Fast Learning from Strings of 2-Letter Rigid Grammars

Yoav Seginer

ILLC, Universiteit van Amsterdam, The Netherlands

Abstract. It is well-known that certain classes of classical categorial grammars are learnable, within Gold's paradigm of identification in the limit, from positive examples. In the search for classes which can be learned efficiently from strings, we study the class of 2-letter rigid grammars, which is the class of classical categorial grammars with an alphabet of two letters, each of which is assigned a single type. The (non-trivial) structure of this class of grammars is studied and it is shown that grammars in this class can be learned very efficiently. The algorithm given for solving this learning problem runs in time linear in the total length of the input strings. After seeing two or more strings in a language, the algorithm can determine precisely the (finite) set of grammars which can generate those strings.

1 Introduction

It is well-known that certain classes of classical categorial grammars are learnable, within Gold's paradigm of identification in the limit, from positive examples. In [3], Kanazawa showed that the class of k-valued categorial grammars, in which every letter in the alphabet (lexicon) is assigned at most k types, is learnable from positive example strings (sequences of letters generated by the grammar). While this general result guarantees that k-valued grammars are learnable, it does not guarantee that this can be done efficiently. In fact, Kanazawa's approach is based on learning from a richer input - structural examples. Structural examples give the learner not only the strings but also the tree structure for each string. For k-valued grammars with $2 \leq k$ it has been shown in [1] that the learning problem from structural examples (under reasonable assumptions) is NP-hard. To extend this method of learning to strings, one needs to try out all possible tree structures for each string. Even for 1-valued grammars (also called *rigid* grammars), where learning from structural examples can be done in polynomial time, the associated method for learning from strings is intractable.

There may be, of course, other methods for learning grammars from strings. These methods may have to be tailored specifically for each class of grammars we wish to learn. To find them, we must both identify grammar classes for which this can be done and find an algorithm that learns the grammars in the class. In this paper we examine what is probably one of the simplest classes of classical categorial grammars - rigid grammars over an alphabet of two letters. We will

see that this class of grammars, which produces a non-trivial class of languages, has a combinatorial structure which allows very efficient learning.

Languages generated by 2-letter rigid grammars may have finite or infinite overlaps with each other or even be contained in each other. The language class does not have the property of finite thickness - there are strings which belong to infinitely many different languages (but, as shown by Kanazawa, the class does have finite elasticity). Many of the languages in this class are not regular.

Underneath this complex surface structure, there are properties which allow for very quick learning. All learning algorithms used here run in time linear in the size of the input strings. Moreover, the algorithms learn all that can be learned from any given input. This means that the algorithms can always give exactly the family of grammars whose languages contain the input strings. When these languages are contained in each other, the minimal language is given (which implies conservative learning).

The exact sequence in which strings are presented to the learner may influence how quickly the algorithms converge to a correct grammar. For every language there is a pair of strings which allows the algorithms to converge immediately to a correct grammar.

In what follows, we confine our attention to rigid grammars over two letters. We begin, in Sect. 2, by giving a labeled graph representation of these grammars. We then characterize the structure of the graphs. This will serve as the basis for the rest of the analysis. In Sect. 3 we give the learning algorithm for these grammars, step by step. At each step, an additional parameter in the structure of the associated graph will be inferred. Even when such inference is not complete, it is shown to be sufficient for the next step in the algorithm to be carried out.

2 A Graph Representation for Grammars

In classical categorial grammars, each string is associated with a type. These types determine the way in which strings may be combined together (by concatenation) into longer strings. The set of types consists of *primitive types* (which include the *distinguished type* t) and *functor types*, which are formed by combining types. For any two types t_1 and t_2, the functor types t_1/t_2 and $t_2\backslash t_1$ are defined. The type t_2 is the *argument type* of the functor types t_1/t_2 and $t_2\backslash t_1$.

If string s_1 has type t_1/t_2 and string s_2 has type t_2, the concatenation $s_1 s_2$ may be formed. Similarly, if s_1 has type $t_2\backslash t_1$, the concatenation $s_2 s_1$ is formed. In both cases, the concatenation is assigned the type t_1. We call this operation a *functor application*. The direction of the slash determines the order in which the two strings are concatenated. When we wish to ignore the direction of the slash, we write $t_1|t_2$ for both functor types t_1/t_2 and $t_2\backslash t_1$.

A categorial grammar is defined over an alphabet (set of letters). Each letter in the alphabet is assigned one or more types. The language generated by a grammar is the set of strings of type t that can be generated from the alphabet by functor applications. A *rigid* grammar is a grammar in which each letter in the alphabet is assigned exactly one type.

In this paper, we work over the alphabet $\{a, b\}$ and use directed graphs to represent rigid grammars over this alphabet. A graph has a single node for every type which can be generated by the corresponding categorial grammar. The nodes, however, are not labeled by types (there is always some arbitrariness possible in the choice of types and we wish to avoid it).

Two kinds of directed edges are used in constructing the graphs, *argument edges* (dashed arrows in diagrams) and *functor edges* (solid arrows in diagrams). An argument edge connects a node representing a functor type with the node representing the argument type of that functor (e.g. $t_1/t_2 \dashrightarrow t_2$). A functor edge connects a node representing a functor type with the node representing the type which is the result of the application of that functor (e.g. $t_1/t_2 \rightarrow t_1$).

The construction of the graph corresponding to a categorial grammar begins with two nodes which are labeled by the letters a and b and by the type assigned to each of these letters by the grammar (the type labeling of nodes is only temporary and is not part of the final graph). We call these two nodes the *start nodes*. At each step in the construction we look for two nodes which are labeled by types t_2 and $t_1|t_2$ but are not yet connected by an argument edge. We first add an argument edge from the node labeled $t_1|t_2$ to the node labeled t_2. If there is no node yet labeled t_1, we create such a node. We then add a functor edge from the node labeled $t_1|t_2$ to the node labeled t_1. We repeat this process until no more edges and nodes can be added (the process must terminate).

Next, we look for a node labeled by the distinguished type t. We select this node to be the *terminal node* (if no such node exists, the categorial grammar generates an empty language and is not interesting). We also assign slashes to nodes which have edges leaving them. Such nodes must be labeled by a functor type and we assign each such node the main slash of its type (i.e. if the type is t_1/t_2, assign / and if the type is $t_2\backslash t_1$, assign \). Finally, we remove all type labeling. For an example, see Fig. 1. When the slashes are removed from a graph G, we get the *edge structure* of G, which we denote by $E(G)$.

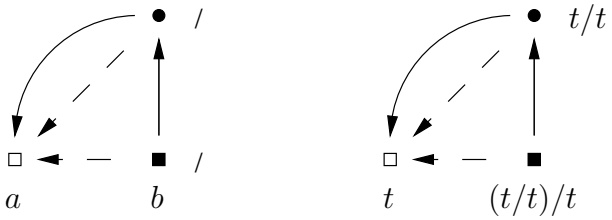

Fig. 1. The graph for the categorial grammar $a \mapsto t, b \mapsto (t/t)/t$. On the left appears the graph as constructed. On the right the nodes of the graph are labeled by the corresponding types (which are not part of the graph). Start nodes are indicated by squares and the terminal node is indicated by a hollow node (in this particular case, the left start node).

Strings can be generated at the graph nodes by a simple rule. Whenever a configuration $\nu_1 \leftarrow\!\!- \nu_2 \rightarrow \nu_3$ of three nodes has strings s_1 and s_2 already generated at nodes ν_1 and ν_2, the concatenation of s_1 and s_2 is generated at ν_3 ($s_1 s_2$ or $s_2 s_1$ depending on the direction of the slash on ν_2). The process begins with the strings a and b at the start nodes. It is not difficult to see that the strings generated at the terminal node of a graph are exactly the language generated by the grammar that the graph represents. We write $L(G)$ for the language generated at the terminal node of the graph G.

2.1 Characterizing the Graphs of 2-Letter Rigid Grammars

There is a simple characterization of the structure of graphs representing 2-letter rigid grammars. We define a *functor path* to be a directed path (in a graph) consisting of functor edges only. Since the graphs contain no directed loops, it is possible to number the nodes along a functor path. We write $p(i)$ for the i'th node on functor path p. The graphs for 2-letter rigid grammars are then characterized by the next theorem (see Fig. 2 for typical examples). Note that this characterization is given only in terms of the edge structure of the graph. Any slash assignment to the nodes is possible.

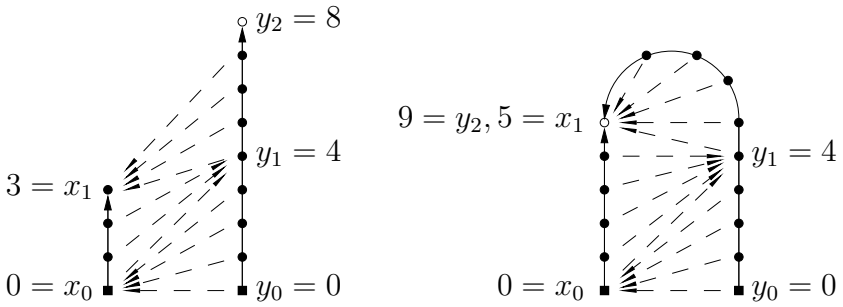

Fig. 2. Typical graphs for 2-letter rigid grammars. A loopless graph (*left*) and a looping graph (*right*). The nodes labeled by x_i and y_j are as in Theorem 1. In diagrams, we always have p_α on the left and p_β on the right.

Theorem 1. *A graph represents a 2-letter rigid categorial grammar iff:*

1. *All graph nodes lie on two functor paths p_α and p_β, which begin at the two start nodes. The two functor paths either do not meet at all or meet only once, at the last node of both paths. The terminal node is the last node on a functor path.*
2. *There are two strictly increasing sequences of natural numbers $\{x_i\}_{0\le i\le m}$ and $\{y_j\}_{0\le j\le n}$, with $0 \le n-1 \le m \le n$ such that $x_0 = y_0 = 0$, the last node in path p_α is $p_\alpha(x_m)$ and the last node in path p_β is $p_\beta(y_n)$. All argument edges are given by:*

(a) *If* $i+1 \leq n$ *then for* $y_i \leq l < y_{i+1}$ *there is an argument edge from* $p_\beta(l)$ *to* $p_\alpha(x_i)$.
(b) *If* $i+1 \leq m$ *then for* $x_i \leq l < x_{i+1}$ *there is an argument edge from* $p_\alpha(l)$ *to* $p_\beta(y_{i+1})$.

2.2 Looping and Loopless Graphs

From the theorem we see that there are two types of graphs. Graphs in which the two functor paths meet we call *looping graphs*. Graphs in which the two functor paths do not meet we call *loopless graphs*. In the analysis of looping graphs, we look separately at the *base* and the *loop* of the looping graph (see Fig. 3). The base of a looping graph is a loopless graph.

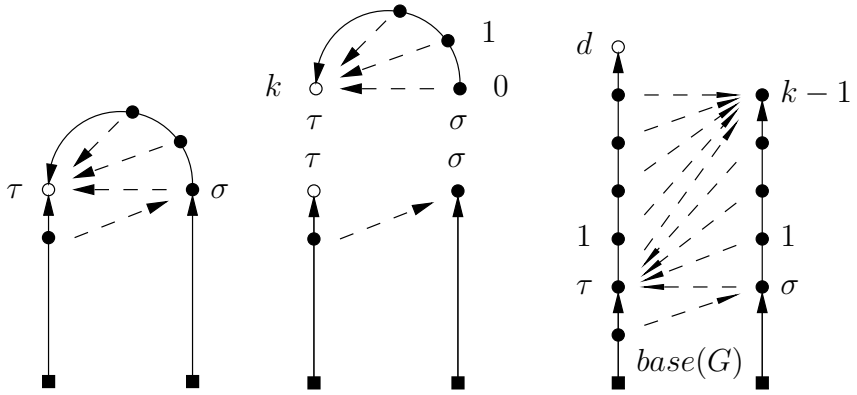

Fig. 3. A looping graph G (*left*), its base (*center bottom*), its loop (*center top*) and the graph $NL(G, d)$ (*right*) which generates a string with the same letter count as strings generated by G. Only the last nodes and argument edge of the base are shown.

In loopless graphs (and therefore also in the base of looping graphs) strings are generated strictly from the bottom up, that is, when a string is generated at a node, it is a concatenation of strings generated at lower nodes. Therefore, only one string is generated at each node of a loopless graph. On the loop, however, the strings generated at the terminal node can be used to generate additional strings on the loop.

In looping graphs, a loop is attached on top of a base. The generation of strings in these graphs therefore begins by generating two strings s_τ and s_σ at the top nodes τ and σ of the base. These two strings then serve as input for the concatenations which take place on the loop. Any string generated on the loop must be a concatenation of copies of the two strings s_τ and s_σ. One can think of s_τ and s_σ as the two words of the language, which the loop then combines into sentences.

A simple consequence of this description of the way strings are generated by a graph is a simple method for distinguishing between languages generated by looping and by loopless graphs. A language generated by a loopless graph consists of only one string, while a language generated by a looping graph is infinite.

3 Learning 2-Letter Rigid Grammars

Since the languages of loopless grammars consist of only one string, they are not interesting in terms of learnability. We therefore concentrate on learning languages generated by looping graphs. However, because the base of a looping graph is a loopless graph, we will return to the analysis of loopless graphs.

To learn the language generated by a looping graph G, a learner must deduce several parameters of the graph G from a sample of strings generated by G. The structure of the class of graphs for 2-letter rigid grammars is such that the learning process has two main stages. First, the learner has to discover the two strings s_τ and s_σ (the words of the language). These two strings are determined by the base of the graph. Having learned s_τ and s_σ, the learner can then learn the way these two strings are put together (on the loop) to construct the final strings (sentences) of the language.

The rest of this paper is concerned with answering these two questions based on a sample of strings from $L(G)$. Answering these questions requires answering a sequence of sub-questions. This can be done efficiently because the questions do not need to be answered simultaneously, but can be answered in sequence. The answer to one question then makes answering the next question in the sequence easy. Very often there remains some uncertainty as to the answer to a certain question. This uncertainty is such, however, that it does not effect the ability to answer the next question in the sequence.

3.1 Letter Assignment to the Start Nodes

The learning procedure is based on the graph representation as given in Theorem 1. The representation given in this theorem is not symmetric, however. There is always an argument edge from the start node of the path p_β (right in the diagrams) to the start node of the path p_α (left in the diagrams) and never the other way around.

There are, therefore, two ways of assigning the letters of the alphabet to the start nodes. The *standard assignment of letters* assigns a to p_α and b to p_β. The inverse assignment creates languages which are "negatives" of the languages with the standard assignment. The learner can distinguish languages with the standard and the inverse assignment of letters by counting the number of a's and b's in the sample strings.

We write $\#_a(s)$ for the number of a's in a string s and $\#_b(s)$ for the number of b's in the string. The pair $[\#_a(s), \#_b(s)]$ which we also denote by $[s]$, we call the *letter count* of the string s. As will be seen in the next section, a great part of

the structure of a grammar can be deduced from the letter counts of the strings in its language. At this stage we only observe that strings generated by graphs with the standard assignment of letters [1] have $\#_b(s) < \#_a(s)$, while strings generated by graphs with the inverse letter assignment have $\#_a(s) < \#_b(s)$. The learner can, therefore, easily distinguish the two types of languages. From now on we assume that the grammar has the standard assignment (for the inverse assignment, we simply have to exchange the roles of a and b).

3.2 Inferring the Strings s_{τ_0} and s_{σ_0}

According to our original plan, we now need to find the strings s_τ and s_σ. This, however, cannot always be done. What we can do, is find two strings s_{τ_0} and s_{σ_0}. The strings s_τ and s_σ are either equal to s_{τ_0} and s_{σ_0} or composed of concatenations of s_{τ_0} and s_{σ_0} (in which case s_{τ_0} and s_{σ_0} are the "syllables" within the words s_τ and s_σ). In either case, every string in $L(G)$ is a concatenation of copies of s_{τ_0} and s_{σ_0} and therefore, these string will do just as well as s_τ and s_σ.

The discovery of s_{τ_0} and s_{σ_0} involves two basic steps. First, the letter counts $[s_{\tau_0}]$ and $[s_{\sigma_0}]$ are determined. Next, strings with these letter counts are extracted from the ends of the available sample strings of $L(G)$.

Since the letter counts of strings generated by a graph G are independent of the slashes assigned to the nodes of G, letter counts of strings generated at the nodes of G can be determined from the edge structure $E(G)$. We therefore attempt to discover the edge structure $E(G)$. As a first step in this direction, we show that the edge structure of a loopless graph can be inferred from the letter count of the single string generated by the graph.

Inferring the Edge Structure of Loopless Graphs. The process of generating a string on a loopless graph begins with the two strings a and b which have letter counts $[1, 0]$ and $[0, 1]$. The generation then proceeds by going "up" the graph, each time concatenating two previously generated strings. Concatenation of strings amounts to the vector addition of their letter counts. This process is very similar to what happens in the so-called Stern-Brocot tree. The edge structure of every loopless graph is equivalent to a branch in the Stern-Brocot tree and the letter count of the string generated by a loopless graph is the same as the vector on the corresponding branch in the tree (see [2] for details about the Stern-Brocot tree).

The Stern-Brocot tree has several interesting properties which translate immediately to properties of loopless graphs. In the Stern-Brocot tree every branch produces a different vector and there is an algorithm which, given a vector in the Stern-Brocot tree, calculates the branch of that vector. We give a version of this algorithm which takes as input a string letter count and outputs an edge structure which generates a string with this letter count. The translation from loopless graphs to tree branches matches two loopless graphs with every branch

[1] This fails for one (simple) graph and the three strings b,ab,ba, which can be dealt with separately.

in the right half of the Stern-Brocot tree. These two graphs are very similar and we fix a representative graph from each such pair, which we call the *standard loopless graph*. The algorithm returns the standard loopless graph.

Algorithm 1. *The algorithm takes as input a letter count $[x, y]$ of some string generated by a loopless graph with the standard letter assignment (so $x \leq y$). Let c_1 and c_2 be variables, which have nodes as their values. We begin with two start nodes and let c_1 be the start node labeled by a and c_2 the start node labeled by b.*

1. *Create a new node, ν. Add an argument edge from c_2 to c_1 and a functor edge from c_2 to ν.*
2. *If $[x, y] = [1, 1]$, stop.*
3. *If $x > 2y$ or $[x, y] = [2, 1]$, set $[x, y] := [x - y, y]$, $c_2 := \nu$ and repeat the algorithm.*
4. *If $x < 2y$, set $[x, y] := [y, x - y]$, $c_2 := c_1$, $c_1 := \nu$ and repeat the algorithm.*

Inferring the Edge Structure of the Base of a Looping Graph. We cannot use Algorithm 1 directly to discover the edge structure of the base of the looping graph G, because the sample strings are generated on the loop of G and not in the base. Fortunately, it turns out that for every string $s \in L(G)$, there is a $0 \leq d$ such that the *loopless* graph $NL(G, d)$, given in Fig. 3, generates a string with the same letter count as s (but not necessarily the exact same string).

Since Algorithm 1 only looks at the letter count of the input string and not at the order of the letters within the string, applying the algorithm to a string $s \in L(G)$ results [2] in the edge structure of the graph $NL(G, d)$, for some d. We are looking for the edge structure of the base of G, and, as can be seen in Fig. 3, the base of G is the bottom of the graph $NL(G, d)$. To get the base of G, all we need to do is remove a few nodes off the top of the graph $NL(G, d)$. The question is how many nodes off the top of $NL(G, d)$ should be removed.

We say that several nodes belong to the same *node sequence* if the nodes just below these nodes (along the functor path) are all connected with argument edges to the same node. This property can be read off a graph directly. Looking at the diagram in Fig. 3, we see that to obtain the base of G, all we need to do is remove the last node sequence from each functor path in $NL(G, d)$.

There remain two slight problems. The first is that when $k = 1$, a node sequence has to be removed only from one functor path, while when $1 < k$, node sequences have to be removed from both functor paths. Since we do not know the value of k, we do not know how many nodes to remove. A second problem is that when $k = 1$, the nodes which should be removed and some nodes in the base of G (which should not be removed) belong to the same node sequence (this can be seen in Fig. 3, since, when $k = 1$, the node labeled $k - 1$ and the node labeled σ in the figure are the same node).

[2] Because Algorithm 1 returns a standard loopless graph and $NL(G, d)$ is not always standard, the graph returned may be slightly different from $NL(G, d)$, but, using two strings from the sample, this problem can easily be solved.

Despite these problems, our algorithm always removes the last node sequence from each of the functor paths of the graph $NL(G,d)$. Note that this results in the same graph for every value of d and therefore we can denote the resulting (edge structure) graph by $\mathcal{B}(G)$. When $1 < k$, $\mathcal{B}(G)$ is exactly the base of G. When $k = 1$, however, too many nodes are removed by the algorithm and, as a result, $\mathcal{B}(G)$ is only the bottom part of the base of G. The missing part, though, is not too large. It is only the two last nodes sequences of the base of G.

Let τ_0 and σ_0 be the top nodes of $\mathcal{B}(G)$. Since string letter counts do not depend on the slashes assigned to a graph, it is possible to calculate, from the edge structure $\mathcal{B}(G)$, the letter counts $[s_{\tau_0}]$ and $[s_{\sigma_0}]$ of the strings s_{τ_0} and s_{σ_0} which are generated at τ_0 and σ_0 (that is, generated after we assign slashes to the edge structure).

In case $1 < k$, the strings s_{τ_0} and s_{σ_0} are exactly the strings s_τ and s_σ. When $k = 1$, because s_{τ_0} and s_{σ_0} are generated lower down in the base of G than s_τ and s_σ, the strings s_τ and s_σ must be concatenations of copies of s_{τ_0} and s_{σ_0}. In either case, every string in $L(G)$ must be a concatenation of copies of the strings s_{τ_0} and s_{σ_0}.

Finally, we note that when the loop length of G is $1 < k$, the length of the node sequence removed above the node σ is $k - 1$. Therefore, once we deduce that indeed $1 < k$, we can immediately deduce the number k.

Extracting s_{τ_0} and s_{σ_0} from the Prefixes of Sample Strings. Let $s \in L(G)$ be a string in the sample available to the learner. The string s is a concatenation of copies of s_{τ_0} and s_{σ_0}. Therefore, s must have at least one of the strings s_{τ_0} or s_{σ_0} as a prefix. It is not known, however, which of the two strings is such a prefix and at which end (left or right) of the string s. Knowing $[s_{\tau_0}]$ and $[s_{\sigma_0}]$, we know not only the lengths of the prefixes we are looking for, but also their letter counts. However, it may happen that a string s has a prefix with letter count $[s_{\tau_0}]$, but that this prefix is not equal to s_{τ_0}. The same can happen with s_{σ_0}.

By looking at Fig. 3, one sees that s_{τ_0} is a concatenation of several copies of s_{σ_0} with a single copy of some other string s_{π_0} which is generated at some node in the base of G. It turns out that this fact guarantees that *at least* one of the three methods described in the next algorithm must succeed in finding at least one of the strings s_{τ_0} or s_{σ_0}:

Algorithm 2. *Given is a string $s \in L(G)$ and the letter counts $[s_{\tau_0}]$ and $[s_{\sigma_0}]$ deduced for G.*

1. *If the left and right prefixes of s of length $|s_{\tau_0}|$ are identical, then this is the string s_{τ_0}.*
2. *If the left and right prefixes of s of length $|s_{\sigma_0}|$ are identical, then this is the string s_{σ_0}.*
3. *If the letter counts of the left and right prefixes of s of length $|s_{\tau_0}|$ are different, then one of these letter counts must be equal to $[s_{\tau_0}]$. The corresponding prefix is s_{τ_0}. The prefix of length $|s_{\sigma_0}|$ at the other end of s must be the string s_{σ_0}.*

Example 1. Let G be a graph with loop length 2 which has $s_{\sigma_0} = s_\sigma = a$, $s_{\pi_0} = s_\pi = b$ and $s_{\tau_0} = s_\tau = s_\sigma s_\sigma s_\pi = aba$ (it is simple to construct the appropriate base for G). Assume both slashes on the loop of G are forward slashes (/). The language $L(G)$ then contains the two strings $s_\sigma s_\tau s_\tau = aabaaba$ and $s_\sigma(s_\sigma s_\tau s_\tau)s_\tau = aaabaabaaba$. The $|s_{\tau_0}|$-length left and right prefixes of the first string are aab and aba. Both have letter count $[s_{\tau_0}] = [2, 1]$ and therefore the algorithm cannot determine which of them is the string s_{τ_0}. For the second string, the prefixes are aaa and aba. Since aaa does not have letter count $[s_{\tau_0}]$, the algorithm can determine that $s_{\tau_0} = aba$. For both strings the algorithm can determine s_{σ_0} since $|s_{\sigma_0}| = 1$ and all prefixes of that length are the single letter a.

3.3 Inferring the Loop Structure

The loop of the graph G generates strings by concatenating copies of s_τ and s_σ. To infer the structure of the loop, we must first parse strings in the sample as concatenations of s_{τ_0} and s_{σ_0} (which are either equal to s_τ and s_σ or substrings of s_τ and s_σ).

Parsing Strings. In order to parse a sample string s as a concatenation of s_{τ_0} and s_{σ_0}, it is enough to know one of the two strings s_{τ_0} or s_{σ_0}. ¿From now on, we assume the learner found s_{τ_0}. The treatment of the case in which the learner discovers s_{σ_0} is similar, in principle, though some of the technical details differ.

To parse a sample string s, the learner simply tries to match s_{τ_0} with segments of s. This begins at one end (left or right) of s. Each time the compared segment of s is identical to s_{τ_0}, that segment is marked as u_τ. When the match fails, the learner assumes an s_{σ_0} was reached and marks the appropriate segment (of known length $|s_{\sigma_0}|$) as u_σ. The process then continues to the next segment. The segments marked u_τ are clearly identical to s_{τ_0}. It is less obvious, but true, that the segments marked u_σ are always identical to each other and have the same letter count as $[s_{\sigma_0}]$. However, depending on the direction of the parse (from the right or left), different parses may result and the string marked by u_σ may differ, as in the following example, where $s_{\tau_0} = baaba$. The parse from the left is given above the string and the parse from the right is given below the string:

$$\begin{array}{ccccccccccccc} & u_\tau & & \overbrace{}^{u_\tau} & & u_\sigma & & \overbrace{}^{u_\tau} & & u_\sigma & \\ baaba & & baa & & ba & & aba & & baa & & ba & & aba \\ & u_\tau & & u_\sigma & & \underbrace{}_{u_\tau} & & u_\sigma & & \underbrace{}_{u_\tau} & \end{array}$$

Whether the two parses are identical or different is a property of the strings s_{τ_0} and s_{σ_0}, and therefore is the same for all strings in $L(G)$. The two parses can differ only by a "shift" of one u_τ. When different, the right to left parse can be transformed into the left to right parse by removing one u_τ from the right end and attaching it at the left end (as in the example). Note that at least one parse must be a *correct* parse, which reflects the way the string was generated by the graph.

Completing the Language Inference. Let l and r be the number of backslashes (\) and forward slashes (/), respectively, assigned to the loop of a graph G (so $l+r=k$). The language $L(G)$ generated by the loop of a graph G is easily seen to consist of the string s_τ and strings of the form $s_1 \dots s_l s_\sigma s_{l+1} \dots s_{l+r}$, where $s_1, \dots, s_{l+r}$ are strings in $L(G)$. We outline the way in which the numbers l and r can be deduced from the parses described in the previous section. Since the parses also give an hypothesis for the string s_{σ_0}, this completes the language inference. The uncertainties involved in the inference procedure may result in a graph being hypothesized which is slightly different from the original graph G that generated the sample. The language hypothesized, however, is always correct.

A language generated by a graph of loop length 1 must have one of the following forms (where k is the number deduced when calculating $\mathcal{B}$ in Sect. 3.2):

1. $L(G) = \{s_{\tau_0}^i (s_{\sigma_0} s_{\tau_0}^{k-1})^{q+n} s_{\sigma_0} s_{\tau_0}^{k-i} \mid 0 \le n\}$.
2. $L(G) = \{s_{\tau_0}^i s_{\sigma_0} (s_{\tau_0}^{k-1} s_{\sigma_0})^{q'} s_{\tau_0} (s_{\tau_0}^{k-1} s_{\sigma_0})^{q+n} s_{\tau_0}^{k-1-i} \mid 0 \le n\}$.
3. $L(G) = \{s_{\tau_0}^i (s_{\sigma_0} s_{\tau_0}^{k-1})^{q+n} s_{\tau_0} (s_{\sigma_0} s_{\tau_0}^{k-1})^{q'} s_{\sigma_0} s_{\tau_0}^{k-1-i} \mid 0 \le n\}$.

When all parses of strings in the sample match one of these forms, the corresponding language is hypothesized. Since no more than one of the above forms can fit the same two strings, any two strings from the sample are sufficient in order to construct such a hypothesis. Any graph of loop length greater than 1, which generates strings with such parses, generates a language which strictly contains this hypothesized language. Therefore, as long as the sample does not contain a string whose parse does not fit the loop length 1 hypothesis, this hypothesis can be safely maintained.

Only one uncertainty remains. The parameter n in the above forms is added to some constant q. For the parses of the sample strings, it is possible to determine $q+n$, but not q and n separately. The solution is to assume that the shortest string in the sample has $n=0$. This guarantees that the hypothesized language is contained in the language $L(G)$. When new strings are added to the sample, a shorter string may require the hypothesized parameter q to be decreased.

Once the sample contains strings whose parses cannot be interpreted as being generated by a loop length 1 graph, the learner can deduce that the language was generated by a graph of loop length greater than 1. This means that $s_{\tau_0} = s_\tau$ and $s_{\sigma_0} = s_\sigma$. Moreover, from the algorithm in Sect. 3.2, the learner also knows the length k of the loop.

Once k is known, there is a linear time algorithm which can determine those values of l, r which can generate the sample. For some samples there may be more than one such pair (some languages overlap). We do not give here the full algorithm, but only a partial algorithm (based on the same principles). This algorithm only finds upper bounds for the numbers l and r. To calculate an upper bound for l, the algorithm should be applied to a parse from left to right. To calculate an upper bound for r, it should be applied from right to left.

Algorithm 3. *Let k be the loop length of G. Given is a parse P of a string $s \in L(G)$. The algorithm begins at one end of P and advances one by one along the symbols of P. The bound calculated by the algorithm is returned in the variable ρ. The algorithm begins with $\rho := k$ and $\delta := 0$.*

1. *If the current symbol in the parse is u_τ, increase δ by 1 and continue to the next symbol in the parse.*
2. *If the current symbol in the parse is u_σ, set $\rho := min(\rho, \delta)$ and then $\delta := max(0, \delta - (k-1))$.*

We conclude by commenting that in every language there is a string from which even the simple Algorithm 3 can calculate the values of l and r exactly. When the left and right parses are identical, the simple string $s_\tau^l s_\sigma s_\tau^r$ will do. When the left and right parses are different, a more complicated string disambiguates the language. For example, if the correct parse is from left to right and $2 \leq l$, then the string $s_\tau^l s_\sigma s_\tau^l s_\sigma s_\tau^{2r+l-2} s_\sigma s_\tau^{2l+r-2} s_\sigma s_\tau^r s_\sigma s_\tau^r$ will always do. Applying Algorithm 3 to the incorrect parse of this string will result in the pair $l-1, r$, which does not sum together to k. This allows the learner to reject the parse.

4 Conclusion

Even the very simple class of 2-letter rigid grammars has a complex structure. While learning a grammar in this class can be done efficiently, it is not a trivial task. The learning algorithm must be carefully devised to make use of the special properties of the class of grammars being learned. Moreover, receiving the "wrong" input may delay (indefinitely) learning the correct grammar. At the same time, receiving the "right" input can lead to very fast convergence to the correct grammar.

The properties of the 2-letter rigid grammars which were used here cannot be easily extended to other classes of grammars (but do apply to any 2-letter sub-language of a rigid grammar). Even the 3-letter rigid grammars are already a more complex class. However, one may try to identify other classes of grammars which can be characterized by parameters which can be solved sequentially. These parameters may be hidden below the surface of the standard representation of categorial grammars through types.

References

1. Costa Florêncio C. *Consistent Identification in the Limit of any of the Classes k-Valued is NP-Hard*, In de Groote P., Morril G., Retoré C., Eds., Logical Aspects of Computational Linguistics, vol. 2099 of Lecture Notes in Artificial Intelligence, p. 125-138, Springer, 2001.
2. Graham R., Knuth D., Potashnik O. *Concrete Mathematics*, 2nd ed. Addison-Wesley, 1994.
3. Kanazawa M. *Learnable Classes of Categorial Grammars*, Studies in Logic Language and Information, CSLI Publications, 1998.