String Transformation Learning

Giorgio Satta
Dipartimento di Elettronica e Informatica
Università di Padova
via Gradenigo, 6/A
I-35131 Padova, Italy
satta@dei.unipd.it

John C. Henderson
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218-2694
jhndrsn@cs.jhu.edu

Abstract

String transformation systems have been introduced in (Brill, 1995) and have several applications in natural language processing. In this work we consider the computational problem of automatically learning from a given corpus the set of transformations presenting the best evidence. We introduce an original data structure and efficient algorithms that learn some families of transformations that are relevant for part-of-speech tagging and phonological rule systems. We also show that the same learning problem becomes NP-hard in cases of an unbounded use of don't care symbols in a transformation.

1 Introduction

Ordered sequences of rewriting rules are used in several applications in natural language processing, including phonological and morphological systems (Kaplan and Kay, 1994), morphological disambiguation, part-of-speech tagging and shallow syntactic parsing (Brill, 1995), (Karlsson et al., 1995). In (Brill, 1995) a learning paradigm, called error-driven learning, has been introduced for automatic induction of a specific kind of rewriting rules called transformations, and it has been shown that the achieved accuracy of the resulting transformation systems is competitive with that of existing systems.

In this work we further elaborate on the error-driven learning paradigm. Our main contribution is summarized in what follows. We consider some families of transformations and design efficient algorithms for the associated learning problem that improve existing methods. Our results are achieved by exploiting a data structure originally introduced in this work. This allows us to simultaneously represent and test the search space of all possible transformations. The transformations we investigate make use of classes of symbols, in order to generalize regularities in rule applications. We also show that when an unbounded number of these symbol classes are allowed within a transformation, then the associated learning problem becomes NP-hard.

The notation we use in the remainder of the paper is briefly introduced here. Σ denotes a fixed, finite alphabet and ε the null string. Σ^* and Σ^+ are the set of all strings and all non-null strings over Σ, respectively. Let $w \in \Sigma^*$. We denote by $|w|$ the length of w. Let $w = uxv$; u is a **prefix** and v is a **suffix** of w; when x is non-null, it is called a **factor** of w. The suffix of w of length i is denoted $\mathit{suff}_i(w)$, for $0 \leq i \leq |w|$. Assume that x is non-null, and $w = u_i x \mathit{suff}_i(w)$ for $\varphi > 0$ different values of i but not for $\varphi + 1$, or x is not a factor of w and $\varphi = 0$. Then we say that φ is the **statistic** of factor x in w.

2 The learning paradigm

The learning paradigm we adopt is called error-driven learning and has been originally proposed in (Brill, 1995) for part of speech tagging applications. We briefly introduce here the basic assumptions of the approach.

A string **transformation** is a rewriting rule denoted as $u \rightarrow v$, where u and v are strings such that $|u| = |v|$. This means that if u appears as a factor of some string w, then u should be replaced by v in w. The application of the transformation might be conditioned by the requirement that some additionally specified pattern matches some part of the string w to be rewritten.

We now describe how transformations can be automatically learned. A pair of strings (w, w') is an **aligned pair** if $|w| = |w'|$. When $w = ux\mathit{suff}_i(w)$, $w' = u'x'\mathit{suff}_i(w')$ and $|x| = |x'|$, we say that factors x and x' occur at **aligned positions** within (w, w'). A multi-set of aligned pairs is called an **aligned corpus**. Let (w, w') be an aligned pair and let τ be some transformation of the form $u \rightarrow v$. The **positive evidence** of τ (w.r.t. (w, w')) is the number of different positions at which factors u and v are aligned within (w, w'). The **negative evidence** of τ (w.r.t. w, w') is the number of different positions at which factors u and u are aligned within

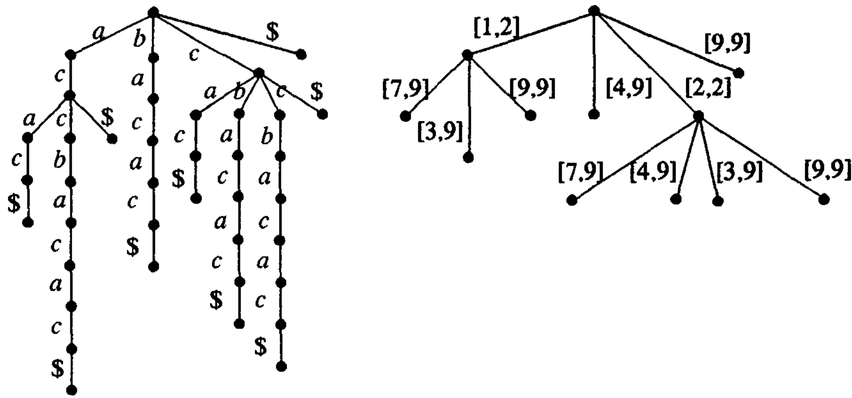

Figure 1: Trie and suffix tree for string $w = accbacac\$$. Pair $[i, j]$ denotes the factor of w starting at position i and ending at position j (hence $[1, 2]$ denotes ac).

(w, w'). Intuitively speaking, positive (negative) evidence is a count of how many times we will do well (badly, respectively) when using τ on w in trying to get w'. The **score** associated with τ is the difference between the positive evidence and the negative evidence of τ. This extends to an aligned corpus in the obvious way. We are interested in the set of transformations that are associated with the highest score in a given aligned corpus, and will develop algorithms to find such a set in the next sections.

3 Data Structures

This section introduces two data structures that are basic to the development of the algorithms presented in this paper.

3.1 Suffix trees

We briefly present here a data structure that is well known in the text processing literature; the reader is referred to (Crochemore and Rytter, 1994) and (Apostolico, 1985) for definitions and further references.

Let w be some non-null string. Throughout the paper we assume that the rightmost symbol of w is an end-marker not found at any other position in the string. The **suffix tree** associated with w is a "compressed" trie of all strings $suff_i(w)$, $1 \leq i \leq |w|$. Edges are labeled by factors of w which are encoded by means of two natural numbers denoting endpoints in the string. An example is reported in Figure 1. An **implicit node** is a node not explicitly represented in the suffix tree, that splits the label of some edge at a given position. (Each implicit node corresponds to some node in the original trie having only one child.) We denote by $parent(p)$ the parent node of (implicit) node p and by $label(p, q)$ the label of the edge spanning (implicit) nodes p and q. Throughout the paper, we take the dominance relation between nodes to be reflexive, unless we write **proper** dominance. We also say that implicit node q **immediately** dominates node p if q splits the arc between $parent(p)$ and p. Of main interest here are the following properties of suffix trees:

- if node p has children $p_1, \ldots, p_d$, then $d \geq 2$ and strings $label(p, p_i)$ differ one from the other at the leftmost symbol;
- all and only the factors of w are represented by paths from the root to some (implicit) node;
- the statistic of factor u of w is the number of leaves dominated by the (implicit) node ending the path representing u.

In the remainder of the paper, we sometimes identify an (implicit) node of a suffix tree with the factor represented by the path from the root to that node.

The suffix tree and the statistics of all factors of w can be constructed/computed in time $O(|w|)$, as reported in (Weiner, 1973) and (McCreight, 1976). McCreight algorithm uses two basic functions to scan paths in the suffix tree under construction. These functions are briefly introduced here and will be exploited in the next subsection. Below, p is a node in a tree and u is a non-null string.

function *Slow_scan*(p, u): Starting at p, scan u symbol by symbol. Return the (implicit) node corresponding to the last matching symbol.

The next function runs faster than *Slow_scan*, and can be used whenever we already know that u is an (implicit) node in the tree (u completely matches some path in the tree).

function *Fast_scan*(p, u): Starting at p, scan u by iteratively (i) finding the edge between the current node and one of its children, that has the same first symbol as the suffix of u yet to be scanned, and (ii) skipping a prefix of u equal to the length of the selected edge label. Return the (implicit) node u.

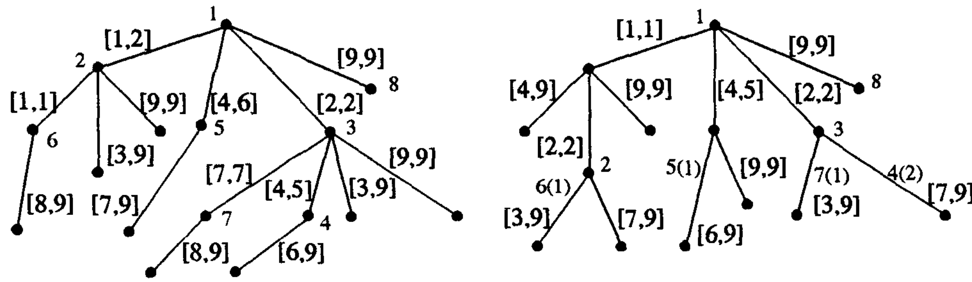

Figure 2: Suffix tree alignment for strings $w = accbacac\$$, $w' = acabacba\$$ and the identity homomorphism $h(a) = a$, $h(b) = b$, $h(c) = c$. Each a-link is denoted by indexing the incident nodes with the same integer number; if the incident node is an implicit node, then we add between parentheses the relative position w.r.t. the arc label.

From each node au in the suffix tree, au some factor, McCreight's algorithm creates a pointer, called an **s-link**, to node u which necessarily exists in the suffix tree. We write $q = \textit{s-link}(p)$ if there is an s-link from p to q.

3.2 Suffix tree alignment

In the next section each transformation will be associated with several strings. Given an input text, we will compute transformation scores by computing statistics of these strings. This can easily be done using suffix trees, *and* by pairing statistics corresponding to the same transformation. The latter task can be done using the data structure originally introduced here.

A total function $h : \Sigma \to \Sigma'$, Σ and Σ' two alphabets, is called a (restricted) **homomorphism**. We extend h to a string function in the usual way by posing $h(\varepsilon) = \varepsilon$ and $h(au) = h(a)h(u)$, $a \in \Sigma$ and $u \in \Sigma^*$. Given $w, w' \in \Sigma^+$, we need to pair each factor u of w with factor $h(u)$ possibly occurring in w'. To solve this problem, we construct the suffix trees T, T' for w, w', respectively. Then we establish an **a-link** (a pointer) from each node u of T, u some factor, to the (implicit) node $h(u)$ of T', if $h(u)$ exists. Furthermore, if factor ua with $a \in \Sigma$ is an (implicit) node of T such that $h(u)$ but not $h(ua)$ are (implicit) nodes of T', we create node u in T (if u was an implicit node) and establish an a-link from u to (implicit) node $h(u)$ of T'. Note that the total number of a-links is $O(|w|)$. The resulting data structure is called here **suffix tree alignment**. An example is reported in Figure 2.

We now specify a method to compute suffix tree alignments. In what follows p, p' are tree nodes and u is a non-null string. Crucially, we assume we can access the s-links of T and T'. Paths u and v in T and T', respectively, are aligned if $v = h(u)$. The next two functions are used to move a-links up and down two aligned paths.

function *Move_link_down*(p, p', u): Starting at p and p', simultaneously scan u and $h(u)$, respectively, using function *Slow_scan*. Stop as soon as a symbol is not matched. At each encountered node of T and at the (implicit) node of T corresponding to the last successful match, create an a-link to the paired (implicit) node of T'. Return the pair of nodes in the lastly created a-link along with the length of the successfully matched prefix of u.

In the next function, we use function *Fast_scan* introduced in Section 3.1, but we run it upward the tree (with the obvious modifications).

function *Move_link_up*(p, p'): Starting at p and p', simultaneously scan the paths to the roots of T and T', respectively, using function *Fast_scan*. Stop as soon as a node of T is encountered that already has an a-link. At each encountered node of T create an a-link to the paired (implicit) node of T'.

We also need a function that "shifts" a-links to a new pair of aligned paths. This is done using s-links. The next auxiliary function takes care of those (implicit) nodes for which the s-link is missing. (This is the case for implicit nodes of T' and for some nodes of T that have been newly created.) We rest on the property that the parent node of any such (implicit) node always has an s-link, when it differs from the root.

function *Up_link_down*(p): If $\textit{s-link}(p)$ is defined then return $\textit{s-link}(p)$. Else, let $p_1 = \textit{parent}(p)$. If p_1 is not the root node, let $p_2 = \textit{s-link}(p_1)$ and return (implicit) node $\textit{Fast_scan}(p_2, \textit{label}(p_1, p))$. If p_1 is the root node, return (implicit) node $\textit{Fast_scan}(p_1, \textit{suff}_{|\textit{label}(p_1,p)|-1}(\textit{label}(p_1, p))$.

function *Shift_link*(p, p'): $p_1 = \textit{Up_link_down}(p)$, $p'_1 = \textit{Up_link_down}(p')$. Return (p_1, p'_1).

We can now present the algorithm for the construction of suffix tree alignments.

Algorithm 1 Let T and T' be the suffix trees for strings w and w', respectively:

$(b_{|w|}, b'_{|w|}, d) \leftarrow \textit{Move_link_down}($root of T,

i	sb_i	b_i	a-link
9	–	ac	1, 2
8	c	c	–
7	ε	cba	4
6	ba	bac	5
5	ac	aca	6
4	ca	ca	7
3	a	ac	–
2	c	c	8
1	ε	\$	–

Figure 3: The table reports the values of sb_i, b_i and the established a-links at each iteration of Algorithm 1, when constructing the suffix tree alignment in Figure 2. To denote a-links we use the same integer numbers as in Figure 2.

```
    root of T', suff_|w|(w))
for i from |w| − 1 downto 1 do begin
    (sb_i, sb'_i) ← Shift_link(b_{i+1}, b'_{i+1})
    Move_link_up(sb_i, sb'_i)
    (b_i, b'_i, dd) ← Move_link_down(sb_i, sb'_i,
        suff_{|w|−d}(w))
    d ← d + dd
    end
end
```

In Figure 3 a sample run of Algorithm 1 is schematically represented.

In the next section we use the following properties of Algorithm 1:

- after T and T' have been processed, for every node p of T representing factor u of w, (implicit) node *a-link*(p) of T' is defined if and only if *a-link*(p) represents factor $h(u)$ of w';
- the algorithm can be executed in time $O(|w| + |w'|)$.

The first property above can be proved as follows. For $1 \leq i \leq |w|$, b_i in Algorithm 1 is (the node representing) the longest prefix of $suff_i(w)$ such that $h(b_i)$ is an (implicit) node of T' (is a factor of w'). This can be proved by induction on $|w| - i$, using the definition of *Move_link_down* and of s-link. We then observe that, if u is a node of T, then factor u is a prefix of some $suff_i(w)$ and either u dominates b_i or b_i properly dominates u in T. If u dominates b_i, then $h(u)$ must be an (implicit) node of T'. In this case an a-link is established from u to $h(u)$ by *Move_link_up* or *Move_link_down*, depending on whether u dominates or is dominated by sb_i in T. If b_i properly dominates u, $h(u)$ does not occur in w'. In this case, node u is never reached by the algorithm and no a-link is established for this node.

The proof of the linear time result is rather long, we only give an outline here. The interesting case is the function *Shift_link*, which is executed $|w| - 1$ times by the algorithm. When executed once on nodes p and p', *Shift_link* uses time $O(1)$ if *s-link*(p) and *s-link*(p') are both defined. In all other cases, it uses an amount of time proportional to the number of (implicit) nodes visited by function *Fast_scan*, which is called through function *Up_link_down*. We use an amortization technique and charge a constant amount of time to the symbols in w and w', for each node visited in this way. Consider the execution of *Shift_link*(b_{i+1}, b'_{i+1}) for some i, $1 \leq i \leq |w| - 1$. Assume that, correspondingly, *Fast_scan* visits nodes $u_1, \ldots, u_d$ of T in this order, with $d \geq 1$ and each u_j some factor of w. Then we have that each u_j is a (proper) prefix of u_{j+1}, and $u_d = sb_i$. For each u_j, $1 \leq j \leq d - 1$, we charge a constant amount of time to the symbol in w "corresponding" to the last symbol of u_j. The visit to u_d, on the other hand, is charged to the ith symbol of w. (Note that charging the visit to u_d to the symbol in w "corresponding" to the last symbol of u_d does not work, since in the case of $sb_i = b_i$ the same symbol would be charged again at the next iteration of the for-cycle.) It is not difficult to see that, in this way, each symbol of w is charged at most once. A similar argument works for visits to nodes of T' by *Fast_scan*, which are charged to symbols of w'. This shows that the time used by all executions of *Shift_link* is $O(|w| + |w'|)$.

Suffix trees and suffix tree alignments can be generalized to finite multi-sets of strings, each string ending with the same end-marker not found at any other position. In this case each leaf holds a record, called **count**, of the number of times the corresponding suffix appears in the entire multi-set, which will be propagated appropriately when computing factor statistic. Most important here, all of the above results still hold for these generalizations. In the next section, we will deal with the multi-set case.

4 Transformation learning

This section deals with the computational problem of learning string transformations from an aligned corpus. We show that some families of transformations can be efficiently learned exploiting the data structures of Section 3. We also consider more general kinds of transformations and show that for this class the learning problem is NP-hard.

4.1 Data representation

We introduce a representation of aligned corpora that reduces the problem of computing the positive/negative evidence of transformations to the problem of computing factor statistics.

Let (w, w') be an aligned pair, $w = a_1 \cdots a_n$ and $w' = a'_1 \cdots a'_n$, with $a_i \in \Sigma$ for $1 \leq i \leq n$, and $n \geq 1$. We define

$$w \times w' = (a_1, a'_1) \cdots (a_n, a'_n). \qquad (1)$$

Note that $w \times w'$ is a string over the new alphabet $\Sigma \times \Sigma$. Let $N \geq 1$ and let $L = \{(w_1, w'_1), \ldots, (w_N, w'_N)\}$ be an aligned corpus. We represent L as a string multi-set over alphabet $\Sigma \times \Sigma$:

$$L_\times = \{w \times w' \mid (w, w') \in L\}, \qquad (2)$$

where $w \times w'$ appears in $L_\times$ as many times as (w, w') appears in L.

4.2 Learning algorithms

Let L be an aligned corpus with N aligned pairs over a fixed alphabet Σ, and let n be the length of the longest string in a pair in L. We start by considering plain transformations of the form

$$u \rightarrow v, \qquad (3)$$

where $u, v \in \Sigma^+$, $|u| = |v|$. We want to find *all* instances of strings $u, v \in \Sigma^*$ such that, in L, $u \rightarrow v$ has score greater or equal than the score of any other transformation. Existing methods for this problem are data-driven. They consider all pairs of factors (with lengths bounded by n) occurring at aligned positions within some pair in L, and update the positive and the negative evidence of the associated transformations. They thus consider $O(Nn^2)$ factor pairs, where each pair takes time $O(n)$ to be read/stored. We conclude that these methods use an amount of time $O(Nn^3)$. We can improve on this by using suffix tree alignments.

Let $L_\times$ be defined as in (2) and let $h_1 : (\Sigma \times \Sigma) \rightarrow (\Sigma \times \Sigma)$ be the homomorphism specified as:

$$h_1((a, b)) = (a, a).$$

Recall that each suffix of a multi-set of strings is represented by a leaf in the associated suffix-tree, because of the use of the end-marker, and that each leaf stores the count of the occurrences of the corresponding suffix in the source multi-set. We schematically specify our first learning algorithm below.

Algorithm 2

Step 1: construct two copies $T_\times$ and $T'_\times$ of the suffix tree associated with $L_\times$ and align them using h_1;
Step 2: visit trees $T_\times$ and $T'_\times$ in post-order, and annotate each node p with the number $e(p)$ computed as the sum of the counts at leaves that p dominates;
Step 3: annotate each node p of $T_\times$ with the score $e(p) - e(p')$, where $p' = a\text{-}link(p)$ if $a\text{-}link(p)$ is an actual node, p' is the node immediately dominated by $a\text{-}link(p)$ if $a\text{-}link(p)$ is an implicit node, and $e(p') = 0$ if $a\text{-}link(p)$ is undefined; make a list of the nodes with the highest annotated score.

Let p be a node of $T_\times$ associated with factor $u \times v$. Integer $e(p)$ computed at Step 2 is the number of times a suffix having $u \times v$ as a prefix appears in strings in $L_\times$. Thus $e(p)$ is the number of different positions at which factors u and v are aligned within $L_\times$ and hence the positive evidence of transformation $u \rightarrow v$ w.r.t. L, as defined in Section 2. Similarly, $e(p')$ is the statistic of factor $u \times u$ and hence the negative evidence of $u \rightarrow v$ (as well as the negative evidence of all transformations having u as left-hand side). It follows that Algorithm 2 records, at Step 3, the transformations having the highest score in L among all transformations represented by nodes of $T_\times$. It is not difficult to see that the remaining transformations, denoted by implicit nodes of $T_\times$, do not have score greater than the one above. The latter transformations with highest score, if any, can be easily recovered by visiting the implicit nodes that immediately dominate the nodes of $T_\times$ recorded at Step 3.

A complexity analysis of Algorithm 2 is straightforward. Step 1 can be executed in time $O(Nn)$, as discussed in Section 3. Since the size of $T_\times$ and $T'_\times$ is $O(Nn)$, all other steps can be easily executed in linear time. Hence Algorithm 2 runs in time $O(Nn)$.

We now turn to a more general kind of transformations. In several natural language processing applications it is useful to generalize over some transformations of the form in (3), by using classes of symbols in Σ. Let $t \geq 1$ and let $C_1, \ldots, C_t$ be a partition of Σ (each $C_i \neq \emptyset$). Consider $\Gamma = \{C_1, \ldots, C_t\}$ as an alphabet. We say that string $a_1 \cdots a_d \in \Sigma^+$ **matches** string $C_{i_1} \cdots C_{i_d} \in \Gamma^+$ if $a_k \in C_{i_k}$ for $1 \leq k \leq d$. We define transformations[1]

$$u\gamma \rightarrow v-, \qquad (4)$$

$u, v \in \Sigma^+$, $|u| = |v|$, $\gamma \in \Gamma^+$, and assume the following interpretation. An occurrence of string u must be rewritten to v in a text whenever u is followed by a substring matching γ. String γ is called the **right context** of the transformation. The positive evidence for such transformation is the number of positions at which factors ux and vx' are aligned within the corpus, for all possible $x, x' \in \Sigma^+$ with x matching γ. (We do not require $x = x'$, since later transformations can change the right context.) The negative evidence for the transformation is the number of positions at which factors ux and ux' are aligned within the corpus, x, x' as above.

We are not aware of any learning method for transformations of the form in (4). A naive method for this task would consider all factor pairs appearing at aligned positions in some pair in L. The left component of each factor must then be split into a string in Σ^+ and a string in Γ^+, to represent a transformation in the desired form. Overall, there are $O(Nn^3)$ possible transformations, and we need time $O(n)$ to read/store each transformation. Then the method uses an amount of time $O(Nn^4)$. Again, we can improve on this. We need a representation for right context strings. Define homomorphism $h_2 : (\Sigma \times \Sigma) \rightarrow \Gamma$ as

$$h_2((a, b)) = C, \quad a \in C.$$

[1] In generative phonology (4) is usually written as $u \rightarrow v \,/\, _\, \gamma$. Our notation can more easily be generalized, as it is needed in some transformation systems.

(h_2 is well defined since Γ is a partition of Σ.) Let also

$$L_\Gamma \quad = \quad \{h_2(w \times w') \mid w \times w' \in L_\times\},$$

where $h_2(w \times w')$ appears in L_Γ as many times as $w \times w'$ appears in $L_\times$.

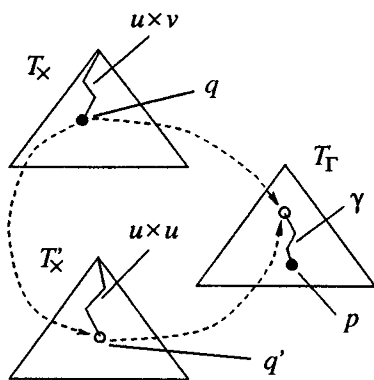

Figure 4: At Step 3 of Algorithm 3, triple (q, e, e') is inserted in $\tau(p)$ if the relations depicted above are realized, where dashed arrows denote a-links, black circles denote nodes, and white circles denote nodes that might be implicit. Integer $e > 0$ is a count of the paths from node q downward, having the form $y \times y'$ with a prefix of y matching γ. Similarly, e' is a count of the paths from node q' downward satisfying the same matching condition with γ. The matching condition is enforced by the fact that the above paths have their ending leaf nodes a-linked to a leaf node of T_Γ dominated by node p.

Below we link a suffix-tree to more than one suffix-tree. In the notation of a-links we then use a subscript indicating the suffix tree of the target node, in order to distinguish among different linkings. We now schematically specify the learning algorithm; additional computational details will be provided later in the discussion of the complexity.

Algorithm 3

Step 1: construct two copies $T_\times$ and $T'_\times$ of the suffix tree associated with $L_\times$ and construct the suffix tree T_Γ associated with L_Γ;

Step 2: align $T_\times$ with $T'_\times$ using h_1 and align the resulting suffix trees $T_\times$ and $T'_\times$ with T_Γ using h_2;

Step 3: for each node p of T_Γ, store a set $\tau(p)$ including all triples (q, e, e') such that (see Figure 4):

- q is a node of $T_\times$ such that $a\text{-}link_{T_\Gamma}(q)$ properly dominates p
- $e > 0$ is the sum of the counts at leaves of $T_\times$ dominated by q that have an a-link to a leaf of T_Γ dominated by p
- if $q' = a\text{-}link_{T'_\times}(q)$ is defined, e' is the sum of the counts at leaves of $T'_\times$ dominated by q' that have an a-link to a leaf of T_Γ dominated by p; otherwise, $e' = 0$;

Step 4: find all pairs (p, q), p a node of T_Γ and $(q, e, e') \in \tau(p)$, such that $e - e'$ is greater than or equal to any other $e_1 - e'_1$, (q_1, e_1, e'_1) in some $\tau(p_1)$.

We next show that if pair (p, q) is found at Step 4, then q represents a factor $u \times v$, p represents a factor $h_2(u \times v)\gamma$, and transformation $u\gamma \rightarrow v$ — has the highest score among all transformations represented by nodes of $T_\times$ and T_Γ. Similarly to the case of Algorithm 2, this is the highest score achieved in L, and other transformations with the same score can be obtained from some of the implicit nodes immediately dominating p and q.

Let p and q be defined as in Step 3 above. Assume that q represents a factor $u \times v$ of some string in $L_\times$ and p represents a factor $\delta\gamma \in \Gamma^*$ of some string in L_Γ, where $|\delta| = |u|$. Since $a\text{-}link_{T_\Gamma}(q)$ dominates p, we must have $h_2(u \times v) = \delta$. Consider a suffix $(u \times v)(x \times x')(y \times y')$ appearing in $\varphi > 0$ strings in $L_\times$, such that $h_2(x \times x') = \gamma$. (This means that x matches γ, and there are at least φ positions at which $u \rightarrow v$ has been applied with a right-context of γ.) We have that string $h_2((u \times v)(x \times x')(y \times y')) = \delta\gamma h_2(y \times y')$ must be a suffix of some strings in L_Γ. It follows that $(u \times v)(x \times x')(y \times y')$ is a leaf of $T_\times$ with a count of φ, $\delta\gamma h_2(y \times y')$ is a leaf of T_Γ, and there is an a-link between these two nodes. Leaf $(u \times v)(x \times x')(y \times y')$ is dominated by q, and leaf $\delta\gamma h_2(y \times y')$ is dominated by p. Then, at Step 3, integer φ is added to e. Since no condition has been imposed above on string x' and on suffix $(y \times y')$, we conclude that the final value of e must be the positive evidence of transformation $u\gamma \rightarrow v$ —. A similar argument shows that the negative evidence of this transformation is stored in e'. It then follows that, at Step 4, Algorithm 3 finds the transformations with the highest score among those represented by nodes of $T_\times$ and T_Γ.

Algorithm 3 can be executed in time $O(Nn^2)$. We only outline a proof of this property here, by focusing on Step 3. To execute this step we visit T_Γ in post order. At leaf node p, we consider the set $F(p)$ of all leaves q of $T_\times$ such that $p = a\text{-}link_{T_\times}(q)$, and the set $F'(p)$ of all leaves q' of $T'_\times$ such that $p = a\text{-}link_{T_\times}(q')$. For each (implicit) node of $T'_\times$ that dominates some node in $F'(p)$ and that is the target of some a-link (from some source node of $T_\times$), we record the sum of the counts of the dominated nodes in $F'(p)$. This can be done in time $O(|F'(p)|\, n)$. For each node q of $T_\times$ dominating some node in $F(p)$, we store in $\tau(p)$ the triple (q, e, e'), since $a\text{-}link_{T_\Gamma}(q)$ necessarily dominates p. We let $e > 0$ be the sum of the counts of the dominated nodes in $F(p)$, and let e' be the value retrieved from the a-link to $T'_\times$, if any. This takes time $O(|F(p)|\, n)$. When p ranges over the leaves of

T_Γ, we have $\sum_p |F(p)| = \sum_p |F'(p)| = O(Nn)$. We then conclude that sets $\tau(p)$ for all leaves p of T_Γ can be computed in time $O(Nn^2)$. At internal node p with children p_i, $1 \leq i \leq d$, $d > 1$, we assume that sets $\tau(p_i)$'s have already been computed. Assume that for some i we have $(q, e_i, e'_i) \in \tau(p_i)$ and $a\text{-}link_{T_\Gamma}(q)$ does not immediately dominate p_i. If $(q, e, e') \in \tau(p)$, we add e_i, e'_i to e, e', respectively; otherwise, we insert (q, e_i, e'_i) in $\tau(p)$. We can then compute sets $\tau(p)$ for all internal nodes p of T_Γ using an amount of time $\sum_p |\tau(p)| = O(Nn^2)$.

4.3 General transformations

We have mentioned that the introduction of classes of alphabet symbols allows abstraction over plain transformations that is of interest to natural language applications. We generalize here transformations in (4) by letting γ be a string over $\Sigma \cup \Gamma$. More precisely, we assume γ has the form:

$$\gamma = u_0 \alpha_1 u_1 \cdots u_{d-1} \alpha_d u_d, \tag{5}$$

where $u_0, u_d \in \Sigma^*$, $u_i \in \Sigma^+$ and $\alpha_j \in \Gamma^+$ for $1 \leq i \leq d-1$ and $1 \leq j \leq d$, and $d \geq 1$. The notion of matching previously defined is now extended in such a way that, for $a, b \in \Sigma$, a matches b if $a = b$. Then the interpretation of the resulting transformation is the usual one. The parameter d in (5) is called the number of **alternations** of the transformation. We have established the following results:

- transformations with a bounded number of alternations can be learned in polynomial time;
- learning transformations with an unbounded number of alternations is NP-hard.

Again, we only give an outline of the proof below.

The first result is easy to show, by observing that in an aligned corpus there are polynomially many occurrences of transformations with a bounded number of alternations. The second result holds even if we restrict ourselves to $|\Sigma| = 2$ and $|\Gamma| = 1$, that is if we use a **don't care** symbol. Here we introduce a decision problem associated with the optimization problem of learning the transformations with the highest score, and outline an NP-completeness proof.

TRANSFORMATION SCORING (TS)
Instance: $\langle L, K \rangle$, with L an aligned corpus, K a positive integer.
Question: Is there a transformation that has score greater than or equal to K w.r.t. L?

Membership in NP is easy to establish for TS. To show NP-hardness, we consider the CLIQUE decision problem for undirected, simple, connected graphs and transform such a problem to the TS problem. (The NP-completeness for the used restriction of the CLIQUE problem (Garey and Johnson, 1979) is easy to establish.) Let $\langle G, K' \rangle$ be an instance of the CLIQUE problem as above, $G = (V, E)$ and $K' > 0$. Without loss of generality, we assume that $V = \{1, 2, \ldots, q\}$. Let $\Sigma = \{a, b\}$; we construct an instance of the TS problem $\langle L, K \rangle$ over Σ as follows. For each $\{i, j\} \in V$ with $i < j$ let

$$w_{i,j} = a^{i-1} b a^{j-i-1} b a^{q-j}. \tag{6}$$

We add to the aligned corpus L:

1. one instance of pair $p_{i,j} = (aw_{i,j}, bw_{i,j})$ for each $i < j$, $\{i, j\} \in E$;
2. q^2 instances of pair $\overline{p}_{i,j} = (aw_{i,j}, aw_{i,j})$ for each $i, j \in V$ with $i < j$ and $\{i, j\} \notin E$;
3. q^2 instances of pair $p_a = (aa^q, ba^q)$.

Also, we set $K = q^2 + \binom{K'}{2}$. The above instance of TS can easily be constructed in polynomial deterministic time with respect to the length of $\langle G, K' \rangle$.

It is easy to show that when $\langle G, K' \rangle$ is a positive instance of the source problem, then the corresponding instance of TS is satisfied by at least one transformation. Assume now that there exists a transformation τ having score greater equal than $K > 0$, w.r.t. L. Since the replacement of a with b is the only rewriting that appears in pairs of L, τ must have the form $a\gamma \rightarrow b$ —. If γ includes some occurrence of b, then τ cannot match p_a and the positive evidence of τ will not exceed $|E| \leq \binom{q}{2} < K$, contrary to our assumption. We then conclude that γ has the form (? denotes the don't care symbol):

$$a^{j_1-1} ? a^{j_2-j_1-1} ? \ldots ? a^{q'-j_d},$$

where $V'' = \{j_1, \ldots, j_d\} \subseteq V$, $d \geq 0$ and $q' \leq q$. If there exists $i, j \in V''$ such that $\{i, j\} \notin E$, then τ would match some pair $\overline{p}_{i,j} \in L$ and it would have negative evidence smaller or equal than q^2. Since the positive evidence of τ cannot exceed $q^2 + |E|$, τ would have a score not exceeding $|E| \leq \binom{q}{2} < K$, contrary to our assumption. Then τ matches no pair $\overline{p}_{i,j} \in L$ and, for each $i, j \in V''$, we have $\{i, j\} \in E$. Since $K - q^2 = \binom{K'}{2}$, at least $\binom{K'}{2}$ pairs $p_{i,j} \in L$ are matched by τ. We therefore conclude that $d \geq K'$ and that V'' is a clique in G of size greater equal than K'. This concludes our outline of the proof.

5 Concluding remarks

With some minor technical changes to function *Up_link_down*, we can align a suffix tree with itself (w.r.t. a given homomorphism). In this way we improve space performance of Algorithms 2 and 3, avoiding the construction of two copies of the same suffix tree. Algorithm 3 can trivially be adapted to learn transformations in (4) where a left context is specified in place of a right context. The algorithm can also be used to learn traditional phonological rules of the form $a \rightarrow b \ / \ _\gamma$, where a, b are single phonemes and γ is a sequence over $\{\text{C}, \text{V}\}$, the classes of consonants and vowels. In this case the

algorithm runs in time $O(Nn)$ (for fixed alphabet). We leave it as an open problem whether rules of the form in (4) can be learned in linear time.

We have been concerned with learning the best transformations that should be applied at a given step. An ordered sequence of transformations can be learned by iteratively learning a single transformation and by processing the aligned corpus with the transformation just learned (Brill, 1995). Dynamic techniques for processing the aligned corpus were first proposed in (Ramshaw and Marcus, 1996) to re-edit the corpus only where needed. Those authors report that this is not space efficient if transformation learning is done by independently testing all possible transformations in the search space (as in (Brill, 1995)). The suffix tree alignment data structure allows simultaneous scoring for all transformations. We can now take advantage of this and design dynamical algorithms that re-edit a suffix tree alignment only where needed, on the line of a similar method for suffix trees in (McCreight, 1976).

An alternative data structure to suffix trees for the representations of string factors, called DAWG, has been presented in (Blumer et al., 1985). We point out here that, because a DAWG is an acyclic graph rather than a tree, straightforward ways of defining alignment between two DAWGs results in a quadratic number of a-links, making DAWGs much less attractive than suffix trees for factor alignment. We believe that suffix tree alignments are a very flexible data structure, and that other transformations could be efficiently learned using these structures.

We do not regard the result in Section 4.3 as a negative one, since general transformations specified as in (5) seem too powerful for the proposed applications in natural language processing, and learning might result in corpus overtraining.

Other than transformation based systems the methods presented in this paper can be used for learning rules of constraint grammars (Karlsson et al., 1995), phonological rule systems as in (Kaplan and Kay, 1994), and in general those grammatical systems using constraints represented by means of rewriting rules. This is the case whenever we can encode the alphabet of the corpus in such a way that alignment is possible.

Acknowledgements

Part of the present research was done while the first author was visiting the Center for Language and Speech Processing, Johns Hopkins University, Baltimore, MD. The second author is a member of the Center for Language and Speech Processing. This work was funded in part by NSF grant IRI-9502312. The authors are indebted to Eric Brill for technical discussions on topics related to this paper.

References

Apostolico, A. 1985. The myriad virtues of suffix trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12. Springer-Verlag, Berlin, Germany, pages 85-96. NATO Advanced Science Institutes, Seires F.

Blumer, A., J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. 1985. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55.

Brill, E. 1995. Transformation-based error-driven learning and natural language processing: A case study in part of speech tagging. *Computational Linguistics.*

Crochemore, M. and W. Rytter. 1994. *Text Algorithms.* Oxford University Press, Oxford, UK.

Garey, M. R. and D. S. Johnson. 1979. *Computers and Intractability.* Freeman and Co., New York, NY.

Kaplan, R. M. and M. Kay. 1994. Regular models of phonological rule sistems. *Computational Linguistics*, 20(3):331–378.

Karlsson, F., A. Voutilainen, J. Heikkilä, and A. Anttila. 1995. *Constraint Grammar. A Language Independent System for Parsing Unrestricted Text.* Mouton de Gruyter.

McCreight, E. M. 1976. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272.

Ramshaw, L. and M. P. Marcus. 1996. Exploring the nature of transformation-based learning. In J. Klavans and P. Resnik, editors, *The Balancing Act—Combining Symbolic and Statistical Approaches to Language.* The MIT Press, Cambridge, MA, pages 135–156.

Weiner, P. 1973. Linear pattern-matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, New York, NY. Institute of Electrical and Electronics Engineers.