$ is an input string. Let η be a node in an elementary tree (identified by the name of the tree and the position of the node in the tree).

Label(η) $\in \Sigma \cup NT \cup \varepsilon$ is the label of the node. The predicate IsInitialRoot(η) is true if and only if η is the root of an initial tree. Parent(η) is the node that is the parent of η or $\perp$ if η has no parent. FirstChild(η) is the node that is the leftmost child of η or $\perp$ if η has no children. Sibling(η) is the node that is the next child of the parent of η (in left to right order) or $\perp$ if there is no such node.

The predicate Substitutable(ρ, η) is true if and only if η is marked for substitution and ρ is the root of an initial tree that can be substituted for η. The predicate Radjoinable(ρ, η) is true if and only if ρ is the root of an elementary right auxiliary tree that can adjoin on η. The predicate Ladjoinable(ρ, η) is true if and only if ρ is the root of an elementary left auxiliary tree that can adjoin on η.

The concept of *covering* is critical to the bottom-up algorithm shown below. Informally speaking, a node η covers a string if and only if the string can be derived starting from η.

More precisely, for every node η in every elementary tree in G, let T' be a copy of the subtree of T that is rooted at η. Extend T' by adding a new root whose only child is the original root of T'. Label the new root of T' with a unique new symbol S'. If there is a node on the frontier of T' that is marked as the foot, relabel this node with ε. This converts T' into an initial tree. Let G_η be an SLCFG that is identical to G except that T' is introduced as an additional initial tree and the start symbol of G_η is S'. The probabilities associated with the the interior nodes of T' are identical to those for the corresponding nodes in T. The probabilities for the root of T' are $P_S = P_L = P_R = 0$, $P_{NL} = P_{NR} = 1$, and crucially $P_I = 1$. $P_I = 0$ for the other initial trees.

The node η covers a string $a_1 \cdots a_n$ with probability p in G if and only if the probability of $a_1 \cdots a_n$ in G_η is p. The node η covers a string $a_1 \cdots a_n$ without left (right) adjunction with probability p in G if and only if the probability of $a_1 \cdots a_n$ in G_η is p without considering derivations where left (right) adjunction occurs on the original root of T'.

(Note that if η is a foot node, T' is an empty tree. The only string covered by η is the empty string; however, the empty string is covered with probability 1, because the empty string is the only string derived by G_η.)

4.2 A bottom-up Algorithm

We can assume without loss of generality that every node in $I \cup A$ has at most two children. (By adding new nodes, any SLCFG can be transformed into an equivalent SLCFG satisfying this condition. This transformation can be readily reversed after parsing has been completed.)

The algorithm stores triples of the form $[\eta, code, p]$ in an $n \times n$ array C. In a triple, *code* is a set over the universe L (for left adjunction) and R (for right adjunction). The fact that $[\eta, code, p] \in C[i,k]$ means that η accounts for the substring $a_{i+1} \cdots a_k$ with probability p. More precisely, for every node η in every elementary tree in G, the algorithm guarantees that when the computation concludes:

- $[\eta, \emptyset, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p without left or right adjunction.
- $[\eta, \{L\}, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p without right adjunction.
- $[\eta, \{R\}, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p without left adjunction.
- $[\eta, \{L,R\}, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p.

The process starts by placing each foot node and each frontier node that is labeled with the empty string in every cell $C[i,i]$ with probability one. This signifies that they each cover the empty string at all positions. The initialization also puts each terminal node η in every cell $C[i,i+1]$ where

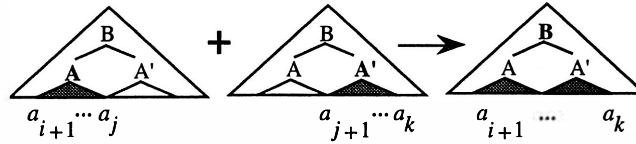

Figure 4: Sibling concatenation.

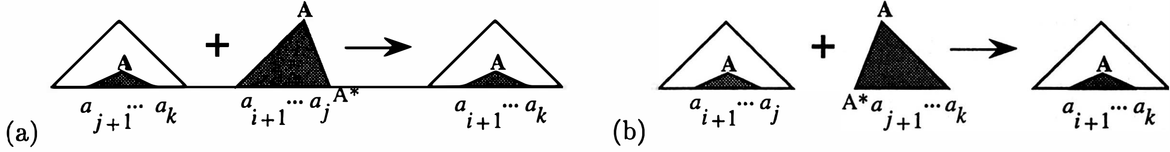

Figure 5: (a) Left concatenation and (b) right concatenation.

```
Procedure Probability(a_1 ··· a_n)
 begin
  for i = 0 to n
   for all foot nodes φ in A, Add(φ, ∅, i, i, 1)
   for all frontier nodes η in A ∪ I where Label(η) = ε, Add(η, ∅, i, i, 1)
  for i = 0 to n − 1
   for all frontier nodes η in A ∪ I where Label(η) = a_{i+1}, Add(η, ∅, i, i + 1, 1)
  for d = 0 to n
   for i = 0 to n − d
    set k = i + d
    for j = i to k
     for all nodes η in G
      if [η, {L, R}, p_1] ∈ C[i, j] and [Sibling(η), {L, R}, p_2] ∈ C[j, k]
       then Add(Parent(η), ∅, i, k, p_1 × p_2)
     for all nodes ρ and η in G where Ladjoinable(ρ, η)
      if [ρ, {L, R}, p_1] ∈ C[i, j] and [η, code, p_2] ∈ C[j, k] and L ∉ code
       then Add(η, L ∪ code, i, k, p_1 × p_2 × P_L(ρ, η))
     for all nodes ρ and η in G where Radjoinable(ρ, η)
      if [η, code, p_1] ∈ C[i, j] and R ∉ code and [ρ, {L, R}, p_2] ∈ C[j, k]
       then Add(η, R ∪ code, i, k, p_1 × p_2 × P_R(ρ, η))
  p = 0
  for all nodes ρ in G where IsInitialRoot(ρ) and Label(ρ) = S
   if [ρ, {L, R}, p_0] ∈ C[0, n] then p = p + p_0 × P_I(ρ)
  return p
 end
Procedure Add(η, code, i, k, p)
 begin
  if [η, code, p'] ∈ C[i, k] for some p' then update [η, code, p'] in C[i, k] to [η, code, p' + p]
  else C[i, k] := C[i, k] ∪ [η, code, p]
  if code = {L, R} then
   if FirstChild(Parent(η)) = η and Sibling(η) = ⊥ then Add(Parent(η), ∅, i, k, p)
   for each node φ such that Substitutable(η, φ), Add(φ, ∅, i, k, p × P_S(η, φ))
  if L ∉ code then Add(η, L ∪ code, i, k, p × P_NL(η))
  if R ∉ code then Add(η, R ∪ code, i, k, p × P_NR(η))
 end
```

Figure 6: A procedure for computing the probability of a string given an SLCFG.

η is labeled a_{i+1} with probability one. The algorithm then considers all possible ways of combining matched substrings into longer matched substrings—it fills the upper diagonal portion of the array $C[i,k]$ $(0 \leq i \leq k \leq n)$ for increasing values of $k-i$.

Two observations are central to the efficiency of this process. Since every auxiliary tree (elementary and derived) in SLCFG is either a left or right auxiliary tree, the substring matched by a tree is always a contiguous string. Further, when matched substrings are combined, the algorithm only has to consider adjacent substrings. (In SLTAG, a tree with a foot can match a pair of strings that are not contiguous—one left of the foot and one right of the foot.)

There are three situations where combination of matched substrings is possible: sibling concatenation, left concatenation, and right concatenation.

As illustrated in Figure 4, sibling concatenation combines the substrings matched by two sibling nodes into a substring matched by their parent. In particular, suppose that there is a node η_0 (labeled B in Figure 4) with two children η_1 (labeled A) and η_2 (labeled A'). If $[\eta_1, \{L,R\}, p_1] \in C[i,j]$ and $[\eta_2, \{L,R\}, p_2] \in C[j,k]$ then $[\eta_0, \emptyset, p_1 \times p_2] \in C[i,k]$.

Left concatenation (see Figure 5a) combines the substring matched by a left auxiliary tree with the substring matched by a node the auxiliary tree can adjoin on. Right concatenation (see Figure 5b) is analogous.

The algorithm (see Figure 6) is written in two parts: a main procedure Probability$(a_1 \cdots a_n)$ and a subprocedure Add$(\eta, \mathit{code}, i, k)$, which adds the triple $[\eta, \mathit{code}, p]$ into $C[i,k]$.

The main procedure repeatedly scans the array C, building up longer and longer matched substrings until it determines all the S-rooted derived trees that match the input. The purpose of the codes ($\{L,R\}$ etc.) is to insure that left and right adjunction can each be applied at most once on a node. The procedure could easily be modified to account for other constraints on the way derivation should proceed, such as those suggested for LTAGs [11].

The procedure Add enters a triple $[\eta, \mathit{code}, p]$ into $C[i,k]$. If some other triple $[\eta, \mathit{code}, p']$ is already present in $C[i,k]$, then the probability p' is updated to $p'+p$ to reflect the fact that an additional derivation of $a_{i+1} \cdots a_k$ has been found. Otherwise, a new triple $[\eta, \mathit{code}, p]$ is added to $C[i,k]$.

The procedure Add also propagates information from one triple to another in situations where the length of the matched string is not increased—i.e., when a node is the only child of its parent, when substitution occurs, and when adjunction is not performed.

The $O(n^3)$ complexity of the algorithm follows from the three nested induction loops on d, i and j. (Although the procedure Add is defined recursively, the number of pairs added to C is bounded by a constant that is independent of sentence length.)

The algorithm does not depend on the fact that SLCFG is lexicalized—it would work equally well if were not lexicalized. If the sum $p'+p$ on the third line of the Add procedure is changed to $\max(p', p)$ the algorithm computes the probability of a most probable derivation. By keeping a record of every attempt to enter a triple into a cell of the array C, one can extend the algorithm so that derivations and therefore the trees they generate can be rapidly recovered.

5 Training an SLCFG

In the general case, the training algorithm for SCFG [5] requires $O(n^3)$-time for each sentence of length n. A training algorithm for SLCFG can be constructed that achieves these same worst case bounds.

To start with, since SLCFG is a restricted case of stochastic lexicalized tree-adjoining grammar (SLTAG), the $O(n^6)$-time inside-outside reestimation algorithm for SLTAG [9] can be used for estimating the parameters of an SLCFG given a training corpus. Straightforward modifications lead to an $O(n^4)$-time algorithm for training an SLCFG. However, this alone does not achieve the full potential of SLCFG.

The same basic construction that underlies the algorithm in the last section can be used as the basis for an $O(n^3)$ inside-outside training algorithm for SLCFG. As in the last section, the key reason for this is that computations involving SLCFG only require the consideration of contiguous strings.

It should be noted that in the special case of a fully bracketed training corpus, the parameters of an SCFG can be estimated in linear time [6, 10]. It is an open question whether this can be done for SLCFG. However, it should be straightforward to design an $O(n^2)$-time training algorithm for SLCFG given a fully bracketed corpus.

6 Conclusion

The preceding sections present stochastic lexicalized context-free grammar (SLCFG). SLCFG combines the processing speed of SCFG with the much greater ability of SLTAG to capture distributional information about words. As such, SLCFG has the potential of being a very useful tool for natural language processing tasks where statistical assessment/prediction is required.

References

[1] T. Booth. Probabilistic representation of formal languages. In *Tenth Annual IEEE Symposium on Switching and Automata Theory*, October 1969.

[2] F. Jelinek. Self-organized language modeling for speech recognition. In Alex Waibel and Kai-Fu Lee, editors, *Readings in speech recognition*. Morgan Kaufmann, San Mateo, California, 1990. Also in IBM Research Report (1985).

[3] F. Jelinek, J. D. Lafferty, and R. L. Mercer. Basic methods of probabilistic context free grammars. Technical Report RC 16374 (72684), IBM, Yorktown Heights, NY, 1990.

[4] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Tree Automata and Languages*. Elsevier Science, 1992.

[5] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[6] Fernando Pereira and Yves Schabes. Inside-outside reestimation from partially bracketed corpora. In 20th *Meeting of the Association for Computational Linguistics (ACL'92)*, Newark, Delaware, 1992.

[7] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[8] Philip Resnik. Probabilistic tree-adjoining grammars as a framework for statistical natural language processing. In *Proceedings of the* 14th *International Conference on Computational Linguistics (COLING'92)*, 1992.

[9] Yves Schabes. Stochastic lexicalized tree-adjoining grammars. In *Proceedings of the* 14th *International Conference on Computational Linguistics (COLING'92)*, 1992.

[10] Yves Schabes, Michael Roth, and Randy Osborne. Parsing the Wall Street Journal with the inside-outside algorithm. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics (EACL'93)*, Utrecht, the Netherlands, April 1993.

[11] Yves Schabes and Stuart Shieber. An alternative conception of tree-adjoining derivation. In 20th *Meeting of the Association for Computational Linguistics (ACL'92)*, 1992.

[12] Yves Schabes and Richard C. Waters. Lexicalized context-free grammar: A cubic-time parsable formalism that strongly lexicalizes context-free grammar. Technical Report 93-04, Mitsubishi Electric Research Labs, 201 Broadway. Cambridge MA 02139, 1993.

[13] Yves Schabes and Richard C. Waters. Lexicalized context-free grammars. In 21st *Meeting of the Association for Computational Linguistics (ACL'93)*, pages 121–129, Columbus, Ohio, June 1993.

[14] C. E. Shannon. Prediction and entropy of printed english. *The Bell System Technical Journal*, 30:50–64, 1951.

[15] J. W. Thatcher. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 5:365–396, 1971.