Efficient Transformation-Based Parsing

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

Eric Brill
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218-2694
brill@cs.jhu.edu

Abstract

In transformation-based parsing, a finite sequence of tree rewriting rules are checked for application to an input structure. Since in practice only a small percentage of rules are applied to any particular structure, the naive parsing algorithm is rather inefficient. We exploit this sparseness in rule applications to derive an algorithm two to three orders of magnitude faster than the standard parsing algorithm.

1 Introduction

The idea of using transformational rules in natural language analysis dates back at least to Chomsky, who attempted to define a set of transformations that would apply to a word sequence to map it from deep structure to surface structure (see (Chomsky, 1965)). Transformations have also been used in much of generative phonology to capture contextual variants in pronunciation, starting with (Chomsky and Halle, 1968). More recently, transformations have been applied to a diverse set of problems, including part of speech tagging, pronunciation network creation, prepositional phrase attachment disambiguation, and parsing, under the paradigm of transformation-based error-driven learning (see (Brill, 1993; Brill, 1995) and (Brill and Resnik, 1994)). In this paradigm, rules can be learned automatically from a training corpus, instead of being written by hand.

Transformation-based systems are typically deterministic. Each rule in an ordered list of rules is applied once wherever it can apply, then is discarded, and the next rule is processed until the last rule in the list has been processed. Since for each rule the application algorithm must check for a matching at all possible sites to see whether the rule can apply, these systems run in $O(\pi\rho n)$ time, where π is the number of rules, ρ is the cost of a single rule matching, and n is the size of the input structure. While this results in fast processing, it is possible to create much faster systems. In (Roche and Schabes, 1995), a method is described for converting a list of transformations that operates on strings into a deterministic finite state transducer, resulting in an optimal tagger in the sense that tagging requires only one state transition per word, giving a linear time tagger whose run-time is independent of the number and size of rules.

In this paper we consider transformation-based parsing, introduced in (Brill, 1993), and we improve upon the $O(\pi\rho n)$ time upper bound. In transformation-based parsing, an ordered sequence of tree-rewriting rules (tree transformations) are applied to an initial parse structure for an input sentence, to derive the final parse structure. We observe that in most transformation-based parsers, only a small percentage of rules are actually applied, for any particular input sentence. For example, in an application of the transformation-based parser described in (Brill, 1993), $\pi = 300$ rules were learned, to be applied at each node of the initial parse structure, but the average number of rules that are successfully applied at each node is only about one. So a lot of time is spent testing whether the conditions are met for applying a transformation and finding out that they are not met. This paper presents an original algorithm for transformation-based parsing working in $O(\rho t \log(t))$ time, where t is the total number of rules applied for an input sentence. Since in practical cases t is smaller than n and we can neglect the $\log(t)$ factor, we have achieved a time improvement of a factor of π. We emphasize that π can be several hundreds large in actual systems where transformations are lexicalized.

Our result is achieved by preprocessing the transformation list, deriving a finite state, deterministic tree automaton. The algorithm then exploits the automaton in a way that obviates the need for checking the conditions of a rule when that rule will not apply, thereby greatly improving parsing run-time over the straightforward parsing algorithm. In a sense, our algorithm spends time only with rules that can be applied, as if it knew in advance which rules cannot be applied during the parsing process.

The remainder of this paper is organized as follows. In Section 2 we introduce some preliminaries, and in Section 3 we provide a representation of transformations that uses finite state, deterministic tree automata. Our algorithm is then specified in Section 4. Finally, in Section 5 we discuss related work in the existing literature.

2 Preliminaries

We review in the following subsections some terminology that is used throughout this paper.

2.1 Trees

We consider ordered trees whose nodes are assigned labels over some finite alphabet Σ; this set is denoted as Σ^T. Let $T \in \Sigma^T$. A node of T is called **leftmost** if it does not have any left sibling (a root node is a leftmost node). The **height** of T is the length of a longest path from the root to one of its leaves (a tree composed of a single node has height zero). We define $|T|$ as the number of nodes in T. A tree $T \in \Sigma^T$ is denoted as A if it consists of a single leaf node labeled by A, and as $A(T_1, T_2, \ldots, T_d)$, $d \geq 1$, if T has root labeled by A with d (ordered) children denoted by $T_1, \ldots, T_d$. Sometimes in the examples we draw trees in the usual way, indicating each node with its label.

What follows is standard terminology from the tree pattern matching literature, with the simplification that we do not use variable terms. See (Hoffmann and O'Donnell, 1982) for general definitions. Let n be a node of T. We say that a tree S **matches** T at n if there exists a mapping from the nodes of S to the nodes of T, such that the following conditions are all satisfied: (i) if n' maps to n'', then n' and n'' have the same label; (ii) the root of S maps to n; and (iii) if n' maps to n'' and n' is not a leaf in S, then n' and n'' have the same degree and the i-th child of n' maps to the i-th child of n''. We say that T and S are **equivalent** if they match each other at the respective root nodes. In what follows trees that are equivalent are not treated as the same object. We say that a tree T' is a **subtree** of T at n if there exists a tree S that matches T at n, and T' consists of the nodes of T that are matched by some node of S and the arcs of T between two such nodes. We also say that T' is matched by S at n. In addition, T' is a **prefix** of T if n is the root of T; T' is the **suffix** of T at n if T' contains all nodes of T dominated by n.

Example 1 Let $T = B(D, C(B(D, B), C))$ and let n be the second child of T's root. $S = C(B, C)$ matches T at n. $S' = B(D, C(B), C))$ is a prefix of S and $S'' = C(B(D, B), C)$ is the suffix of T at n. $\Box$

We now introduce a tree replacement operator that will be used throughout the paper. Let S be a subtree of T and let S' be a tree having the same number of leaves as S. Let $n_1, n_2, \ldots, n_l$ and $n'_1, n'_2, \ldots, n'_l$, $l \geq 1$, be all the leaves from left to right of S and S', respectively. We write $T[S/S']$ to denote the tree obtained by embedding S' within T in place of S, through the following steps: (i) if the root of S is the i-th child of a node n_f in T, the root of S' becomes the i-th child of n_f; and (ii) the (ordered) children of n_i in T, if any, become the children of n'_i, $1 \leq i \leq l$. The root of $T[S/S']$ is the root of T if node n_f above exists, and is the root of S' otherwise.

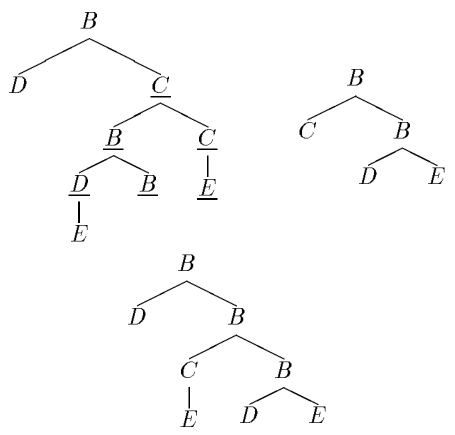

Figure 1: From left to right, top to bottom: tree T with subtree S indicated using underlined labels at its nodes; tree S' having the same number of leaves as S; tree $T[S/S']$ obtained by "replacing" S with S'.

Example 2 Figure 1 depicts trees T, S' and T' in this order. A subtree S of T is also indicated using underlined labels at nodes of T. Note that S and S' have the same number of leaves. Then we have $T' = T[S/S']$. $\Box$

2.2 Tree automata

Deterministic (bottom-up) tree automata were first introduced in (Thatcher, 1967) (called FRT there). The definition we propose here is a generalization of the canonical one to trees of any degree. Note that the transition function below is computed on a number of states that is independent of the degree of the input tree. Deterministic tree automata will be used later to implement the bottom-up tree pattern matching algorithm of (Hoffmann and O'Donnell, 1982).

Definition 1 *A* deterministic tree automaton *(*DTA*) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and δ is a transition function mapping $Q^2 \times \Sigma$ into Q.*

Informally, a DTA M walks through a tree T by visiting its nodes in post-order, one node at a time. Every time a node is read, the current state of the device is computed on the basis of the states reached upon reading the immediate left sibling and the rightmost child of the current node, if any. In this way the decision of the DTA is affected not only by the portion of the tree below the currently read node, but also by each subtree rooted in a left sibling of the current node. This is formally stated in what follows. Let $T \in \Sigma^T$ and let n be one of its nodes, labeled by a. The state **reached** by M upon reading n is recursively specified as:

$$\hat{\delta}(T, n) \quad = \quad \delta(X, X', a), \qquad (1)$$

where $X = q_0$ if n is a leftmost node, $X = \hat{\delta}(T, n')$ if n' is the immediate left sibling of n; and $X' = q_0$ if n is a leaf node, $X' = \hat{\delta}(T, n'')$ if n'' is the rightmost child of n. The tree language recognized by M is the set

$$L(M) \quad = \quad \{T \mid \hat{\delta}(T, n) \in F,\ T \in \Sigma^T,\ n \text{ the root of } T\}. \qquad (2)$$

Example 3 Consider the infinite set $L = \{B(A, C), B(A, B(A, C)), B(A, B(A, B(A, C))), \ldots\}$ consisting of all right-branching trees with internal nodes labeled by B and with strings $A^n C$, $n \geq 1$ as their yields. Let $M = (Q, \{A, B, C\}, \delta, q_0, \{q_{BC}\})$ be a DTA specified as follows: $Q = \{q_0, q_A, q_{BC}, q_{-1}\}$; $\delta(q_0, q_0, A) = q_A$, $\delta(q_A, q_0, C) = \delta(q_A, q_{BC}, B) = q_{BC}$ and q_{-1} is the value of all other entries of δ. It is not difficult to see that $L(M) = L$. $\square$

Observe that when we restrict to monadic trees, that is trees whose nodes have degree not greater than one, the above definitions correspond to the well known formalisms of deterministic finite state automata, the associated extended transition function, and the regular languages.

2.3 Transformation-based parsing

Transformation-based parsing was first introduced in (Brill, 1993). Informally, a transformation-based parser assigns to an input sentence an initial parse structure, in some uniform way. Then the parser iteratively checks an ordered sequence of tree transformations for application to the initial parse tree, in order to derive the final parse structure. This results in a deterministic, linear time parser. In order to present our algorithm, we abstract away from the assignment of the initial parse to the input, and introduce below the notion of transformation-based tree rewriting system. The formulation we give here is inspired by (Kaplan and Kay, 1994) and (Roche and Schabes, 1995). The relationship between transformation-based tree rewriting systems and standard term-rewriting systems will be discussed in the final section.

Definition 2 *A* transformation-based tree rewriting system *(TTS) is a pair $G = (\Sigma, R)$, where Σ is a finite alphabet and $R = \langle r_1, r_2, \ldots, r_\pi \rangle$, $\pi \geq 1$, is a finite sequence of tree rewriting rules having the form $Q \rightarrow Q'$, with $Q, Q' \in \Sigma^T$ and such that Q and Q' have the same number of leaves.*

If $r = (Q \rightarrow Q')$, we write lhs(r) for Q and rhs(r) for Q'. We also write lhs(R) for $\{\text{lhs}(r) \mid r \in R\}$. (Recall that we regard lhs(r_i) and lhs(r_j), $i \neq j$, as different objects, even if these trees are equivalent.) We define $|r| = |\text{lhs}(r)| + |\text{rhs}(r)|$.

The notion of transformation associated with a TTS $G = (\Sigma, R)$ is now introduced. Let $C, C' \in \Sigma^T$. For any node n of C and any rule $r = (Q \rightarrow Q')$ of G, we write

$$C \quad \stackrel{r,n}{\Longrightarrow} \quad C' \qquad (3)$$

if Q does not match C at n and $C = C'$; or if Q matches C at n and $C' = C[S/Q'_c]$, where S is the subtree of T matched by Q at n and Q'_c is a fresh copy of Q'. Let $\langle n_1, n_2, \ldots, n_t \rangle$, $t \geq 1$, be the post-ordered sequence of all nodes of C. We write

$$C \quad \stackrel{r}{\Rightarrow} \quad C' \qquad (4)$$

if $C_{i-1} \stackrel{r,n_i}{\Longrightarrow} C_i$, $1 \leq i \leq t$, $C_0 = C$ and $C_t = C'$. Finally, we define the translation induced by G on Σ^T as the map $M(G) = \{(C, C') \mid C \in \Sigma^T,\ C_{i-1} \stackrel{r_i}{\Rightarrow} C_i \text{ for } 1 \leq i \leq \pi,\ C_0 = C,\ C_\pi = C'\}$.

3 Rule representation

We develop here a representation of rule sequences that makes use of DTA and that is at the basis of the main result of this paper. Our technique improves the preprocessing phase of a bottom-up tree pattern matching algorithm presented in (Hoffmann and O'Donnell, 1982), as it will be discussed in the final section.

Let $G = (\Sigma, R)$ be a TTS, $R = \langle r_1, r_2, \ldots, r_\pi \rangle$. In what follows we construct a DTA that "detects" each subtree of an input tree that is equivalent to some tree in lhs(R). We need to introduce some additional notation. Let N be the set of all nodes from the trees in lhs(R). Call N_r the set of all root nodes (in N), N_m the set of all leftmost nodes, N_l the set of all leaf nodes, and N_a the set of all nodes labeled by $a \in \Sigma$. For each $q \in 2^N$, let $right(q) = \{n \mid n \in N,\ n' \in q,\ n \text{ has immediate left sibling } n'\}$ and let $up(q) = \{n \mid n \in N,\ n' \in q,\ n \text{ has rightmost child } n'\}$. Also, let q_0 be a fresh symbol.

Definition 3 *G is associated with a* DTA *$A_G = (2^N \cup \{q_0\}, \Sigma, \delta_G, q_0, F)$, where $F = \{q \mid q \in 2^N,\ (q \cap N_r) \neq \emptyset\}$ and δ_G is specified as follows:*

(i) $\delta_G(q_0, q_0, a) = N_a \cap N_m \cap N_l$;

(ii) $\delta_G(q_0, q', a) = N_a \cap N_m \cap (N_l \cup up(q'))$, *for* $q' \neq q_0$;

(iii) $\delta_G(q, q_0, a) = N_a \cap N_l \cap (N_r \cup right(q))$, *for* $q \neq q_0$;

(iv) $\delta_G(q, q', a) = N_a \cap up(q') \cap (N_r \cup right(q))$, *for* $q \neq q_0 \neq q'$.

Observe that each state of A_G simultaneously carries over the recognition of several suffixes of trees in lhs(R). These processes are started whenever A_G reads a leftmost node n with the same label as a leftmost leaf node in some tree in lhs(R) (items (i) and (ii) in Definition 3). Note also that we do not require any matching of the left siblings when we match the root of a tree in lhs(R) (items (iii) and (iv)).

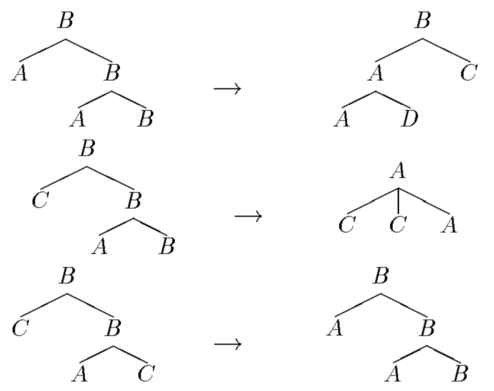

Figure 2: From top to bottom: rules r_1, r_2 and r_3 of G.

Example 4 Let $G = (\Sigma, R)$, where $\Sigma = \{A, B, C, D\}$ and $R = \langle r_1, r_2, r_3 \rangle$. Rules r_i are depicted in Figure 2. We write n_{ij} to denote the j-th node in a post-order enumeration of the nodes of lhs(r_i), $1 \leq i \leq 3$ and $1 \leq j \leq 5$. (Therefore n_{35} denotes the root node of lhs(r_3) and n_{22} denotes the first child of the second child of the root node of lhs(r_2).) If we consider only the useful states, that is those states that can be reached on an actual input, the DTA $A_G = (Q, \Sigma, \delta, q_0, F)$, is specified as follows: $Q = \{q_i \mid 0 \leq i \leq 11\}$, where $q_1 = \{n_{11}, n_{12}, n_{22}, n_{32}\}$, $q_2 = \{n_{21}, n_{31}\}$, $q_3 = \{n_{13}, n_{23}\}$, $q_4 = \{n_{33}\}$, $q_5 = \{n_{14}\}$, $q_6 = \{n_{24}\}$, $q_7 = \{n_{34}\}$, $q_8 = \{n_{15}\}$, $q_9 = \{n_{35}\}$, $q_{10} = \{n_{25}\}$, $q_{11} = \emptyset$; $F = \{q_8, q_9, q_{10}\}$. The transition function δ, restricted to the useful states, is specified in Figure 3. Note that among the $2^{15} + 1$ possible states, only 12 are useful. □

$\delta(q_0, q_0, A)$	$=$	q_1	$\delta(q_0, q_0, C)$	$=$	q_2
$\delta(q_1, q_0, B)$	$=$	q_3	$\delta(q_1, q_0, C)$	$=$	q_4
$\delta(q_1, q_3, B)$	$=$	q_5	$\delta(q_2, q_3, B)$	$=$	q_6
$\delta(q_2, q_4, B)$	$=$	q_7	$\delta(q_0, q_5, B)$	$=$	q_8
$\delta(q_0, q_6, B)$	$=$	q_9	$\delta(q_0, q_7, B)$	$=$	q_{10}

Figure 3: Transition function of G. For all $(q, q', a) \in Q^2 \times \Sigma$ not indicated above, $\delta(q, q', a) = q_{11}$.

Although the number of states of A_G is exponential in $|N|$, in practical cases most of these states are never reached by the automaton on an actual input, and can therefore be ignored. This happens whenever there are few pairs of suffix trees of trees in lhs(R) that share a common prefix tree but no tree in the pair matches the other at the root node. This is discussed at length in (Hoffmann and O'Donnell, 1982), where an upper bound on the number of useful states is provided.

The following lemma provides a characterization of A_G that will be used later.

Lemma 1 *Let n be a node of $T \in \Sigma^T$ and let n' be the root node of $r \in R$. Tree* lhs(r) *matches T at n if and only if $n' \in \hat{\delta}_G(T, n)$.*

Proof (outline). The statement can be shown by proving the following claim. Let m be a node in T and m' be a node in lhs(r). Call $m_1, \ldots, m_k = m$, $k \geq 1$, the ordered sequence of the left siblings of m, with m included, and call $m'_1, \ldots, m'_{k'} = m'$, $k' \geq 1$, the ordered sequence of the left siblings of m', with m' included. If $m' \notin N_r$, then the two following conditions are equivalent:

- $m' \in \hat{\delta}_G(T, m)$;
- $k = k'$ and, for $1 \leq i \leq k$, the suffix of lhs(r) at m'_i matches T at m_i.

The claim can be shown by induction on the position of m' in a post-order enumeration of the nodes of lhs(r). The lemma then follows from the specification of set F and the treatment of set N_r in items (iii) and (iv) in Definition 3. □

We also need a function mapping $F \times \{1..(\pi + 1)\}$ into $\{1..\pi\} \cup \{\perp\}$, specified as $(\min \emptyset = \perp)$:

$$next(q, i) = \min\{j \mid i \leq j \leq \pi,\ \text{lhs}(r_j) \text{ has root node in } q\}. \quad (5)$$

Assume that $q \in F$ is reached by A_G upon reading a node n (in some tree). In the next section $next(q, i)$ is used to select the index of the rule that should be next applied at node n, after the first $i - 1$ rules of R have been considered.

4 The algorithm

We present a translation algorithm for TTS that can immediately be converted into a transformation-based parsing algorithm. We use all definitions introduced in the previous sections. To simplify the presentation, we first make the assumption that the order in which we apply several instances of the same rule to a given tree does not affect the outcome. Later we will deal with the general case.

4.1 Order-free case

We start with an important property that is used by the algorithm below and that can be easily shown (see also (Hoffmann and O'Donnell, 1982)). Let $G = (\Sigma, R)$ be a TTS and let h_G be the maximum height of a tree in lhs(R). Given trees T and S, S a subtree of T, we write $local(T, S)$ to denote the set of all nodes of S and the first h_G proper ancestors of the root of S in T (when these nodes are defined).

Lemma 2 *Assume that* lhs(r), $r \in R$, *matches a tree* T *at some node* n. *Let* $T \overset{r,n}{\Longrightarrow} T'$ *and let* S *be the copy of* rhs(r) *used in the rewriting. For every node* n' *not included in* $local(T', S)$, *we have* $\hat{\delta}_G(T, n') = \hat{\delta}_G(T', n')$. $\square$

We precede the specification of the method with an informal presentation. The following three data structures are used. An associative list *state* associates each node n of the rewritten input tree with the state reached by A_G upon reading n. If n is no longer a node of the rewritten input tree, *state* associates n with the emptyset. A set $rule(i)$ is associated with each rule r_i, containing some of the nodes of the rewritten input tree at which lhs(r_i) matches. A heap data structure H is also used to order the indices of the non-empty sets $rule(i)$ according to the priority of the associated rules in the rule sequence. All the above data structures are updated by a procedure called *update*.

To compute the translation $M(G)$ we first visit the input tree with A_G and initialize our data structures in the following way. For each node n, *state* is assigned a state of A_G as specified above. If rule r_i must be applied first at n, n is added to $rule(i)$ and H is updated. We then enter a main loop and retrieve elements from the heap. When i is retrieved, rule r_i is considered for application at each node n in $rule(i)$. It is important to observe that, since some rewriting of the input tree might have occurred in between the time n has been inserted in $rule(i)$ and the time i is retrieved from H, it could be that the current rule r_i can no longer be applied at n. Information in *state* is used to detect these cases. Crucial to the efficiency of our algorithm, each time a rule is applied only a small portion of the current tree needs to be reread by A_G, in order to update our data structures, as specified by Lemma 2 above. Finally, the main loop is exited when the heap is empty.

Algorithm 1 Let $G = (\Sigma, R)$ be a TTS, $R = \langle r_1, r_2, \ldots, r_\pi \rangle$ and let $T \in \Sigma^T$ be an input tree. Let $A_G = (2^N \cup \{q_0\}, \Sigma, \delta_G, q_0, F)$ be the DTA associated with G and $\hat{\delta}_G$ the reached state function. Let also i be an integer valued variable, *state* be an associative array, $rule(i)$ be an initially empty set, for $1 \leq i \leq \pi$, and let H be a heap data structure. ($n \rightarrow rule(i)$ adds n to $rule(i)$; $i \rightarrow H$ inserts i in H; $i \leftarrow H$ assigns to i the least element in H, if H is not empty.) The algorithm is specified in Figure 4. $\square$

Example 4 (continued) We describe a run of Algorithm 1 working with the sample TTS $G = (\Sigma, R)$ previously specified (see Figure 2).

```
proc update(oldset, newset, j)
for each node n ∈ oldset
    state(n) ← ∅
for each node n ∈ newset do
    state(n) ← δ̂_G(C, n)
    if state(n) ∈ F and next(state(n), j) ≠⊥ then do
      if rule(next(state(n), j)) = ∅
        then next(state(n), j) → H
      n → rule(next(state(n), j))
    od
od

main
C ← T; i ← 1
update(∅, nodes of C, i)
while H not empty do
    i ← H
    for each node n ∈ rule(i) s.t. the root of lhs(r_i)
      is in state(n) do
      S ← the subtree of C matched by lhs(r_i) at n
      S' ← copy of rhs(r_i)
      C ← C[S/S']
      update(nodes of S, local(C, S'), i + 1)
    od
od
return C.
```

Figure 4: Translation algorithm computing $M(G)$ for a TTS G.

Let $C_i \in \Sigma^T$, $1 \leq i \leq 3$, be as depicted in Figure 5. We write m_{ij} to denote the j-th node in a post-order enumeration of the nodes of C_i, $1 \leq i \leq 3$ and $1 \leq j \leq 7$. Assume that C_1 is the input tree.

After the first call to procedure *update*, we have $state(m_{17}) = q_{10} = \{n_{25}\}$ and $state(m_{16}) = q_8 = \{n_{15}\}$; no other final state is associated with a node of C_1. We also have that $rule(1) = \{m_{16}\}$, $rule(2) = \{m_{17}\}$, $rule(3) = \emptyset$ and H contains indices 1 and 2.

Index 1 is then retrieved from H and the only node in $rule(1)$, i.e., m_{16}, is considered. Since the root of lhs(r_1), i.e., node n_{15}, belongs to q_8, m_{16} passes the test in the head of the for-statement in the main program. Then r_1 is applied to C_1, yielding C_2. Observe that $m_{11} = m_{21}$ and $m_{17} = m_{27}$; all the remaining nodes of C_2 are fresh nodes.

The next call to *update*, associated with the application of r_1, updates the associative list *state* in such a way that $state(m_{27}) = q_9 = \{n_{35}\}$, and no other final state is associated with a node of C_2. Also, we now have $rule(1) = \{m_{16}\}$, $rule(2) = \{m_{27}\}$ (recall that $m_{17} = m_{27}$), $rule(3) = \{m_{27}\}$, and H contains indices 2 and 3.

Index 2 is next retrieved from H and node m_{27} is considered. However, at this point the root of lhs(r_2), i.e., node n_{25}, does no longer belong to $state(m_{27})$, indicating that r_2 is no longer applicable to that node. The body of the for-statement in the

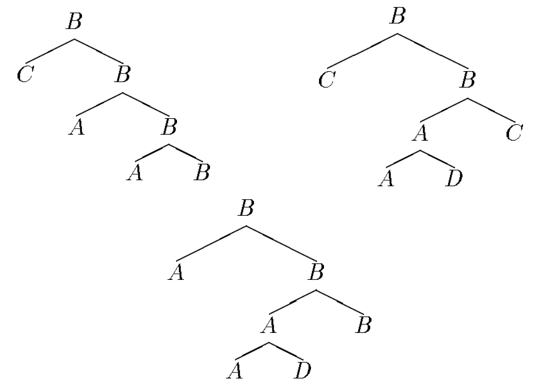

Figure 5: From left to right, top to bottom: trees C_1, C_2 and C_3. In the sample TTS G we have $(C_1, C_3) \in M(G)$, since $C_1 \overset{r_1}{\Rightarrow} C_2 \overset{r_2}{\Rightarrow} C_2 \overset{r_3}{\Rightarrow} C_3$.

main program is not executed this time.

Finally, index 3 is retrieved from H and node m_{27} is again considered, this time for the application of rule r_3. Since the root of $\mathrm{lhs}(r_3)$, i.e., node n_{35}, belongs to $state(m_{27})$, r_3 is applied to C_2 at node m_{27}, yielding C_3. Data structures are again updated by a call to procedure *update* with the second parameter equal to 4. Then state q_8 is associated with node m_{37}, the root node of C_3. Despite of the fact that $q_8 \in F$, we now have $next(q_8, 4) = \bot$. Therefore rule r_1 is not considered for application to C_3. Since H is now empty, the computation terminates returning C_3. $\Box$

The results in Lemma 1 and Lemma 2 can be used to show that, in the main program, a node n passes the test in the head of the for-statement if and only if $\mathrm{lhs}(r_i)$ matches C at n. The correctness of Algorithm 1 then follows from the definition of the heap data structure.

We now turn to computational complexity issues. Let $\rho = \max_{1 \leq i \leq \pi} |r_i|$. For $T \in \Sigma^T$, let also $t(T)$ be the total number of rules that are successfully applied on a run of Algorithm 1 on input T, counting repetitions.

Theorem 1 *The running time of Algorithm* 1 *on input tree* T *is* $O(|T| + \rho t(T) \log(t(T)))$.

Proof. We can implement our data structures in such a way that each of the primitive access operations that are executed by the algorithm takes a constant amount of time.

Consider each instance of the membership of a node n in a set $rule(i)$ and represent it as a pair (n, i). We call *active* each pair (n, i) such that $\mathrm{lhs}(r_i)$ matches C at n at the time i is retrieved from H. As already mentioned, these pairs pass the test in the head of the for-loop in the main program. The number of active pairs is therefore $t(T)$. All remaining pairs are called *dead*. Note that an active pair (n, i) can turn at most $|\mathrm{lhs}(r_i)| + h_R$ active pairs into dead ones, through a call to the procedure *update*. Hence the total number of dead pairs must be $O(\rho t(T))$. We conclude that the number of pairs totally instantiated by the algorithm is $O(\rho t(T))$.

It is easy to see that the number of pairs totally instantiated by the algorithm is also a bound on the number of indices inserted in or retrieved from the heap. Then the time spent by the algorithm with the heap is $O(\rho t(T) \log(t(T)))$ (see for instance (Cormen, Leiserson, and Rivest, 1990)). The first call to the procedure *update* in the main program takes time proportional to $|T|$. All remaining operations of the algorithm will now be charged to some active pair.

For each active pair, the body of the for-loop in the main program and the body of the *update* procedure are executed, taking an amount of time $O(\rho)$. For each dead pair, only the test in the head of the for-loop is executed, taking a constant amount of time. This time is charged to the active node that turned the pair under consideration into a dead one. In this way each active node is charged an extra amount of time $O(\rho)$.

Every operation executed by the algorithm has been considered in the above analysis. We can then conclude that the running time of Algorithm 1 is $O(|T| + \rho t(T) \log(t(T)))$. $\Box$

Let us compare the above result with the time performance of the standard algorithm for transformation-based parsing. The standard algorithm checks each rule in R for application to an initial parse tree T, trying to match the left-hand side of the current rule at each node of T. Using the notation of Theorem 1, the running time is then $O(\pi \rho |T|)$. In practical applications, $t(T)$ and $|T|$ are very close (of the order of the length of the input string). Therefore we have achieved a time improvement of a factor of $\pi / \log(t(T))$. We emphasize that π might be several hundreds large if the learned transformations are lexicalized. Therefore we have improved the asymptotic time complexity of transformation-based parsing of a factor between two to three orders of magnitude.

4.2 Order-dependent parsing

We consider here the general case for the TTS translation problem, in which the order of application of several instances of rule r to a tree can affect the final result of the rewriting. In this case rule r is called *critical*. According to the definition of translation induced by a TTS, a critical rule should always be applied in post-order w.r.t. the nodes of the tree to be rewritten. The solution we propose here for critical rules is based on a preprocessing of the rule sequence of the system.

We informally describe the technique presented below. Assume that a critical rule r is to be applied at several matching nodes of a tree C. We partition the matching nodes into two sets. The first set contains all the nodes n at which the matching of lhs(r) overlaps with a second matching at a node n' dominated by n. All the remaining matching nodes are inserted in the second set. Then rule r is applied to the nodes of the second set. After that, the nodes in the first set are in turn partitioned according to the above criterion, and the process is iterated until all the matching nodes have been considered for application of r. This is more precisely stated in what follows.

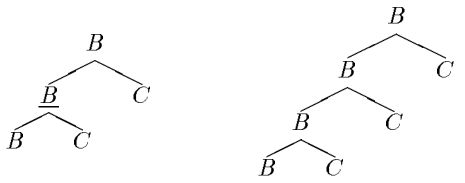

Figure 6: From left to right: trees Q and Q_p. Node p of Q is indicated by underlying its label.

We start with some additional notation. Let $r = (Q \rightarrow Q')$ be a tree-rewriting rule. Also, let p be a node of Q and let S be the suffix of Q at p. We say that p is *periodic* if (i) p is not the root of Q; and (ii) S matches Q at the root node. It is easy to see that the fact that lhs(r) has some periodic node is a necessary condition for r to be critical. Let the root of S be the i-th child of a node n_f in Q, and let Q_c be a copy of Q. We write Q_p to denote the tree obtained starting from Q by excising S and by letting the root of Q_c be the new i-th child of n_f. Finally, call n_1 the root of Q_p and n_2 the root of Q.

Example 5 Figure 6 depicts trees Q and Q_p. The periodic node p of Q under consideration is indicated by underlying its label. $\Box$

Let us assume that rule r is critical and that p is the only periodic node in Q. We add Q_p to set lhs(R) and construct A_G accordingly. Algorithm 1 should then be modified as follows. We call *p-chain* any sequence of one or more subtrees of C, all matched by Q, that partially overlap in C. Let n be a node of C and let $q = state(n)$. Assume that $n_2 \in q$ and call S the subtree of C at n matched by Q (S exists by Lemma 1). We distinguish two possible cases.

Case 1: If $n_1 \in q$, then we know that Q also matches some portion of C that overlaps with S (at the node matched by the periodic node p of Q). In this case S belongs to a p-chain consisting of at least two subtrees and S is not the bottom-most subtree in the p-chain.

Case 2: If $n_1 \notin q$, then we know that S is the bottom-most subtree in a p-chain.

Let i be the index of rule r under consideration. We use an additional set $chain(i)$. Each node n of C such that $n_2 \in state(n)$ is then inserted in $chain(i)$ if $state(n)$ satisfies Case 1 above, and is inserted in $rule(i)$ otherwise. Note that $chain(i)$ is non-empty only in case $rule(i)$ is such. Whenever i is retrieved from H, we process each node n in $rule(i)$, as usual. But when we update our data structures with the procedure *update*, we also look for matchings of lhs(r_i) at nodes of C in $chain(i)$. The overall effect of this is that each p-chain is considered in a bottom-up fashion in the application of r. This is compatible with the post-order application requirement.

The above technique can be applied for each periodic node in a critical rule, and for each critical rule of G. This only affects the size of A_G, not the time requirements of Algorithm 1. In fact, the proposed preprocessing can at worst double h_G.

5 Discussion

In this section we relate our work with the existing literature and further discuss our result.

There are several alternative ways in which one could see transformation-based rewriting systems. TTS's are closely related to a class of graph rewriting systems called neighbourhood-controlled embedding graph grammars (NCE grammars; see (Janssens and Rozenberg, 1982)). In fact our definition of the $\stackrel{r,n}{\Longrightarrow}$ relation and of the underlying [/] operator has been inspired by similar definitions in the NCE formalism. Apart from the restriction to tree rewriting, the main difference between NCE grammars and TTS's is that in the latter formalism the productions are totally ordered, therefore there is no recursion.

Ordered trees can also be seen as ground terms. If we extend the alphabet Σ with variable symbols, we can redefine the $\stackrel{r,n}{\Longrightarrow}$ relation through variable substitution. In this way a TTS becomes a particular kind of term-rewriting system. The idea of imposing a total order on the rules of a term-rewriting system can be found in the literature, but in these cases all rules are reconsidered for application at each step in the rewriting, using their priority (see for instance the priority term-rewriting systems (Baeten, Bergstra, and Klop, 1987)). Therefore these systems allow recursion. There are cases in which a critical rule in a TTS does not give rise to order-dependency in rewriting. Methods for deciding the confluency property for a term-rewriting system with critical pairs (see (Dershowitz and Jouannaud, 1990) for definitions and an overview) can also be used to detect the above cases for TTS.

As already pointed out, the translation problem investigated here is closely related with the standard tree pattern matching problem. Our automata A_G (Definition 3) can be seen as an abstraction of the bottom-up tree pattern matching algorithm presented in (Hoffmann and O'Donnell, 1982). While that result uses a representation of the pattern set (our set lhs(R)) requiring an amount of space which is exponential in the degree of the pattern trees, as an improvement, our transition function does not depend on this parameter. However, in the worst case the space requirements of both algorithm are exponential in the number of nodes in lhs(R) (see the analysis in (Hoffmann and O'Donnell, 1982)). As already discussed in Section 3, the worst case condition is hardly met in natural language applications.

Polynomial space requirements can be guaranteed if one switches to top-down tree pattern matching algorithms. One such a method is reported in (Hoffmann and O'Donnell, 1982), but in this case the running-time of Algorithm 1 cannot be maintained. Faster top-down matching algorithms have been reported in (Kosaraju, 1989) and (Dubiner, Galil, and Magen, 1994), but these methods seems impractical, due to very large hidden constants.

A tree-based extension of the very fast algorithm described in (Roche and Schabes, 1995) is in principle possible for transformation-based parsing, but is likely to result in huge space requirements and seems impractical. The algorithm presented here might then be a good compromise between fast parsing and reasonable space requirements.

When restricted to monadic trees, our automaton A_G comes down to the finite state device used in the well-known string pattern matching algorithm of Aho and Corasick (see (Aho and Corasick, 1975)), requiring linear space only. If space requirements are of primary importance or when the rule set is very large, our method can then be considered for string-based transformation rewriting as an alternative to the already mentioned method in (Roche and Schabes, 1995), which is faster but has more onerous space requirements.

Acknowledgements

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 also 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 with Alberto Apostolico, Rao Kosaraju, Fernando Pereira and Murat Saraclar for technical discussions on topics related to this paper. The authors whish to thank an anonymous referee for having pointed out important connections between TTS and term-rewriting systems.

References

Aho, A. V. and M. Corasick. 1975. Efficient string matching: An aid to bibliographic search. *Communications of the Association for Computing Machinery*, 18(6):333–340.

Baeten, J., J. Bergstra, and J. Klop. 1987. Priority rewrite systems. In *Proc. Second International Conference on Rewriting Techniques and Applications, LNCS 256*, pages 83–94, Berlin, Germany. Springer-Verlag.

Brill, E. 1993. Automatic grammar induction and parsing free text: A transformation-based approach. In *Proceedings of the 31st Meeting of the Association of Computational Linguistics*, Columbus, Oh.

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

Brill, E. and P. Resnik. 1994. A transformation-based approach to prepositional phrase attachment disambiguation. In *Proceedings of the Fifteenth International Conference on Computational Linguistics (COLING-1994)*, Kyoto, Japan.

Chomsky, N. 1965. *Aspects of the Theory of Syntax*. The MIT Press, Cambridge, MA.

Chomsky, N. and M. Halle. 1968. *The Sound Pattern of English*. Harper and Row.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest. 1990. *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

Dershowitz, N. and J. Jouannaud. 1990. Rewrite systems. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier and The MIT Press, Amsterdam, The Netherlands and Cambridge, MA, chapter 6, pages 243–320.

Dubiner, M., Z. Galil, and E. Magen. 1994. Faster tree pattern matching. *Journal of the Association for Computing Machinery*, 41(2):205–213.

Hoffmann, C. M. and M. J. O'Donnell. 1982. Pattern matching in trees. *Journal of the Association for Computing Machinery*, 29(1):68–95.

Janssens, D. and G. Rozenberg. 1982. Graph grammars with neighbourhood-controlled embedding. *Theoretical Computer Science*, 21:55–74.

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

Kosaraju, S. R. 1989. Efficient tree-pattern matching. In *Proceedings of the 30 Conference on Foundations of Computer Science (FOCS)*, pages 178–183.

Roche, E. and Y. Schabes. 1995. Deterministic part of speech tagging with finite state transducers. *Computational Linguistics*.

Thatcher, J. W. 1967. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1:317–322.