Training Tree Transducers

Jonathan Graehl*
University of Southern California

Kevin Knight**
University of Southern California

Jonathan May†
University of Southern California

Many probabilistic models for natural language are now written in terms of hierarchical tree structure. Tree-based modeling still lacks many of the standard tools taken for granted in (finite-state) string-based modeling. The theory of tree transducer automata provides a possible framework to draw on, as it has been worked out in an extensive literature. We motivate the use of tree transducers for natural language and address the training problem for probabilistic tree-to-tree and tree-to-string transducers.

1. Introduction

Much natural language work over the past decade has employed probabilistic finite-state transducers (FSTs) operating on strings. This has occurred somewhat under the influence of speech recognition research, where transducing acoustic sequences to word sequences is neatly captured by left-to-right stateful substitution. Many conceptual tools exist, such as Viterbi decoding (Viterbi 1967) and forward–backward training (Baum and Eagon 1967), as well as software toolkits like the AT&T FSM Library and USC/ISI's Carmel.[1] Moreover, a surprising variety of problems are attackable with FSTs, from part-of-speech tagging to letter-to-sound conversion to name transliteration.

However, language problems like machine translation break this mold, because they involve massive re-ordering of symbols, and because the transformation processes seem sensitive to hierarchical tree structure. Recently, specific probabilistic tree-based models have been proposed not only for machine translation (Wu 1997; Alshawi, Bangalore, and Douglas 2000; Yamada and Knight 2001; Eisner 2003; Gildea 2003), but also for summarization (Knight and Marcu 2002), paraphrasing (Pang, Knight, and Marcu 2003), natural language generation (Langkilde and Knight 1998; Bangalore and Rambow 2000; Corston-Oliver et al. 2002), parsing, and language modeling (Baker 1979; Lari and Young 1990; Collins 1997; Chelba and Jelinek 2000; Charniak 2001; Klein

* Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292. E-mail: graehl@isi.edu.
** Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292. E-mail: knight@isi.edu.
† Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292. E-mail: jonmay@isi.edu.

1 `www.research.att.com/sw/tools/fsm and www.isi.edu/licensed-sw/carmel.`

Submission received: 30 October 2003; revised submission received: 30 August 2007; accepted for publication: 20 October 2007.

and Manning 2003). It is useful to understand generic algorithms that may support all these tasks and more.

Rounds (1970) and Thatcher (1970) independently introduced tree transducers as a generalization of FSTs. Rounds was motivated by natural language:

> Recent developments in the theory of automata have pointed to an extension of the domain of definition of automata from strings to trees . . . parts of mathematical linguistics can be formalized easily in a tree-automaton setting . . . We investigate decision problems and closure properties. Our results should clarify the nature of syntax-directed translations and transformational grammars . . . (Rounds 1970)

The Rounds/Thatcher tree transducer is very similar to a left-to-right FST, except that it works top-down, pursuing subtrees independently, with each subtree transformed depending only on its own passed-down state. This class of transducer, called R in earlier works (Gécseg and Steinby 1984; Graehl and Knight 2004) for "root-to-frontier," is often nowadays called T, for "top-down".

Rounds uses a mathematics-oriented example of a T transducer, which we repeat in Figure 1. At each point in the top-down traversal, the transducer chooses a production to apply, based *only* on the current state and the current root symbol. The traversal continues until there are no more state-annotated nodes. Non-deterministic transducers may have several productions with the same left-hand side, and therefore some free choices to make during transduction.

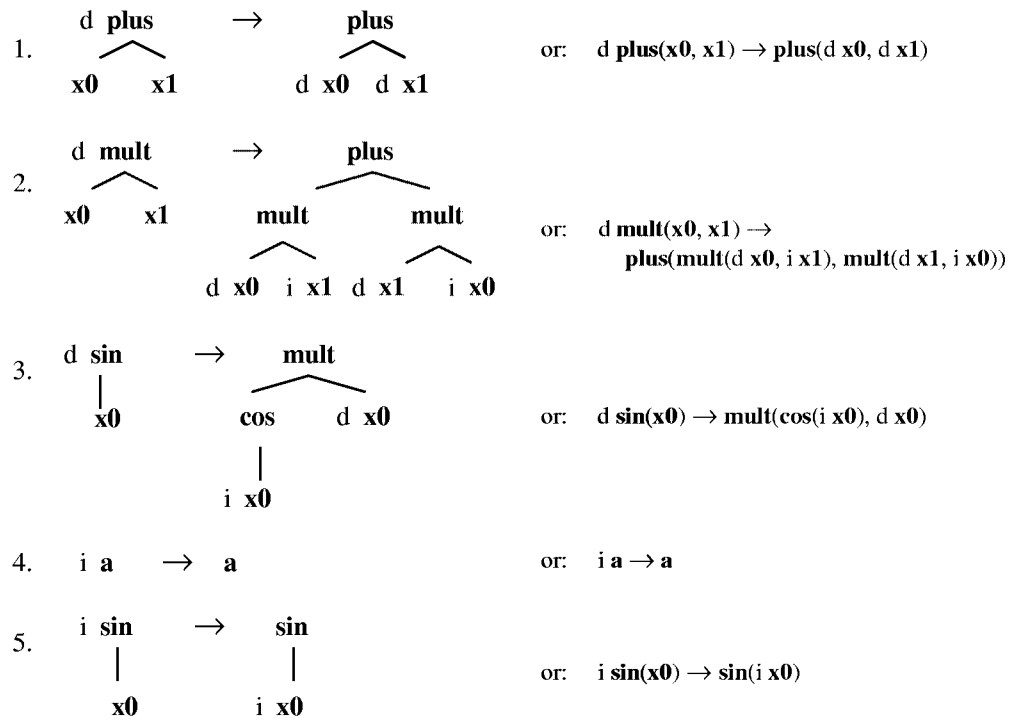
A T transducer compactly represents a potentially infinite set of input/output tree pairs: exactly those pairs (T1, T2) for which some sequence of productions applied to T1 (starting in the initial state) results in T2. This is similar to an FST, which compactly represents a set of input/output string pairs; in fact, T is a generalization of FST. If we think of strings written down vertically, as degenerate trees, we can convert any FST into a T transducer by automatically replacing FST transitions with T productions, as follows: If an FST transition from state q to state r reads input symbol A and outputs symbol B, then the corresponding T production is q A(x0) → B(r x0). If the FST transition output is epsilon, then we have instead q A(x0) → r x0, or if the input is epsilon, then q x0 → B(r x0). Figure 2 depicts a sample FST and its equivalent T transducer.

T does have some extra power beyond path following and state-based record-keeping. It can copy whole subtrees, and transform those subtrees differently. It can also delete subtrees without inspecting them (imagine by analogy an FST that quits and accepts right in the middle of an input string). Variants of T that disallow copying and deleting are called LT (for *linear*) and NT (for *nondeleting*), respectively.

One advantage to working with tree transducers is the large and useful body of literature about these automata; two excellent surveys are Gécseg and Steinby (1984) and Comon et al. (1997). For example, it is known that T is not closed under composition (Rounds 1970), and neither are LT or B (the "bottom-up" cousin of T), but the non-copying LB is closed under composition. Many of these composition results are first found in Engelfriet (1975).

The power of T to change the structure of an input tree is surprising. For example, it may not be initially obvious how a T transducer can transform the English structure S(PRO, VP(V, NP)) into the Arabic equivalent S(V, PRO, NP), as it is difficult to move the subject PRO into position between the verb V and the direct object NP. First, T productions have no lookahead capability—the left-hand-side of the S production

Transducer alphabet: {**0**, **1**, **a**, **y**, **sin**, **cos**, **plus**, **mult**}
Transducer states: {d, i} Note: d means "take derivative", i means "identity"
Transducer rules:



Transducer in action:

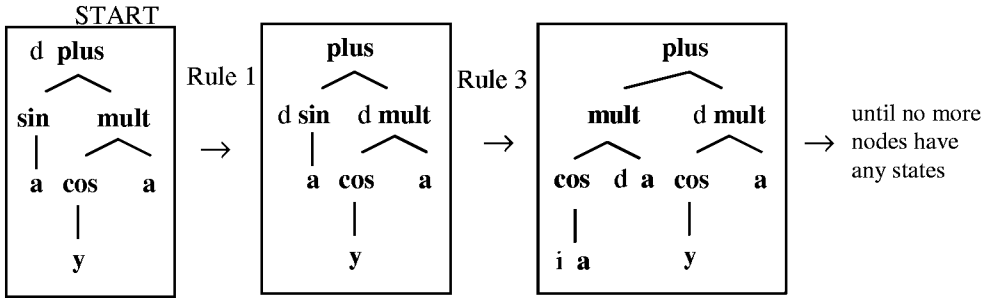
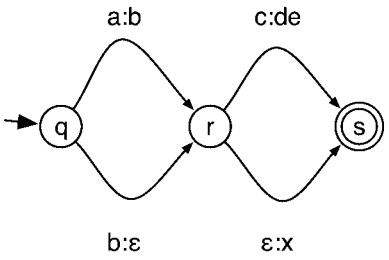


Figure 1
Part of a sample T tree transducer, adapted from Rounds (1970).

consists only of q S(x0, x1), although we want the English-to-Arabic transformation to apply only when it faces the entire structure q S(PRO, VP(V, NP)). However, we can simulate lookahead using states, as in these productions:

q S(x0, x1) → S(qpro x0, qvp.v.np x1)
qpro PRO → PRO
qvp.v.np VP(x0, x1) → VP(qv x0, qnp x1)



Transducer alphabet: **{a, b, c, d, e, x, END}**

Transducer states: {q, r, s}

Transducer rules:

1. q **a** → **b** (**x0**) — r **x0**
2. q **b** (**x0**) → r **x0**
3. r **c** (**x0**) → **d** (**e** (s **x0**))
4. r **x0** → **x** (s **x0**)
5. s **END** → **END**

Figure 2
An FST and its equivalent T transducer.

By omitting rules like qpro NP → ..., we ensure that the entire production sequence will dead-end unless the first child of the input tree is in fact PRO. So finite lookahead (into inputs we don't delete) is not a problem. But these productions do not actually move the subtrees around. The next problem is how to get the PRO to appear between the V and NP, as in Arabic. This can be carried out using copying. We make two copies of the English VP, and assign them different states, as in the following productions. States encode instructions for extracting/positioning the relevant portions of the VP. For example, the state qleft.vp.v means "assuming this tree is a VP whose left child is V, output only the V, and delete the right child":

q S(x0, x1) → S(qleft.vp.v x1, qpro x0, qright.vp.np x1)
qpro PRO → PRO
qleft.vp.v VP(x0, x1) → qv x0
qright.vp.np VP(x0, x1) → qnp x1

With these rules, the transduction proceeds as in Figure 3. This ends our informal presentation of tree transducers.

Although general properties of T are understood, there are many algorithmic questions. In this article, we take on the problem of *training* probabilistic T transducers. For many language problems (machine translation, paraphrasing, text compression, etc.), it is possible to collect training data in the form of tree pairs and to distill linguistic knowledge automatically. Our problem statement is: Given (1) a particular transducer

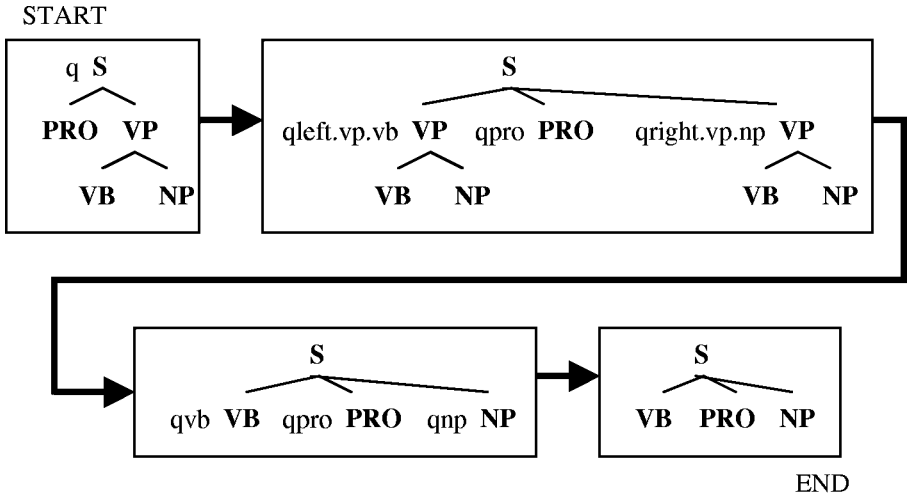


Figure 3
Multilevel re-ordering of nodes in a T-transducer.

with rules R, and (2) a finite training set of sample input/output tree pairs, we want to produce (3) a probability estimate for each rule in R such that we maximize the probability of the output trees given the input trees. As with the forward–backward algorithm, we seek at least a local maximum. Tree transducers with weights have been studied (Kuich 1999; Engelfriet, Fülöp, and Vogler 2004; Fülöp and Vogler 2004) but we know of no existing training procedure.

Sections 2–4 of this article define basic concepts and recall the notions of relevant automata and grammars. Sections 5–7 describe a novel tree transducer training algorithm, and Sections 8–10 describe a variant of that training algorithm for trees and strings. Section 11 presents an example linguistic tree transducer and provides empirical evidence of the feasibility of the training algorithm. Section 12 describes how the training algorithm may be used for training context-free grammars. Section 13 discusses related and future work.

2. Trees

T_Σ is the set of *(rooted, ordered, labeled, finite) trees over alphabet* Σ. An *alphabet* is a finite set. (see Table 1)

$T_\Sigma(X)$ are the *trees over alphabet* Σ, *indexed by* X—the subset of $T_{\Sigma\cup X}$ where only leaves may be labeled by X ($T_\Sigma(\emptyset) = T_\Sigma$). *Leaves* are nodes with no children.

The *nodes* of a tree t are identified one-to-one with its *paths*: $paths_t \subset paths \equiv \mathbb{N}^* \equiv \bigcup_{i=0}^{\infty} \mathbb{N}^i$ ($\mathbb{N}^0 \equiv \{()\}$). The *size* of a tree is the number of nodes: $|t| = |paths_t|$. The path to the root is the empty sequence (), and p_1 *extended by* p_2 is $p_1 \cdot p_2$, where $\cdot$ is the concatenation operator:

$$(a_1, \ldots, a_n) \cdot (b_1, \ldots, b_m) \equiv (a_1, \ldots, a_n, b_1, \ldots, b_m)$$

For $p \in paths_t$, $rank_t(p)$ is the number of children, or *rank*, of the node at p, and $label_t(p) \in \Sigma$ is its *label*. The *ranked label* of a node is the pair $labelandrank_t(p) \equiv (label_t(p), rank_t(p))$. For $1 \leq i \leq rank_t(p)$, the i^{th} *child* of the node at p is located at

Table 1
Notation guide.

Notation	Meaning
$(w)FS(T,A)$	(weighted) finite-state string (transducers,acceptors)
$(w)RTG$	(weighted) regular tree grammars (generalizes PCFG)
$(x)(L)(N)T(s)$	(extended) (linear) (nondeleting) top–down tree(-to-string) transducers
$(S)T(A,S)G$	(synchronous) tree (adjoining,substitution) grammars
$(S,P)CFG$	(synchronous,probabilistic) context-free grammars
$\mathbb{R}^+$	positive real numbers
$\mathbb{N}$	natural numbers: $\{1,2,3,\ldots\}$
$\emptyset$	empty set
$\equiv$	equals (by definition)
$\|A\|$	size of finite set A
X^*	Kleene star of X, i.e., strings over alphabet X: $\{(x_1,\ldots,x_n) \mid n \geq 0\}$
$a \cdot b$	String concatenation: $(1)\cdot(2,3) = (1,2,3)$
$<_{lex}$	lexicographic (dictionary) order: $() < (1) < (1,1) < \ldots < (1,2) < \ldots$
Σ	alphabet (set of symbols) (commonly: input tree alphabet)
$t \in T_\Sigma$	t is a tree with label alphabet Σ
$T_\Sigma(X)$	... and with *variables* from additional leaf label alphabet X
$A(t)$	tree constructed by placing a unary A above tree t
$A((x_1,\ldots,x_n))$	tree constructed by placing an n-ary A over leaves $(x_1,\ldots,x_n)$
p	tree path, e.g., (a,b) is the b^{th} child of the a^{th} child of root
$paths$	the set of all tree paths ($\equiv \mathbb{N}^*$)
$paths_t$	subset of $paths$ that lead to actual nodes in t
$paths_t(\{A,B\})$	paths that lead to nodes labeled A or B in t
$t \downarrow p$	the subtree of t with root at p, so that $(t \downarrow p) \downarrow q = t \downarrow (p \cdot q)$
$rank_t(p)$	the number of children of the node p of t
$label_t(p)$	the label of node p of t
$labelandrank_t(p)$	the pair $(label_t(p), rank_t(p))$
$t[p \leftarrow t']$	substitution of tree t' for the subtree $t \downarrow p$
$t[p \leftarrow t'_p, \forall p \in P]$	parallel substitution of tree t'_p for each $t \downarrow p$
$yield_t(X)$	the left $\rightarrow$ right concatenation of the X labels of the leaves of t
$S \in N$	start nonterminal of a regular tree grammar
P, R	productions of a regular tree grammar, rules of a tree transducer
$D(M)$	derivations (keeping a list of applied rewrites) of M
$LD(M)$	*leftmost* derivations of M
$w_M(d \in D(M))$	*weight* of a derivation d: product of weight of each rule usage
$W_M(x)$	*total weight* of x in M: sum of weight of all $LD(M)$ producing x
$\mathcal{L}(M)$	weighted tree set, tree relation, or tree-to-string relation of M
Δ	output tree alphabet
$Q_i \in Q$	initial (start) state of a transducer
$\lambda \in \boldsymbol{xTPAT}_\Sigma$	functions from T_Σ to $\{0,1\}$ that examine finitely many paths
$\boldsymbol{True}$	the tree pattern $\boldsymbol{True}(t) \equiv 1, \forall t$
$s \in \Sigma^*$	s is a string from alphabet Σ, e.g., () the empty string
$s[i]$	i^{th} letter of string s - the i^{th} projection π^i
$indices_s$	i such that $s[i]$ exists: $(1,\ldots,\|s\|)$
$letters_s$	set of all letters $s[i]$ in s
$\|s\|$	length of string; $\|s\| = \|indices_s\|$, *not* $\|letters_s\|$
$spans_s$	Analogous to tree paths, pairs (i,j) denoting substrings
$s \downarrow (i,j)$	The substring $(s[i],\ldots,s[j-1])$ indicated by the span $(i,j) \in spans_s$
$s \downarrow [i]$	same as $s[i]$; $[i]$ stands for the span $(i, i+1)$
$s[p \leftarrow s']$	Substitution of string s' for span p of s
$s[p \leftarrow s'_p, \forall p \in P]$	Parallel (non-overlapping) substitution of string s'_p for each $s \downarrow p$

path $p \cdot (i)$. The *subtree at path p of t* is $t \downarrow p$, defined by $paths_{t\downarrow p} \equiv \{q \mid p \cdot q \in paths_t\}$ and $labelandrank_{t\downarrow p}(q) \equiv labelandrank_t(p \cdot q)$.

The *paths to X in t* are $paths_t(X) \equiv \{p \in paths_t \mid label_t(p) \in X\}$.

A set of paths $F \subset paths$ is a *frontier* iff it is *pairwise prefix-independent*:

$$\forall p_1, p_2 \in F, p \in paths : p_1 = p_2 \cdot p \implies p_1 = p_2$$

We write $\mathcal{F}$ for the set of all frontiers. F is a *frontier of t*, if $F \subset \mathcal{F}_t$ is a frontier whose paths are all valid for t—$\mathcal{F}_t \equiv \mathcal{F} \cap paths_t$.

For $t, s \in T_\Sigma(X), p \in paths_t$, $t[p \leftarrow s]$ is the *substitution of s for p* in t, where the subtree at path p is replaced by s. For a frontier F of t, the *parallel substitution of* t'_p *for the frontier* $F \in \mathcal{F}_t$ *in t* is written $t[p \leftarrow t'_p, \forall p \in F]$, where there is a $t'_p \in T_\Sigma(X)$ for each path p. The result of a parallel substitution is the composition of the serial substitutions for all $p \in F$, replacing each $t \downarrow p$ with t'_p. (If F were not a frontier, the result would vary with the order of substitutions sharing a common prefix.) For example: $t[p \leftarrow t \downarrow p \cdot (1), \forall p \in F]$ would splice out each node $p \in F$, replacing it by its first subtree.

Trees may be written as strings over $\Sigma \cup \{(,)\}$ in the usual way. For example, the tree $t = \mathrm{S(NP, VP(V, NP))}$ has $labelandrank_t((2)) = (\mathrm{VP}, 2)$ and $labelandrank_t((2,1)) = (\mathrm{V}, 0)$. Commas, written only to separate symbols in Σ composed of several typographic letters, should not be considered part of the string. For example, if we write $\sigma(t)$ for $\sigma \in \Sigma, t \in T_\Sigma$, we mean the tree with $label_{\sigma(t)}(()) \equiv \sigma$, $rank_{\sigma(t)}(()) \equiv 1$ and $\sigma(t) \downarrow (1) \equiv t$. Using this notation, we can give a definition of $T_\Sigma(X)$:

$$\text{If } x \in X, \text{ then } x \in T_\Sigma(X) \tag{1}$$

$$\text{If } \sigma \in \Sigma, \text{ then } \sigma \in T_\Sigma(X) \tag{2}$$

$$\text{If } \sigma \in \Sigma \text{ and } t_1, \ldots, t_n \in T_\Sigma(X), \text{ then } \sigma(t_1, \ldots, t_n) \in T_\Sigma(X) \tag{3}$$

The *yield of X in t* is $yield_t(X)$, the concatenation (in lexicographic order[2]) over paths to leaves $l \in paths_t$ (such that $rank_t(l) = 0$) of $label_t(l) \in X$—that is, the string formed by reading out the leaves labeled with X in left-to-right order. The usual case (the *yield of t*) is $yield_t \equiv yield_t(\Sigma)$. More precisely,

$$yield_t(X) \equiv \begin{cases} l & \text{if } r = 0 \wedge l \in X \text{ where } (l, r) \equiv labelandrank_t(()) \\ () & \text{if } r = 0 \wedge l \notin X \\ \bullet_{i=1}^{r} yield_{t\downarrow(i)}(X) & \text{otherwise} \qquad \text{where } \bullet_{i=1}^{r} s_i \equiv s_1 \cdot \ldots \cdot s_r \end{cases}$$

3. Regular Tree Grammars

In this section, we describe the **regular tree grammar**, a common way of compactly representing a potentially infinite set of trees (similar to the role played by the regular grammar for strings). We describe the version where trees in a set have different weights, in the same way that a weighted finite-state acceptor gives weights for strings

2 $() <_{lex} (a)$, $(a_1) <_{lex} (a_2)$ iff $a_1 < a_2$, $(a_1) \cdot b_1 <_{lex} (a_2) \cdot b_2$ iff $a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 <_{lex} b_2)$.

Σ = {S, NP, VP, PP, PREP, DET, N, V, run, the, of, sons, daughters}
N = {qnp, qpp, qdet, qn, qprep}
S = q
P = {q $\rightarrow^{1.0}$ S(qnp, VP(VB(run))),
qnp $\rightarrow^{0.6}$ NP(qdet, qn),
qnp $\rightarrow^{0.4}$ NP(qnp, qpp),
qpp $\rightarrow^{1.0}$ PP(qprep, np),
qdet $\rightarrow^{1.0}$ DET(the),
qprep $\rightarrow^{1.0}$ PREP(of),
qn $\rightarrow^{0.5}$ N(sons),
qn $\rightarrow^{0.5}$ N(daughters)}

Sample generated trees:

S(NP(DT(the), N(sons)),
VP(V(run)))

(with probability 0.3)

S(NP(NP(DT(the), N(sons)),
PP(PREP(of), NP(DT(the), N(daughters)))),
VP(V(run)))

(with probability 0.036)

Figure 4
A sample weighted regular tree grammar (wRTG).

in a regular language; when discussing weights, we assume the commutative semiring $(\{r \in \mathbb{R} \mid r \geq 0\}, +, \cdot, 0, 1)$ of nonnegative reals with the usual sum and product.

A **weighted regular tree grammar** (wRTG) G is a quadruple (Σ, N, S, P), where Σ is the alphabet, N is the finite set of *nonterminals*, $S \in N$ is the *start (or initial) nonterminal*, and $P \subseteq N \times T_\Sigma(N) \times \mathbb{R}^+$ is a finite set of *weighted productions* ($\mathbb{R}^+ \equiv \{r \in \mathbb{R} \mid r > 0\}$). A production (lhs, rhs, w) is written $lhs \rightarrow^w rhs$ (if w is omitted, the multiplicative identity 1 is assumed). Productions whose rhs contains no nonterminals ($rhs \in T_\Sigma$) are called *terminal productions*, and rules of the form $A \rightarrow^w B$, for $A, B \in N$ are called ϵ*-productions*, or *state-change productions*, and can be used in lieu of multiple initial nonterminals.

Figure 4 shows a sample wRTG. This grammar generates an infinite number of trees.

We define the binary derivation relation on *terms* $T_\Sigma(N)$ and derivation *histories* $(T_\Sigma(N \times (paths \times P)^*)$:

$$\Rightarrow_G \equiv \left\{ ((a,h),(b,h \cdot (p,(l,r,w)))) \mid (l,r,w) \in P \wedge \right.$$
$$p \in paths_a(\{l\}) \wedge$$
$$\left. b = a[p \leftarrow r] \right\}$$

That is, $(a, h) \Rightarrow_G (b, h \cdot (p, (l, r, w)))$ iff b may be derived from a by using the rule $l \rightarrow^w r$ to replace the nonterminal leaf l at path p with r. The reflexive, transitive closure of $\Rightarrow_G$ is written $\Rightarrow_G^*$, and the *derivations of* G, written $D(G)$, are the ways the start nonterminal may be expanded into entirely terminal trees:

$$D(G) \equiv \left\{ (t, h) \in T_\Sigma \times (paths \times P)^* \mid (S, ()) \Rightarrow_G^* (t, h) \right\}$$

We also project the $\Rightarrow_G^*$ relation so that it refers only to trees: $t' \Rightarrow_G^* t$ iff $\exists h', h \in (paths \times P)^* : (t', h') \Rightarrow_G^* (t, h)$.

We take the product of the used weights to get the *weight of a derivation* $d \in D(G)$:

$$w_G((t, (h_1, \ldots, h_n)) \in D(G)) \equiv \prod_{i=1}^{n} w_i \text{ where } h_i = (p_i, (l_i, r_i, w_i))$$

The *leftmost derivations of* G build a tree preorder from left to right (always expanding the leftmost nonterminal in its string representation):

$$LD(G) \equiv \left\{ (t, ((p_1, r_1), \ldots, (p_n, r_n))) \in D_G \mid \forall 1 \leq i < n : p_{i+1} \not<_{lex} p_i \right\}$$

The *total weight of* t *in* G is given by $W_G : T_\Sigma \rightarrow \mathbb{R}$, the sum of the weights of leftmost derivations producing t: $W_G(t) \equiv \sum_{(t,h) \in LD(G)} w_G((t, h))$. Collecting the total weight of every possible (nonzero weight) output tree, we call $\mathcal{L}(G)$ the *weighted tree language* of G, where $\mathcal{L}(G) = \{(t, w) \mid W_G(t) = w \wedge w > 0\}$ (the unweighted tree language is simply the first projection).

For every weighted context-free grammar, there is an equivalent wRTG that generates its weighted derivation trees (whose yield is a string in the context-free language), and the yield of any regular tree language is a context-free string language (Gécseg and Steinby 1984). We can also interpret a regular tree grammar as a context-free string grammar with alphabet $\Sigma \cup \{(,)\}$.

wRTGs generate (ignoring weights) exactly the *recognizable tree languages*, which are sets of trees accepted by a non-transducing automaton version of T. This acceptor automaton is described in Doner (1970) and is actually a closer mechanical analogue to an FSA than is the rewrite-rule-based wRTG. RTGs are closed under intersection (Gécseg and Steinby 1984), and the constructive proof also applies to weighted wRTG intersection. There is a normal form for wRTGs analogous to that of regular grammars: Right-hand sides are a single terminal root with (optional) nonterminal children. What is sometimes called a *forest* in natural language generation (Langkilde 2000; Nederhof and Satta 2002) is a *finite* wRTG without loops—for all valid derivation trees, each nonterminal may only occur once in any path from root to a leaf:

$$\forall n \in N, t \in T_\Sigma(N), h \in (paths \times P)^* : (n, ()) \Rightarrow_G^* (t, h) \implies paths_t(\{n\}) = \emptyset$$

RTGs produce tree sets equivalent to those produced by tree substitution grammars (TSGs) (Schabes 1990) up to relabeling. The relabeling is necessary because RTGs distinguish states and tree symbols, which are conflated in TSGs at the elementary tree root. Regular tree languages are strictly contained in tree sets of tree adjoining grammars (TAG; Joshi and Schabes 1997), which generate string languages strictly between the context-free and indexed languages. RTGs are essentially TAGs without auxiliary trees

and their adjunction operation; the productions correspond exactly to TAG's initial trees and the elementary tree substitution operation.

4. Extended-LHS Tree Transducers (xT)

Section 1 informally described the root-to-frontier transducer class T. We saw that T allows, by use of states, finite lookahead and arbitrary rearrangement of non-sibling input subtrees removed by a finite distance. However, it is often easier to write rules that explicitly represent such lookahead and movement, relieving the burden on the user to produce the requisite intermediary rules and states. We define xT, a generalization of weighted T. Because of its good fit to natural language problems, xT is already briefly touched on, though not defined, in Section 4 of Rounds (1970).

A *weighted extended-lhs top-down tree transducer* M is a quintuple $(\Sigma, \Delta, Q, Q_i, R)$ where Σ is the input alphabet, and Δ is the output alphabet, Q is a finite set of states, $Q_i \in Q$ is the *initial (or start, or root) state*, and $R \subseteq Q \times \boldsymbol{xTPAT}_\Sigma \times T_\Delta(Q \times \mathit{paths}) \times \mathbb{R}^+$ is a finite set of *weighted transformation rules*. $\boldsymbol{xTPAT}_\Sigma$ is the set of *finite tree patterns*: predicate functions $f : T_\Sigma \to \{0, 1\}$ that depend only on the label and rank of a finite number of fixed paths of their input. A rule $(q, \lambda, \mathit{rhs}, w)$ is written $\mathrm{q}\,\lambda \to^w \mathit{rhs}$, meaning that an input subtree matching λ while in state q is transformed into *rhs*, with $Q \times \mathit{paths}$ leaves replaced by their (recursive) transformations. The $Q \times \mathit{paths}$ leaves of a *rhs* are called *nonterminals* (there may also be *terminal* leaves labeled by the output tree alphabet Δ).

xT is the set of all such transducers T; the set of conventional top-down transducers, is a subset of xT where the rules are restricted to use finite tree patterns that depend only on the root: $\boldsymbol{TPAT}_\Sigma \equiv \{p_{\sigma,r}(t)\}$ where $p_{\sigma,r}(t) \equiv (\mathit{label}_t(()) = \sigma \wedge \mathit{rank}_t(()) = r)$.

Rules whose *rhs* are a pure T_Δ with no states/paths for further expansion are called *terminal rules*. Rules of the form $\mathrm{q}\,\lambda \to^w q'\,()$ are ϵ-*rules*, or *state-change rules*, which substitute state q' for state q without producing output, and stay at the current input subtree. Multiple initial states are not needed: we can use a single start state Q_i, and instead of each initial state q with starting weight w add the rule $Q_i\ \boldsymbol{True} \to^w q\ ()$ (where $\boldsymbol{True}(t) \equiv 1, \forall t$).

We define the binary derivation relation for xT transducer M on partially transformed terms and derivation histories $T_{\Sigma\cup\Delta\cup Q} \times (\mathit{paths} \times R)^*$:

$$\Rightarrow_M \equiv \left\{ ((a,h),(b,h\cdot(i,(q,\lambda,\mathit{rhs},w)))) \mid (q,\lambda,\mathit{rhs},w) \in R \wedge \right.$$
$$i \in \mathit{paths}_a \wedge q = \mathit{label}_a(i) \wedge \lambda(a \downarrow (i\cdot(1))) = 1 \wedge$$
$$\left. b = a\left[i \leftarrow \mathit{rhs}\left[\begin{array}{r} p \leftarrow q'(a \downarrow (i\cdot(1)\cdot i')), \\ \forall p \in \mathit{paths}_{\mathit{rhs}} : \mathit{label}_{\mathit{rhs}}(p) = (q', i') \end{array} \right] \right] \right\}$$

That is, b is derived from a by application of a rule $\mathrm{q}\,\lambda \to^w \mathit{rhs}$ to an unprocessed input subtree $a \downarrow i$ which is in state q, replacing it by output given by *rhs* with variables (q', i') replaced by the input subtree at relative path i' in state q'.[3]

Let $\Rightarrow_M^*$, $D(M)$, $LD(M)$, w_M, W_M, and $\mathcal{L}(M)$ (the *weighted tree relation* of M) follow from the single-step $\Rightarrow_M$ exactly as they did in Section 3, except that the arguments are

3 Recall that $q(a)$ is the tree whose root is labeled q and whose single child is the tree a.

input and output instead of just output, with initial terms $Q_i(t)$ for each input $t \in T_\Sigma$ in place of S:

$$D(M) \equiv \left\{ (t, t', h) \in T_\Sigma \times T_\Delta \times (paths \times R)^* \mid (Q_i(t), ()) \Rightarrow_M^* (t', h) \right\}$$

We have given a rewrite semantics for our transducer, similar to wRTG. In the intermediate terms of a derivation, the active frontier of computation moves top-down, with everything above that frontier forming the top portion of the final output. The next rewrite always occurs somewhere on the frontier, and in a *complete derivation*, the frontier finally shrinks and disappears. In wRTG, the frontier consisted of the nonterminal-labeled leaves. In xT, the frontier items are not nonterminals, but pairs of state and input subtrees. We choose to represent these pairs as subtrees of terms with labels taken from $\Sigma \cup \Delta \cup Q$, where the state is the parent of the input subtree. In fact, given an $M \in xT$ and an input tree t, we can take all the (finitely many) pairs of input subtrees and states as nonterminals in a wRTG G, with all the (finitely many) possible single-step derivation rewrites of M applied to t as productions (taking the weight of the xT rule used), and the initial term $Q_i(t)$ as the start nonterminal, isomorphic to the derivations of the M which start with $Q_i(t)$: $(d, h) \in D(G)$ iff $(t, d, h) \in D(M)$. Such derivations are exactly how all the outputs of an input tree t are produced: when the resulting term d is in T_Δ, we say that (t, d) is in the tree relation and that d is an output of t.

Naturally, there may be input trees for which no complete derivation exists—such inputs are not in the domain of the weighted tree relation, having no output. It is known that $domain(M) \equiv \{i \mid \exists o, w : (i, o, w) \in \mathcal{L}(M)\}$, the set of inputs that produce any output, is always a recognizable tree language (Rounds 1970).

The *sources* of a rule $r = (q, l, rhs, w) \in R$ are the input-paths in the *rhs*:

$$sources(r) \equiv \{i' \mid \exists p \in paths_{rhs}(Q \times paths), q' \in Q : label_{rhs}(p) = (q', i')\}$$

If the sources of a rule refer to input paths that do not exist in the input, then the rule cannot apply (because $a \downarrow (i \cdot (1) \cdot i')$ would not exist). In the traditional statement of T, $sources(r)$ are the *variables* x_i, standing for the i^{th} child of the root at path (i), and the right hand sides of rules refer to them by name: (q_i, x_i). In xT, however, we refer to the mapped input subtrees by path (and we are not limited to the immediate children of the root of the subtree under transformation, but may choose any frontier of it).

A transducer is *linear* if for all its rules r, $sources(r)$ are a frontier and occur at most once: $\forall p_1, p_2 \in paths_{rhs}(Q \times paths), p \in paths - \{()\} : p_1 \neq p_2 \cdot p$. A transducer is *deterministic* if for any input, at most one rule matches per state:

$$\forall q \in Q, t \in T_\Sigma, r = (q, p, r, w), r' = (q', p', r', w') \in R : \\ p(t) = 1 \wedge p'(t) = 1 \implies r = r'$$

or in other words, the rules for a given state have patterns that partition possible input trees. A transducer is *deleting* if there are rules in which (for some matching inputs) entire subtrees are not used in their *rhs*.

In practice, we will be interested mostly in *concrete* transducers, where the patterns fully specify the labels and ranks of an input subtree including all the ancestors of $sources(r)$. Naturally, T are concrete. We have taken to writing concrete rules' patterns as trees with variables X in the leaves (at the sources), and using those same

variables in the *rhs* instead of writing the corresponding path in the *lhs*. For example: $q\ A(x_0{:}B, C)\ \rightarrow^{w}\ q'\ x_0$ means a xT rule (q, λ, rhs, w) with $rhs = (q', (1))$ and

$$\lambda \equiv (labelandrank_t(()) = (A, 1) \wedge label_t((1)) = B \wedge labelandrank_t((2)) = (C, 0))$$

It might be convenient to convert any xT transducer to an equivalent T transducer, then process it with T-based algorithms—in such a case, xT would just be syntactic sugar for T. We can automatically generate T productions that use extra states to emulate the finite lookahead and movement available in xT (as demonstrated in Section 1), but with one fatal flaw: Because of the definition of $\Rightarrow_M$, xT (and thus T) only has the ability to process input subtrees that produce corresponding output subtrees (alas, there is no such thing as an empty tree), and because ***TPAT*** can only inspect the root node while deriving replacement subtrees, T can check only the parts of the input subtree that lie along paths that are referenced in the *rhs* of the xT rule. For example, suppose we want to transform NP(DET, N) (but not, say, NP(ADJ, N)) into the tree N using rules in T. Although this is a simple xT rule, the closest we can get with T would be q NP(x0, x1) → q.N x1, but we cannot check both subtrees without emitting two independent subtrees in the output (which rules out producing just N). Thus, xT is a bit more powerful than T.

5. Parsing an xT Tree Relation

Derivation trees for a transducer $M = (\Sigma, \Delta, Q, Q_i, R)$ are T_R (trees labeled by rules) isomorphic to complete leftmost M-derivations. Figure 5 shows derivation trees for a particular transducer. In order to generate derivation trees for M automatically, we build a modified transducer M'. This new transducer produces derivation trees on its output instead of normal output trees. M' is (Σ, R, Q, Q_i, R'), with[4]

$$R' \equiv \{(q, \lambda, r(yield_{rhs}(Q \times paths)), w) \mid r = (q, \lambda, rhs, w) \in R\}$$

That is, the original *rhs* of rules are flattened into a tree of depth 1, with the root labeled by the original rule, and all the non-expanding Δ-labeled nodes of the *rhs* removed, so that the remaining children are the nonterminal yield in left to right order. Derivation trees deterministically produce a single weighted output tree, and for concrete transducers, a single input tree.

For every leftmost derivation there is exactly one corresponding derivation tree: We start with a sequence of leftmost derivations and promote rules applied to paths that are prefixes of rules occurring later in the sequence (the first will always be the root), or, in the other direction, list out the rules of the derivation tree in order.[5] The weights of derivation trees are, of course, just the product of the weights of the rules in them.[6]

The derived transducer M' nicely produces derivation trees for a given input, but in explaining an observed (input/output) pair, we must restrict the possibilities further. Because the transformations of an input subtree depend only on that subtree and its state, we can build a compact wRTG that produces exactly the weighted derivation trees corresponding to M-transductions $(I, ()) \Rightarrow^*_M (O, h)$ (Algorithm 1).

4 By $r((t_1, \ldots, t_n))$, we mean the tree $r(t_1, \ldots, t_n)$.

5 Some path concatenation is required, because paths in histories are absolute, whereas the paths in rule *rhs* are relative to the input subtree.

6 Because our product is commutative, the order does not matter.

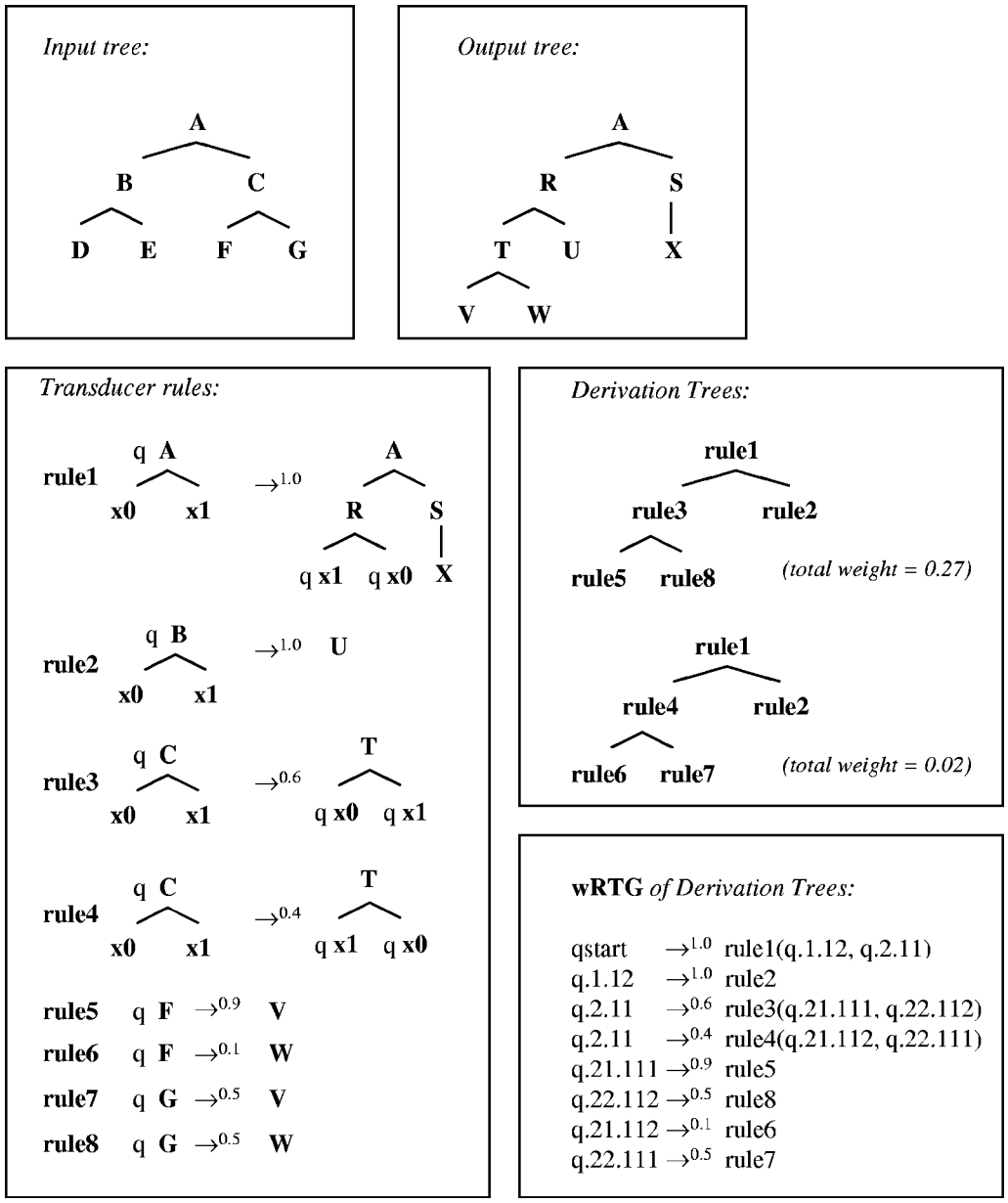


Figure 5
Derivation trees for a T tree transducer.

Algorithm 1 makes use of **memoization**—the possible derivations for a given (q, i, o) are constant, so we store answers for all past queries in a lookup table and return them, avoiding needless recomputation. Even if we prove that there are no derivations for some (q, i, o), successful subhypotheses met during the proof may recur and are kept, but we do avoid adding productions we know can't succeed. We have in the worst case to visit all $|Q| \cdot |I| \cdot |O|$ (q, i, o) pairs and apply all $|R|$ transducer rules successfully at each of them, so time and space complexity, proportional to the size of the (unpruned) output wRTG, are both $O(|Q| \cdot |I| \cdot |O| \cdot |R|)$, or $O(Gn^2)$, where n is the total size of the

Algorithm 1. ***Deriv*** (derivation forest for $I \Rightarrow^*_{xT} O$)

Input: xT transducer $M = (\Sigma, \Delta, Q, Q_i, R)$ and observed tree pair $I \in T_\Sigma$, $O \in T_\Delta$.
Output: derivation wRTG $G = (R, N \subseteq Q \times paths_I \times paths_O, S, P)$ generating all weighted derivation trees for M that produce O from I. Returns *false* instead if there are no such trees. $O(G|I||O|)$ time and space complexity, where G is a grammar constant.

begin
  $S \leftarrow (Q_i, (), ())$, $N \leftarrow \emptyset$, $P \leftarrow \emptyset$, $memo \leftarrow \emptyset$
  if $\mathbf{PRODUCE}_{I,O}(S)$ **then**
    $N \leftarrow \{n \mid \exists (n', rhs, w) \in P : n = n' \vee n \in yield_{rhs}(Q \times paths_I \times paths_O)\}$
    return $G = (R, N, S, P)$
  else
    return *false*
end

PRODUCE$_{I,O}(\alpha = (q, i, o) \in Q \times paths_I \times paths_O)$ ***returns boolean*** $\equiv$ **begin**
  if $\exists (\alpha, r) \in memo$ **then return** r
  $memo \leftarrow memo \cup \{(\alpha, true)\}$
  anyrule? $\leftarrow$ *false*
  for $r = (q, \lambda, rhs, w) \in R : \lambda(I \downarrow i) = 1 \wedge \mathbf{Match}_{O,\Delta}(rhs, o)$ **do**
    $(o_1, \ldots, o_n) \leftarrow paths_{rhs}(Q \times paths)$ sorted by $o_1 <_{lex} \ldots <_{lex} o_n$ //$n = 0$ if there are no *rhs* variables
    $labelandrank_{derivrhs}(()) \leftarrow (r, n)$ //*derivrhs* `is a newly created tree`
    for $j \leftarrow 1$ ***to*** n **do**
      $(q', i') \leftarrow label_{rhs}(o_j)$
      $\beta \leftarrow (q', i \cdot i', o \cdot o_j)$
      if $\neg$**PRODUCE**$_{I,O}(\beta)$ **then next** r
      $labelandrank_{derivrhs}((j)) \leftarrow (\beta, 0)$
    anyrule? $\leftarrow$ *true*
    $P \leftarrow P \cup \{(\alpha, derivrhs, w)\}$
  $memo \leftarrow memo \cup \{(\alpha$,*anyrule?*$)\}$
  return *anyrule?*
end

$\mathbf{Match}_{t,\Sigma}(t', p) \equiv \forall p' \in path(t') : label(t', p') \in \Sigma \implies labelandrank_{t'}(p') = labelandrank_t(p \cdot p')$

input and output trees, and G is the grammar constant accounting for the states and rules (and their size).

If the transducer contains cycles of state-change rules, then the generated derivation forest may have infinitely many trees in it, and thus the memoization of ***PRODUCE*** must temporarily assume that the alignment (q, i, o) under consideration will succeed upon reaching itself, through such a cycle, even though the answer is not yet conclusive (it may be conclusively *true*, but not *false*). Although it would be possible to detect these cycles (setting "pending" rather than *true* for the interim in *memo*) and deal with them more severely, we can just remove the surplus later in linear time, using Algorithm 2, which is an implementation (for wRTG) of a well-known method of pruning useless

Algorithm 2. ***RTGPrune*** (wRTG useless nonterminal/production identification)

Input: wRTG $G = (\Sigma, N, S, P)$, with $P = (p_1, \ldots, p_m)$ and $p_i = (q_i, t_i, w_i)$.
Output: For all $n \in N$, $B[n] = (\exists t \in T_\Sigma : n \Rightarrow_G^* t)$ (*true* if n derives some output tree t with no remaining nonterminals, *false* if it's useless), and $A[n] = (\exists t \in T_\Sigma, t' \in T_\Sigma(\{n\}) : S \Rightarrow_G^* t' \Rightarrow_G^* t)$ (n additionally can be produced from an S using only productions that can appear in complete derivations). Time and space complexity are linear in the total size of the input: $O(|N| + \sum_{i=1}^{m} (1 + |paths_{t_i}|))$

```
begin
    M ← ∅
    for n ∈ N do B[n] ← false, Adj[n] ← ∅
    for i ← 1 to m do
        Y ← {label_{t_i}(p) | p ∈ paths_{t_i}(N)}
        // Y are the unique N in rhs of rule i
        for n ∈ Y do Adj[n] ← Adj[n] ∪ {i}
        if |Y| = 0 then M ← M ∪ {i}
        r[i] ← |Y|
    for n ∈ M do REACH(n)
    /* Now that B[n] are decided, compute A[n]                    */
    for n ∈ N do A[n] ← false
    USE(S)
end

REACH(n) ≡ begin
    B[n] ← true
    for i ∈ Adj[n] do
        if ¬B[q_i] then
            r[i] ← r[i] − 1
            if r[i] = 0 then REACH(q_i)
end

USE(n) ≡ begin
    A[n] ← true
    for n' s.t. ∃(n, t, w) ∈ R : n' ∈ yield_t(N) do
        /* for n' that are in the rhs of rules whose lhs is n     */
        if ¬A[n'] ∧ B[n'] then USE(n')
end
```

productions from a CFG (Hopcroft and Ullman 1979).[7] We eliminate *all* the remains of failed subforests, by removing all nonterminals n, and any productions involving n, where Algorithm 2 gives $A[n] = false$.

In the next section, we show how to compute the contribution of a nonterminal to the weighted trees produced by a wRTG, in a generalization of Algorithm 2 that gives us weights that we accumulate per rule over the training examples, for EM training.

7 The idea is to first remove all nonterminals (and productions referring to them) that don't yield any terminal string, and after that, to remove those which are not reachable top-down from S.

6. Inside–Outside for wRTG

Given a wRTG $G = (\Sigma, N, S, P)$, we can compute the sums of weights of trees derived using each production by adapting the well-known inside–outside algorithm for weighted context-free (string) grammars (Lari and Young 1990).

Inside weights β_G for a nonterminal or production are the sum of weights of all trees that can be derived from it:

$$\beta_G(n \in N) \equiv \sum_{(n,r,w)\in P} w \cdot \beta_G(r)$$

$$\beta_G(r \in T_\Sigma(N) \mid (n,r,w) \in P\}) \equiv \prod_{p\in paths_r(N)} \beta_G(label_r(p))$$

By definition, $\beta_G(S)$ gives the sum of the weights of all trees generated by G. For the wRTG generated by $\boldsymbol{Deriv}(M, I, O)$, this is exactly $W_M(I, O)$.

The recursive definition of β does not assume a non-recursive wRTG. In the presence of derivation cycles with weights less than 1, β can still be evaluated as a convergent sum over an infinite number of trees.

The output of ***Deriv*** will always be non-recursive provided there are no cycles of ϵ-rules in the transducer. There is usually no reason to build such cycles, as the effect (in the unweighted case) is just to make all implicated states equivalent.

Outside weights α_G are for each nonterminal the sums over all its occurrences in complete derivations in the wRTG of the weight of the whole tree, excluding the occurrence subtree weight (we define this without resorting to division for cancellation, but in practice we may use division by $\beta_G(n)$ to achieve the same result).

$$\alpha_G(n \in N) \equiv \begin{cases} 1 & \text{if } n = S \\ \overbrace{\sum_{p,(n',r,w)\in P:label_r(p)=n} w \cdot \alpha_G(n')}^{\text{uses of n in productions}} \cdot \underbrace{\prod_{p'\in paths_r(N)-\{p\}} \beta_G(label_r(p'))}_{\text{sibling nonterminals}} & \text{otherwise.} \end{cases}$$

Provided that useless nonterminals and productions were removed by Algorithm 2, and none of the rule weights are 0, all of the nonterminals in a wRTG will have nonzero α and β. Conversely, if useless nonterminals weren't removed, they will be detected when computing inside–outside weights by virtue of their having zero values, so they may be safely pruned without affecting the generated weighted tree language.

Finally, given inside and outside weights, the sum of weights of trees using a particular production is $\gamma_G((n,r,w) \in P) \equiv \alpha_G(n) \cdot w \cdot \beta_G(r)$. Here we rely on the commutativity of the product (the left-out inside part reappears on the right of the inside part, even when it wasn't originally the last term).

Computing α_G and β_G for nonrecursive wRTG is a straightforward translation of the recursive definitions (using memoization to compute each result only once) and is $O(|G|)$ in time and space. Or, without using memoization, we can take a topological sort

using the dependencies induced by the equations for the particular forest, and compute in that order. In case of a recursive wRTG, the equations may still be solved (usually iteratively), and it is easy to guarantee that the sums converge by appropriately keeping the rule weights of state-change productions less than one.

7. EM Training

Expectation-Maximization (EM) training (Dempster, Laird, and Rubin 1977) works on the principle that the likelihood (product over all training examples of the sum of all model derivations for it) can be maximized subject to some normalization constraint on the parameters,[8] by repeatedly:

1. Computing the *expectation* of decisions taken for all possible ways of generating the training corpus given the current parameters, accumulating (over each training example) parameter *counts* $\vec{c}$ of the portion of all possible derivations using that parameter's decision:

$$\forall \tau \in \mathit{parameters} :$$
$$c_\tau \equiv E_{t \in \mathit{training}} \left[\frac{\sum_{d \in \mathit{derivations}_t} (\#\text{ of times } \tau \text{ used in } d) \cdot p_{\mathit{parameters}}(d)}{\sum_{d \in \mathit{derivations}_t} p_{\mathit{parameters}}(d)} \right]$$

2. *Maximizing* by assigning the counts to the parameters and renormalizing:

$$\forall \tau \in \mathit{parameters} : \tau \leftarrow \frac{c_\tau}{Z_\tau(\vec{c})}$$

Each iteration is guaranteed to increase the likelihood until a local maximum is reached. Normalization may be affected by tying or fixing of parameters. The derivations for training examples do not change, but the model weights for them do. Using inside–outside weights, we can efficiently compute these weighted sums over all derivations for a wRTG, and thus, using Algorithm 1, over all xT derivations explaining a given input/output tree pair.

A simpler version of ***Deriv*** that computes derivation trees for a wRTG given an output tree could similarly be used to train weights for wRTG rules.[9]

Each EM iteration takes time linear in the size of the transducer and linear in the size of the derivation tree grammars for the training examples. The size of the derivation trees is at worst $O(Gn^2)$, so for a corpus of N examples with maximum input/output size n, an iteration takes at worst time $O(NGn^2)$. Typically, we expect only a small fraction of possible states and rules will apply to a given input/output subtree mapping.

8 Each parameter gives the probability of a single model decision, and a derivation's probability is the product of all the decisions producing it.

9 One may also use ***Deriv*** unmodified to train an identity (or constant-input) transducer with one rule per wRTG production, having exactly the range of the wRTG in question, and of course transforming training trees to appropriate tree pairs.

The recommended normalization function computes the sum of all the counts for rules having the same state, which results in trained model weights that give a joint probability distribution over input/output tree pairs.

Attempts at conditional normalization can be problematic, unless the patterns for all the rules of a given state can be partitioned into sets so that for any input, only patterns from at most one set may match. For example, if all the patterns specify the label and rank of the root, then they may be partitioned along those lines. *Input-epsilon* rules, which always match (with pattern ***True***), would make the distribution inconsistent by adding extra probability mass, unless they are required (in what is no longer a partition) to have their counts normalized against *all* the partitions for their state (because they transform inputs that could fall in any of them).

One can always marginalize a joint distribution for a particular input to get true conditional probabilities. In fact, no method of assigning rule weights can generally compute exact conditional probabilities; remarginalization is already required: take as the normalization constant the inside weight of the root derivation forest corresponding to all the derivations for the input tree in question.

Even using normalization groups that lead to inconsistent probability distributions, EM may still compute some empirically useful local maximum. For instance, placing each *q lhs* in its own normalization group might be of interest; although the inside weights of a derivation forest would sum to some $s > 1$, ***Train*** would divide the counts earned by each participating rule by s (Algorithm 3).

8. Strings

We have covered tree-to-tree transducers; we now turn to tree-to-string transducers. In the automata literature, such transductions are called **generalized syntax-directed translation** (Aho and Ullman 1971), and are used to specify compilers that (deterministically) transform high-level source-language trees into linear target-language code. Tree-to-string transducers have also been applied to the machine translation of natural languages (Yamada and Knight 2001; Eisner 2003). Tree-to-string transduction is appealing when trees are only available on the input side of a training corpus. Furthermore, tree/string relationships are less constrained than tree/tree, allowing the possibility of simpler models to account for natural language transformations. (Though we will not pursue it here, string-to-string training should also be possible with tree-based models, if only string-pair data is available; string/string relations induced by tree transformations are sometimes called *translations* in the automata literature.)

$\Sigma^\star$ are the *strings over alphabet* Σ. For $s = (s_1, \ldots, s_n)$, the *length* of s is $|s| \equiv n$ and the i^{th} *letter* is $s[i] \equiv s_i$, for all $i \in indices_s \equiv \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$. $indices_s(X)$ is the subset $\{i \in indices_s \mid i[s] \in X\}$. The *letters in* s are $letters_s = \{l | \exists i \in indices_s : s[i] = l\}$. The *spans* of s are $spans_s = \{(a,b) \in \{\mathbb{N}^2 \mid 1 \leq a \leq b \leq n+1\}$, and the *substring at span* $p = (a,b)$ of s is $s \downarrow p \equiv (s_a, \ldots s_{b-1})$, with $s \downarrow (a,a) = ()$. We use the shorthand $[i] \equiv (i, i+1)$ for all $i \in \mathbb{N}$, so $s \downarrow [i] = s[i]$. The *substitution of* t *for a span* $(a,b) \in spans_s$ *in* s is $s[(a,b) \leftarrow t] \equiv (s \downarrow (1,a)) \cdot t \cdot (s \downarrow (b, n+1))$.[10]

A *partition* is a set of non-overlapping spans P—$\forall (a,b),(c,d) \in P : c \leq d \leq a \vee b \leq c \leq d \vee (a,b) = (c,d)$, and the *parallel substitution of* s'_p *for the partition* P *of* s is written $s[p \leftarrow s'_p, \forall p \in P]$. In contrast to parallel tree substitution, we cannot take any

10 $a \cdot b$ is string concatenation, defined already in Section 2.

composition of the individual substitutions, because the replacement substrings may be of different length, changing the referent of subsequent spans. It suffices to perform a series of individual substitutions, in right to left order—$(a_n, b_n), \ldots, (a_i, b_i), \ldots, (a_1, b_1)$ $(a_i \geq b_{i+1}, \forall 1 \leq i < n)$.

Algorithm 3. ***Train*** (EM training for tree transducers)

Input: xR transducer $M = (\Sigma, \Delta, Q, Q_d, R)$ with initial rule weights, observed weighted tree pairs $T \in T_\Sigma \times T_\Delta \times \mathbb{R}^+$, minimum relative log-likelihood change for convergence $\epsilon \in \mathbb{R}^+$, maximum number of iterations $maxit \in \mathbb{N}$, and for each rule $r \in R$: prior counts (for a *Dirichlet prior*) $prior : R \mapsto \mathbb{R}$ for smoothing, and normalization function $Z_r : (R \mapsto \mathbb{R}) \mapsto \mathbb{R}$ used to update weights from counts $w'_r \leftarrow count(r)/Z_r(count)$.

Output: New rule weights $W \equiv \{w_r \mid r \in R\}$.

begin

  for $(i, o, w) \in T$ **do**

    $d_{i,o} \leftarrow$ ***Deriv***(M, i, o) `// Algorithm 1`

    if $d_{i,o} = false$ **then**

      $T \leftarrow T - \{(i, o, w)\}$

      Warn(more rules are needed to explain (i, o))

    Compute inside–outside weights for $d_{i,o}$

    If Algorithm 2 (***RTGPrune***) has not already been used to do so, remove all useless nonterminals n (and associated rules) whose $\beta_{d_{i,o}}(n) = 0$ or $\alpha_{d_{i,o}}(n) = 0$

  $i \leftarrow 0, L \leftarrow -\infty, \delta \leftarrow \epsilon$

  for $r = (q, \lambda, rhs, w) \in R$ **do** $w_r \leftarrow w$

  while $\delta \geq \epsilon \wedge i < maxit$ **do**

    for $r \in R$ **do** $count[r] \leftarrow prior(r)$

    $L' \leftarrow 0$

    for $(i, o, w_{example}) \in T$ `// Estimate`

    do

      let $D \equiv d_{i,o} \equiv (R, N, S, P)$

      compute α_D, β_D using latest $W \equiv \{w_r \mid r \in R\}$ `// see Section 6`

      for $\rho = (n, rhs, w) \in P$ **do**

        $\gamma_D(\rho) \leftarrow \alpha_D(n) \cdot w \cdot \beta_D(rhs)$

        let $r \equiv label_{rhs}(())$

        $count[r] \leftarrow count[r] + w_{example} \cdot \frac{\gamma_D(\rho)}{\beta_D(S)}$

      $L' \leftarrow L' + \log \beta_D(S) \cdot w_{example}$

    for $r \in R$ `// Maximize`

    do

      $w_r \leftarrow \frac{count[r]}{Z_r(count)}$ `// e.g., joint`

      $Z_r(\vec{c}) \equiv \sum_{r'=(q_r, d, e, f) \in R} c(r'), \forall r = (q_r, \lambda, rhs, w) \in R$

    $\delta \leftarrow \frac{L' - L}{|L'|}$

    $L \leftarrow L', i \leftarrow i + 1$

end

9. Extended Tree-to-String Transducers (xTs)

A *weighted extended-lhs root-to-frontier tree-to-string transducer* M is a quintuple $(\Sigma, \Delta, Q, Q_i, R)$ where Σ is the input alphabet, Δ is the output alphabet, Q is a finite set of states, $Q_i \in Q$ is the *initial (or start, or root) state*, and $R \subseteq Q \times \boldsymbol{xTPAT}_\Sigma \times (\Delta \cup (Q \times \mathit{paths}))^\star \times \mathbb{R}^+$ is a finite set of *weighted transformation rules*, written $\mathrm{q}\,\lambda \rightarrow^w \mathit{rhs}$. A rule says that to transform an input subtree matching λ while in state q, replace it by the string of *rhs* with its nonterminal ($Q \times \mathit{paths}$) letters replaced by their (recursive) transformation.

xTs is the same as xT, except that the *rhs* are strings containing some nonterminals instead of trees containing nonterminal leaves. By taking the yields of the *rhs* of an xT transducer's rules, we get an xTs that derives exactly the weighted strings that are the yields of the weighted trees generated by its progenitor.

As discussed in Section 1, we may consider strings as isomorphic to degenerate, monadic-spined right-branching trees, for example, the string (a, b, c) is the tree $C(a, C(b, C(c, \mathit{END})))$. Taking the yield of such a tree, but with *END* yielding the empty string, we have the corresponding string. We choose this correspondence instead of flat trees (e.g., $C(a, b, c)$) because our derivation steps proceed top-down, choosing the states for all the children at once (what's more, we don't allow symbols C to have arbitrary rank). If all the *rhs* of an xTs transducer are transformed into such trees, then we have an xT transducer. The yields of that transducer's output trees for any input are the same as the outputs of the xTs transducer for the same input, but again, only if *END* is considered to yield the empty string. Note that in general the produced output trees will not have the canonical right-branching monadic spine that we use to encode strings,[11] so that yield-taking is a nontrivial operation. Finally, consider that for a given transducer, the same output yield may be derived via many output trees, which may differ in the number and location of *END*, and in the branching structure induced by multi-variable *rhs*. Because this leads to additional difficulties in inferring the possible derivations given an observed output string, we must study tree-to-string relations apart from tree relations.

Just as wRTG can generate PCFG derivation trees, xTs can generate tree/string pairs comparable to a Synchronous CFG (SCFG), with the tree being the CFG derivation tree of the SCFG input string, with one caveat: an *epsilon* leaf symbol (we have used *END*) must be introduced which must be excluded from yield-taking, after which the string-to-string translations are identical.

We define the binary derivation relation on $(\Delta \cup (Q \times T_\Sigma))^\star \times (\mathbb{N} \times R)^*$ (strings of output letters and state-labeled input trees and their derivation history)

$$\Rightarrow_M \equiv \left\{ ((a,h),(b,h\cdot(i,(q,\lambda,\mathit{rhs},w)))) \;\middle|\; \exists (q,\lambda,\mathit{rhs},w) \in R, i \in \mathit{indices}_a : \begin{array}{r} a[i] = (q,I) \in Q \times T_\Sigma \wedge \\ \lambda(I) = 1 \wedge \\ b = a\left[[i] \leftarrow \mathit{rhs}\left[\begin{array}{r}[p] \leftarrow (q', I \downarrow i'),\\ \forall p \in \mathit{indices}_{\mathit{rhs}} : \mathit{rhs}[p] = (q', i') \in Q \times \mathit{paths}\end{array}\right]\right] \end{array} \right\}$$

11 In the special case that all *rhs* contain at most one variable, and that every variable appears in the final position of its *rhs*, the output trees do, in fact, have the same canonical monadic-spined form. For these transducers there is no meaningful difference between xTs and xT.

where at position i, an input tree I (labeled by state q) in the string a is replaced by a *rhs* from a rule that matches it. Of course, the variables $(q', i') \in Q \times \mathit{paths}$ in the *rhs* get replaced by the appropriate pairing of $(q', I \downarrow i')$. Each rewrite flattens the string of trees by breaking one of the trees into zero or more smaller trees, until (in a complete derivation) only letters from the output alphabet Δ remain. As with xT, rules may only apply if the paths in them exist in the input (if $i' \in \mathit{paths}_I$), even if the tree pattern doesn't mention them.

Let $\Rightarrow_M^*$, $D(M)$, $LD(M)$, w_M, W_M, and $\mathcal{L}(M)$ (the *weighted tree-to-string relation* of M) follow from the single-step $\Rightarrow_M$ exactly as they did in Section 4.[12]

10. Parsing an xTs Tree-to-String Relation

Derivation trees for an xTs transducer are defined by an analogous xT transducer, exactly as they were for derivation trees for xT, where the nodes are labeled by rules to be applied preorder, with the i^{th} child rewriting the i^{th} variable in the *rhs* of its parent node.

Algorithm 4 (***SDeriv***) is the tree-to-string analog of Algorithm 1 (***Deriv***), building a tree grammar that generates all the weighted derivation trees explaining an observed input tree/output string pair for an xTs transducer.

SDeriv differs from ***Deriv*** in the use of arbitrary output string spans instead of output subtrees. The looser alignment constraint causes additional complexity: There are $O(m^2)$ spans of an observed output string O of length m, and each binary production over a span has $O(m)$ ways of dividing the span in two (we also have the n different input subtrees and q different rule states).

There is no way to fix in advance a tree structure over the training example and transducer rule output strings without constraining the derivations to be consistent with the bracketing. Another way to think of this is that any xTs derivation implies a specific tree bracketing over the output string. In order to compute the derivations using the tree-to-tree ***Deriv***, we would have to take the union of forests for all the possible output trees with the given output yield.

SDeriv takes time and space linear to the size of the output: $O(Gnm^3)$ where G combines the states and rules into a single grammar constant, and n is the size of the input tree. The reduced $O(m^2)$ space bound from 1-best CFG parsing does not apply, because we want to keep all successful productions and split points, not only the best for each item.

We use the presence of terminals in the right hand side of rules to constrain the alignments of output subspans to nonterminals, giving us minimal-sized subproblems tackled by ***VarsToSpan***.

The canonicalization of same-substring spans is most obviously applicable to zero-length spans (which become $(1, 1)$, no matter where they arose), but in the worst case, every input label and output letter is unique, so nothing further is gained. Canonicalization may also be applied to input subtrees. By canonicalizing, we effectively name subtrees and substrings by value, instead of by path/span, increasing best-case sharing and reducing the size of the output. In practice, we still use paths and spans, and hash to a canonical representative if desired.

12 Because the locations in derivation histories are string indexes now rather than tree paths, we use the usual $<$ on naturals as the ordering constraint for leftmost derivations.

Algorithm 4. ***SDeriv*** (derivation forest for $I \Rightarrow^*_{xTs} O$)

Input: xTs transducer $M = (\Sigma, \Delta, Q, Q_i, R)$, observed input tree $I \in T_\Sigma$, and output string $O = (o_1, \ldots, o_n) \in \Delta^*$

Output: derivation wRTG $G = (R \cup \{\natural\}, N \subseteq N', S, P)$ generating all weighted derivation trees for M that produce O from I, with $N' \equiv ((Q \times paths_I \times spans_O) \cup (paths_I \times spans_O \times (Q \times paths)^*))$. Returns *false* instead if there are no such trees.

begin
  $S \leftarrow (Q_i, (), (1, n))$, $N \leftarrow \emptyset$, $P \leftarrow \emptyset$, $memo \leftarrow \emptyset$
  if $\mathbf{PRODUCE}_{I,O}(S)$ **then**
    $N \leftarrow \{n \mid \exists (n', rhs, w) \in P : n = n' \vee n \in yield_{rhs}(N')\}$
    return $G = (R \cup \{\natural\}, N, S, P)$
  else
    return *false*
end

PRODUCE$_{I,O}(\alpha = (q \in Q, in \in paths_I, out = (a, b) \in spans_O))$ ***returns boolean*** $\equiv$ **begin**
  if $\exists (\alpha, r) \in memo$ **then return** r
  $memo \leftarrow memo \cup \{(\alpha, true)\}$
  $anyrule? \leftarrow false$
  for $rule = (q, pat, rhs, w) \in R : pat(I \downarrow in) = 1 \ \wedge$ $\mathbf{Feasible}_O(rhs, out)$ **do**
    $(r_1, \ldots, r_k) \leftarrow indices_{rhs}(\Delta)$ in increasing order
    /* $k \leftarrow 0$ `if there are none` */
    $p_0 \leftarrow a - 1$, $p_{k+1} \leftarrow b$
    $r_0 \leftarrow 0$, $r_{k+1} \leftarrow |rhs| + 1$
    for $p = (p_1, \ldots, p_k) : (\forall 1 \le i \le k : O[p_i] = rhs[r_i]) \wedge$
    $(\forall 0 \le i \le k : p_k < p_{k+1} \wedge (r_{k+1} - r_k = 1 \implies p_{k+1} - p_k = 1))$ **do**
      /* `for all alignments` p `between` $rhs[r_i]$ `and` $O[p_i]$, `such that order, beginning/end, and immediate adjacencies in` rhs `are observed in` O. `The degenerate` $k = 0$ `has just` $p = ()$. */
      $label_{derivrhs}(()) \leftarrow (rule)$
      $v \leftarrow 0$
      for $i \leftarrow 0$ ***to*** k **do**
        /* `variables` $rhs \downarrow (r_i + 1, r_{i+1})$ `must generate` $O \downarrow (p_i + 1, p_{i+1})$ */
        if $r_i + 1 = r_{i+1}$ **then next** i
        $v \leftarrow v + 1$
        $spangen \leftarrow (in, (p_i + 1, p_{i+1}), rhs \downarrow (r_i + 1, r_{i+1}))$
        $n \leftarrow \mathbf{VarsToSpan}_{I,O}(spangen)$
        if $n = false$ **then next** p
        $labelandrank_{derivrhs}((v)) \leftarrow (n, 0)$
      $anyrule? \leftarrow true$
      $rank_{derivrhs}(()) = v$
      $P \leftarrow P \cup \{\alpha, derivrhs, w)\}$
  $memo \leftarrow memo \cup \{(\alpha, anyrule?)\}$
  return *anyrule?*
end

Feasible$_O(rhs, span) \equiv \forall l \in letters_{rhs} : l \in \Delta \implies l \in letters_{O \downarrow span}$

Algorithm SDeriv (cont.) $\natural$-labeled nodes are generated as artifacts of sharing by cons-nonterminals of derivations for the same spans.

VarsToSpan$_{I,O}$
($wholespan = (in \in paths_I, out = (a,b) \in spans_O, nonterms \in (Q \times paths)^*)$) ***returns*** $N' \cup \{false\} \equiv$

```
/* Adds all the productions that can be used to map from parts of the
   nonterminal string referring to subtrees of I↓in into O↓out and
   returns the appropriate derivation-wRTG nonterminal if there was a
   completely successful derivation, or false otherwise.                */
```

begin
  $ret \leftarrow false$
  if $|nonterms| = 1$ **then**
    $(q', i') \leftarrow nonterms[1]$
    if PRODUCE$_{I,O}(q', in \cdot i', out)$ **then return** $(q', in \cdot i', out)$
    return *false*
  $wholespan \leftarrow (in,$ ***CANONICAL***$_O(out), nonterms)$
  if $\exists (wholespan, r) \in memo$ **then return** r
  for $s \leftarrow b$ ***to*** a **do**
    /* the first nonterminal will cover the span (a,s)   */
    $(q', i') \leftarrow nonterms[1]$ /* nonterms will never be empty   */
    $spanfirst \leftarrow (q', i \cdot i', (a, s))$
    if $\neg$**PRODUCE**$_{I,O}(spanfirst)$ **then next** s
    $label_{spanlist}(()) \leftarrow \natural$
    /* cons node for sharing; left child expands to rules used for this
       nonterminal, right child expands to rest of nonterminal/span
       derivation   */
    $labelandrank_{spanlist}((1)) \leftarrow (spanfirst, 0)$
    /* first child: expansions of first nonterminal   */
    $rank_{spanlist}(()) \leftarrow 2$
    $spanrest \leftarrow (in, (s, b), nonterms \downarrow (2, |nonterms| + 1))$
    /* second child: expansions of rest of nonterminals   */
    $n \leftarrow$ ***VarsToSpan***$_{I,O}(spanrest)$
    if $n = false$ **then next** s
    $labelandrank_{spanlist}((2)) \leftarrow (n, 0)$
    $P \leftarrow P \cup (wholespan, spanlist, 1)$
    $ret \leftarrow wholespan$
  $memo \leftarrow memo \cup \{(wholespan, ret)\}$
  return ret
end

CANONICAL$_O((a,b)) \equiv \min\{(x,y) \mid O \downarrow (x,y) = O \downarrow (a,b) \wedge x \geq 1\}$

The enumeration of matching rules and alignments of terminals in the rule *rhs* to positions in the output substring is best interleaved; the loops are nested for clarity of presentation only. We use an FSA of subsequences of the output string (skipping forward to a desired letter in constant time with an index on outgoing transitions), and a trie of the rules' outputs (grouping by collapsing *rhs* variable sequences into a single "skip" symbol), and intersect them, visiting alignments and sets of rules in the rule

index. The choice of expansion sites against an input subtree proceeds by exhaustive backtracking, since we want to enumerate all matching patterns. Each of these sets of rules is further indexed against the input tree in a kind of leftmost trie.[13] ***Feasible*** is redundant in the presence of such indexing.

Static grammar analysis could also show that certain transducer states always (or never) produce an empty string, or can only produce a certain subset of the terminal alphabet. Such proofs would be used to restrict the alignments considered in ***VarsToSpan***.

We have modified the usual derivation tree structure to allow sharing the ways an output span may align to a *rhs* substring of multiple consecutive variables; as a consequence, we must create some non-rule-labeled nodes, labeled by ♮ (with rank 2). ***Train*** collects counts only for rule-labeled nodes, and the inside–outside weight computations proceed in ignorance of the labels, so we get the same sums and counts as if we had non-binarized derivation trees. Instead of a consecutive *rhs* variable span of length n generating n immediate rule-labeled siblings, it generates a single right-branching binarized list of length n with each suffix generated from a (shared) nonterminal. As in LISP, the left child is the first value in the list, and the right child is the (binarized) rest of the list. As the base case, we have $♮(n_1, n_2)$ as a list of two nonterminals (single variable runs refer to their single nonterminal directly without any ♮ wrapping; we use no explicit null list terminator). Just as in CFG parsing, it would be necessary without binarization to consider exponentially many productions, corresponding to choosing an n-partition of the span length; the binarized nonterminals in our derivation RTG effectively share the common suffixes of the partitions.

SDeriv could be restated in terms of parsing with a binarized set of rules, where only some of the binary nonterminals have associated input trees; however, this would complicate collecting counts for the original, unbinarized transducer rules.

If there are many cyclical state-change transitions (e.g., $q\, x_0 \rightarrow q'\, x_0$), a nearly worst-case results for the memoized top-down recursive descent parsing of ***SDeriv***, because for every reachable alignment, nearly every state would apply (but after pruning, the training proceeds optimally). An alternative bottom-up ***SDeriv*** would be better suited in general to input-epsilon heavy transducers (where there is no tree structure consumed to guide the top-down choice of rules). The worst-case time and space bounds would be the same, but (output) lexical constraints would be used earlier.

The weighted derivation tree grammar produced by ***SDeriv*** may be used (after removing useless productions with Algorithm 2) exactly as before to perform EM training with ***Train***. In doing so, we generalize the standard inside–outside training of probabilistic context-free grammar (PCFG) on raw text (Baker 1979). In Section 12, we demonstrate this by creating an xTs transducer that transforms a fixed single-node dummy tree to the strings of some arbitrary CFG, and train it on a corpus in which the dummy input tree is paired with each training string as its output.

11. Translation Modeling Experiment

It is possible to cast many current probabilistic natural language models as T-type tree transducers. In this section, we implement the translation model of Yamada and Knight (2001) and train it using the EM algorithm.

13 To make a trie of complete tree patterns, represent them canonically as strings interleaving *paths* leftmost for expansion, and *labelandrank* that must agree with the concurrent location in the input tree.

Figure 6 shows a portion of the bilingual English-tree/Japanese-string corpus used in Yamada and Knight (2001) and here. Figures 7 and 8 show the generative model and parameters; the parameter values shown were learned via specialized EM re-estimation formulae described in this article's appendix. According to the model, an English tree becomes a Japanese string in four steps.

First, every node is re-ordered, that is, its children are permuted probabilistically. If there are three children, then there are six possible permutations whose probabilities add up to 1. The re-ordering depends only on the child label sequence, and not on any wider or deeper context. Note that the English trees in Figure 6 are already flattened in pre-processing because the model cannot perform complex re-orderings such as the one we described in Section 1, S(PRO,VP(V,NP)) → V, PRO, NP.

```
------------------------------------------------------------
ENGLISH:    (VB (NN hypocrisy)
                (VB is)
                (JJ (JJ abhorrent)
                    (TO (TO to) (PRP them))))
JAPANESE:   kare ha gizen ga daikirai da
------------------------------------------------------------
ENGLISH:    (VB (PRP he)
                (VB has)
                (NN (JJ unusual) (NN ability))
                (IN (IN in) (NN english)))
JAPANESE:   kare ha eigo ni zubanuke-ta sainou wo mot-te iru
------------------------------------------------------------
ENGLISH:    (VB (PRP he)
                (VB was)
                (JJ (JJ ablaze)
                    (IN (IN with) (NN anger))))
JAPANESE:   kare ha mak-ka ni nat-te okot-te i-ta
------------------------------------------------------------
ENGLISH:    (VB (PRP i)
                (VB abominate)
                (NN snakes))
JAPANESE:   hebi ga daikirai da
------------------------------------------------------------

    etc.
```

Figure 6
A portion of a bilingual tree/string training corpus.

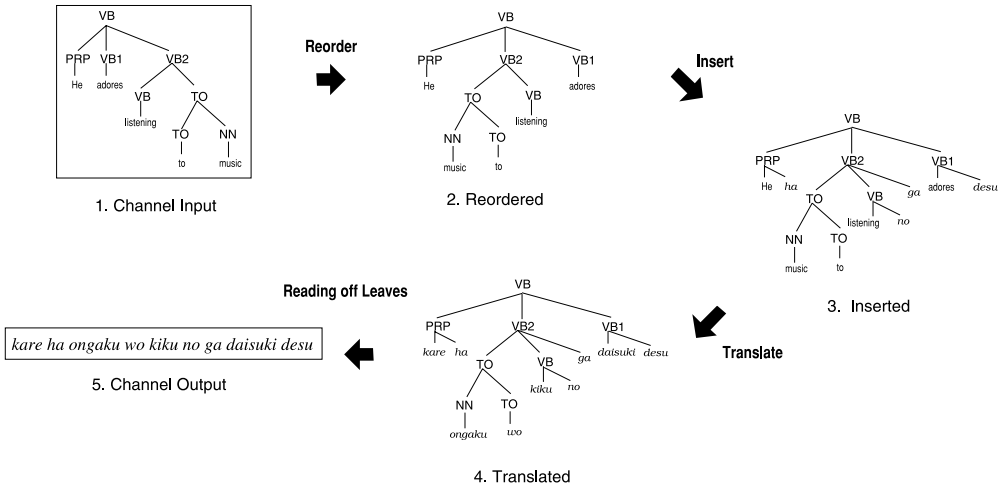


Figure 7
The translation model of Yamada and Knight (2001).

original order	reordering	P(reorder)
PRP VB1 VB2	PRP VB1 VB2	0.074
	PRP VB2 VB1	0.723
	VB1 PRP VB2	0.061
	VB1 VB2 PRP	0.037
	VB2 PRP VB1	0.083
	VB2 VB1 PRP	0.021
VB TO	VB TO	0.251
	TO VB	0.749
TO NN	TO NN	0.107
	NN TO	0.893
⋮	⋮	⋮

r–table

parent	TOP	VB	VB	VB	TO	TO	...
node	VB	VB	PRP	TO	TO	NN	...
P(None)	0.735	0.687	0.344	0.709	0.900	0.800	...
P(Left)	0.004	0.061	0.004	0.030	0.003	0.096	...
P(Right)	0.260	0.252	0.652	0.261	0.007	0.104	...

n–table

w	P(ins–w)
ha	0.219
ta	0.131
wo	0.099
no	0.094
ni	0.080
te	0.078
ga	0.062
⋮	⋮
desu	0.0007
⋮	⋮

E	adores		he		i		listening		music		to		...
J	*daisuki*	1.000	*kare*	0.952	*NULL*	0.471	*kiku*	0.333	*ongaku*	0.900	*ni*	0.216	...
			NULL	0.016	*watasi*	0.111	*kii*	0.333	*naru*	0.100	*NULL*	0.204	
			nani	0.005	*kare*	0.055	*mi*	0.333			*to*	0.133	
			da	0.003	*shi*	0.021					*no*	0.046	
			shi	0.003	*nani*	0.020					*wo*	0.038	
			⋮	⋮	⋮	⋮					⋮	⋮	

t–table

Figure 8
The parameter tables of Yamada and Knight (2001).

Second, at every node, a decision is made about inserting a Japanese function word. This is a three-way decision at each node—insert to the left, insert to the right, or do not insert—and it depends on the labels of the node and its parent.

Third, English leaf words are translated probabilistically into Japanese, independent of context.

Fourth, the internal nodes are removed, leaving only the Japanese string.

This model effectively provides a formula for P(Japanese string | English tree) in terms of individual parameters, and EM training seeks to maximize the product of these conditional probabilities across the whole tree/string corpus.

We now build a trainable xTs tree-to-string transducer that embodies the same P(Japanese string | English tree).

It is a four-state transducer. For the main state (and start state) q, meaning "translate this (sub)tree," we have three rules:

q x0 → i x0, r x0
q x0 → r x0, i x0
q x0 → r x0

State i means "produce a Japanese function word out of thin air." We include an i rule for every Japanese word in the vocabulary:

i x0 → "de"
i x0 → "kuruma"
i x0 → "wa"
. . .

State r means "re-order my children and then recurse." For internal nodes, we include a rule for every parent/child sequence and every permutation thereof:

r NN(x0:CD, x1:NN) → q x0, q x1
r NN(x0:CD, x1:NN) → q x1, q x0
. . .

The *rhs* sends the child subtrees back to state q for recursive processing. However, for English leaf nodes, we instead transition to a different state t, so as to prohibit any subsequent Japanese function word insertion:

r NN(x0:"car") → t x0
r CC(x0:"and") → t x0
. . .

State t means "translate this word," and we have a rule for every pair of co-occurring English and Japanese words:

t "car" → "kuruma"
t "car" → "wa"
t "car" → *e*
. . .

This follows Yamada and Knight (2001) in also allowing English words to disappear (the *rhs* of the last rule is an empty string).

Every rule in the xTs transducer has an associated weight and corresponds to exactly one of the model parameters.

The transducer just described, which we will subsequently call *simple*, is unfaithful in one respect so far: The insert-function-word decision is independent of context, whereas Yamada and Knight (2001) specifies it is conditioned on the node and parent labels. We modify the simple transducer into a new *exact* transducer by replacing the q

state with a set of states of the form q.parent, indicating the parent symbol of the current node being processed. The start state then becomes q.TOP, and the q rules are rewritten to specify the current node. Thus, every parent/child pair in the corpus gets its own set of insert-function-word rules:

q.TOP x0:VB → i x0, r x0
q.TOP x0:VB → r x0, i x0
q.TOP x0:VB → r x0
q.VB x0:NN → i x0, r x0
q.VB x0:NN → r x0, i x0
q.VB x0:NN → r x0
...

The r rules now need to send parent information when they recurse to the q.parent states:

r NN(x0:CD, x1:NN) → q.NN x0, q.NN x1
r NN(x0:CD, x1:NN) → q.NN x1, q.NN x0
...

The i and t rules stay the same.

This modification adds to our new transducer model all the contextual information specified in Yamada and Knight (2001). However, upon closer inspection one can see that the exact transducer is in fact *overspecified* in the reordering, or r rules. Yamada and Knight only conditions reordering on the child sequence, thus, for example, the reordering of JJ(JJ NN) is not distinct from the reordering of NN(JJ NN). As specified in ***Train*** a separate parameter is estimated for each rule in the transducer. We thus introduce rule *tying* to ensure the exact transducer is not misnamed. By designating a set of transducer rules as tied we indicate that a single count collection and parameter estimation is performed for the entire set during ***Train***. We denote tied rules by marking each rule in the same tied class with the symbol @ and a common integer. Thus the JJ(JJ NN) and NN(JJ NN) reordering rules described previously are modified as follows:

r JJ(x0:JJ, x1:NN) → q.JJ x0, q.JJ x1 @ 1
r JJ(x0:JJ, x1:NN) → q.JJ x1, q.JJ x0 @ 2
r NN(x0:JJ, x1:NN) → q.NN x0, q.NN x1 @ 1
r NN(x0:JJ, x1:NN) → q.NN x1, q.NN x0 @ 2

All reordering rules with the same input and output variable sequence are in the same tied class, and thus receive the same probability, independent of their parent symbols. We consider the four-state transducer initially specified as our *simple* model, and the modification that introduces parent-dependent q states and tied reordering rules as the *exact* model, since it is a precise xTs transducer formulation of the model of Yamada and Knight (2001).

As a means of providing empirical evidence of the utility of this approach, we built both the simple and exact transducers and trained them using the EM algorithm described in Section 7. We next compare the alignments and transition probabilities achieved by generic tree transducer operations with the model-specific implementation of Yamada and Knight (2001).

We obtained the corpus used as training data in Yamada and Knight (2001). This corpus is a set of 2,060 Japanese/English sentence pairs from a dictionary, preprocessed

Table 2
A comparison of the three transducer models used to simulate the model of Yamada and Knight (2001).

model	states	initial rules	rules after training	training time (hours)	% link match	% sent. match
simple	4	98,033	12,413	16.95	87.42	52.66
exact	28	98,513	12,689	17.42	96.58	81.46
perfect	29	186,649	24,492	53.19	99.85	99.47

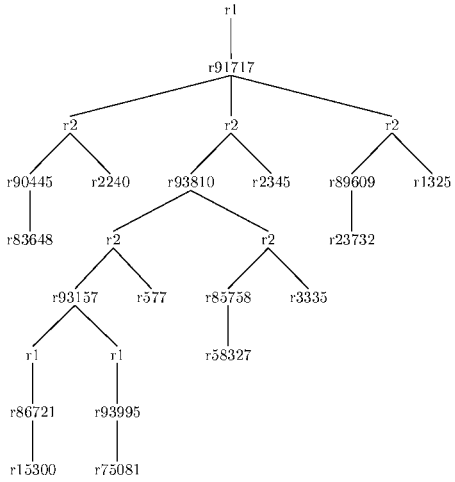
as described in Yamada and Knight. There are on average 6.9 English words per sentence and sentences range in size from 2 to 20 words. We built the simple and exact unweighted transducers described above; Table 2 summarizes their initial sizes. The exact model has 24 more states than the simple; this is due to the parent-dependent modification to q. The 480 additional rules are due to insertion rules dependent on parent and child information.

We then ran our training algorithm on the unweighted transducers and the training corpus. Because the derivation tree grammars produced by ***SDeriv*** can be large and time-intensive to compute, we calculated them once prior to training, saved them to disk, and then read them at each iteration of the training algorithm.[14] Following Yamada and Knight (2001), we chose a normalization partition (Z in ***Train***) such that we obtain the probabilities of all the rules given their complete left hand side,[15] and set the Dirichlet prior counts uniformly to 0. We ran 20 iterations of the EM algorithm using ***Train***. The time to construct derivation forests and run 20 iterations of EM for the various models is in Table 2. Note also the size of the transducers after training in Table 2; a rule is considered to be no longer in the transducer if it is estimated to have conditional probability 0.0001 or less.

Because we are trying to duplicate the training experiment of Yamada and Knight (2001), we wish to compare the word-to-word alignments discovered by that work to those discovered by ours. We recovered alignments from our trained transducers as follows: For each tree/string pair we obtained the most likely sequence of rules that derives the output string from an input tree, the *Viterbi derivation*. Figure 9 shows the Viterbi derivation tree and rules for an example sentence. By following the sequence of applied rules we can also determine which English words translate to which Japanese words, and thus construct the *Viterbi word alignment*. We obtained the full set of alignments induced in Yamada and Knight and compared them to the alignments learned from our transducers.

In Table 2 we report link match accuracy[16] as well as sentence match accuracy. The simple transducer is clearly only a rough approximation of the model of Yamada and Knight (2001). The exact model is much closer, but the low percentage of exact sentence matches is a concern. When comparing the parameter table values reported by Yamada and Knight with our rule weights we see that the two systems learned

14 In all models the size on disk in native Java binary object format was about 2.7 GB.
15 $Z_r(\vec{c}) \equiv \sum_{r'=(q_r,\lambda,e,f)\in R} c(r'), \forall r = (q_r, \lambda, g, h) \in R$.
16 As this model induces 1-to-1 word alignments, we report accuracy as the number of links matching those reported by Yamada and Knight (2001) as a percentage of the total number of links.



ID(r)	Rule(r)	p(r)
r1	q x0 → r x0	0.645
r91717	r VB(x0:PRP, x1:VB, x2:VB) → q x0, q x2, q x1	0.742
r2	q x0 → r x0, i x0	0.328
r90445	r PRP(x0:"he") → t x0	1.000
r83648	t "he" ↦ "kare"	0.953
r2240	i x0 → "ha"	0.201
r93810	r VB(x0:VB, x1:TO) → q x1, q x0	0.673
r93157	r TO(x0:TO, x1:NN) → q x1, q x0	0.327
r86721	r NN(x0:"music") → t x0	1.000
r15300	t "music" → "ongaku"	0.900
r93995	r TO(x0:"to") → t x0	1.000
r75081	t "to" → *e*	0.377
r577	i x0 → "wo"	0.115
r85758	r VB(x0:"listening") → t x0	1.000
r58327	t "listening" → "kiku"	0.333
r3335	i x0 → "no"	0.105
r2345	i x0 ↦ "ga"	0.067
r89609	r VB(x0:"adores") → t x0	1.000
r23732	t "adores" ↦ "daisuki"	1.000
r1325	i x0 → "desu"	0.001

Figure 9
A Viterbi derivation tree and the referenced rules.

different probability distributions in multiple instances. A sample of these parameter value differences can be seen in Figure 10.

In an effort to determine the reason for the discrepancy in weights between the parameter values learned in our exact transducer representation of Yamada and Knight (2001), we contacted the authors[17] and learned that, unreported in the paper, the original code contained a constraint that specifically bars an unaligned foreign word insertion immediately prior to a NULL English word translation. We incorporate this change to our model by simply modifying our transducer, rather than by changing our programming code. The new transducer, which we call *perfect*, is a modification of the exact transducer as follows.

We introduce an additional state s, denoting a translation taking place immediately after an unaligned foreign function word insertion. We then introduce the following additional rules.

For every rule that inserts a foreign function word, add an additional rule denoting an insertion immediately before a translation, and tie these rules together, for example:

q.VB x0:NN → i x0, r x0 @ 23
q.VB x0:NN → i x0, s x0 @ 23
q.VB x0:NN → r x0, i x0 @ 24
q.VB x0:NN → s x0, i x0 @ 24
. . .

To allow subsequent translation, "transition" rules for state s analogous to the transition rules described previously must also be added, for example:

s NN(x0:"car") → s x0
s CC(x0:"and") → s x0
. . .

17 We are grateful to Kenji Yamada for providing full parameter tables and Viterbi alignments from the original source.

rule	simple	exact	perfect	YK01
r VB(x0:PRP x1:VB x2:VB)→				
q.VB x0 q.VB x1 q.VB x2	.121	.056	.074	.074
q.VB x0 q.VB x2 q.VB x1	.682	.684	.723	.723
q.VB x1 q.VB x0 q.VB x2	.073	.080	.061	.061
q.VB x1 q.VB x2 q.VB x0	.030	.037	.037	.037
q.VB x2 q.VB x0 q.VB x1	.052	.121	.083	.083
q.VB x2 q.VB x1 q.VB x0	.043	.022	.021	.021
r VB(x0:VB x1:TO)→				
q.VB x0 q.VB x1	.253	.252	.251	.251
q.VB x1 q.VB x0	.747	.748	.749	.749
r TO(x0:TO x1:NN)→				
q.TO x0 q.TO x1	.188	.098	.107	.107
q.TO x1 q.TO x0	.812	.902	.893	.893

(a) r-table rules

rule	simple[a]	exact	perfect[b]	YK01
q.TOP x0:VB →				
r x0	.677	.756	.736	.736[c]
i x0 r x0	.025	.004	.004	.004
r x0 i x0	.298	.240	.260	.260
q.TOP x0:VB →				
r x0	.677	.675	.687	.687
i x0 r x0	.025	.057	.061	.061
r x0 i x0	.298	.267	.252	.252
q.VB x0:PRP →				
r x0	.677	.250	.344	.344
i x0 r x0	.025	.004	.004	.004
r x0 i x0	.298	.746	.652	.652
q.VB x0:TO →				
r x0	.677	.758	.709	.709
i x0 r x0	.025	.030	.030	.030
r x0 i x0	.298	.213	.261	.261
q.TO x0:TO →				
r x0	.677	.803	.900	.900
i x0 r x0	.025	.009	.003	.003
r x0 i x0	.298	.189	.097	.097[d]
q.TO x0:NN →				
r x0	.677	0	.800	.800
i x0 r x0	.025	1.0	.096	.096
r x0 i x0	.298	0	.104	.104
i x0 →				
"ha"	.195	.227	.219	.219
"ta"	.123	.125	.131	.131
"wo"	.115	.102	.099	.099
"no"	.103	.095	.094	.094
"ni"	.089	.081	.080	.080
"te"	.080	.076	.078	.078
"ga"	.064	.062	.062	.062
...	...	...	...	...
"desu"	.0006	.0008	.0007	.0007

[a] No specific parent-child information
[b] Two tied rules for each parameter value
[c] Incorrectly transcribed as 0.735 in (Yamada and Knight 2001)
[d] Incorrectly transcribed as 0.007 in (Yamada and Knight 2001)

(b) n-table rules

rule	simple	exact	perfect	YK01
t "adores"→				
"daisuki"	1.000	1.000	1.000	1.000
t "he"→				
"kare"	.951	.954	.952	.952
e	.020	.016	.016	.016
"nani"	.005	.004	.005	.005
"da"	.003	.003	.003	.003
"shi"	.00001	.003	.003	.003
t "i"→				
e	.529	.516	.471	.471
"watasi"	.105	.112	.111	.111
"kare"	.058	.048	.055	.055
"shi"	.011	.011	.020	.020[a]
"nani"	.013	.020	.020	.020

[a] Incorrectly transcribed as 0.021 in (Yamada and Knight 2001)

rule	simple	exact	perfect	YK01
t "listening"→				
"kiku"	.333	.333	.333	.333
"kii"	.333	.333	.333	.333
"mi"	.333	.333	.333	.333
t "music"→				
"ongaku"	.900	.900	.900	.900
"naru"	.100	.100	.100	.100
t "to"→				
"ni"	.193	.204	.216	.216
e	.301	.252	.204	.204
"to"	.105	.114	.113	.113[a]
"no"	.006	.031	.046	.046
"wo"	.007	.029	.038	.038

[a] Incorrectly transcribed as 0.133 in (Yamada and Knight 2001)

(c) t-table rules

Figure 10
Rule probabilities corresponding to the parameter tables of Yamada and Knight (2001).

Finally, for each *non-null* translation rule, add an identical translation rule starting with s instead of t, and tie these rules, for example:

t "car" → "kuruma" @ 54
t "car" → "wa" @ 55
t "car" → *e*
s "car" → "kuruma" @ 54
s "car" → "wa" @ 55
...

Note that there is no corresponding null translation rule from state s; this is in accordance with the insertion/NULL translation restriction.

As can be seen in Table 2 the Viterbi alignments learned from this "perfect" transducer are virtually identical to those reported in Yamada and Knight (2001). No

rule probability in the learned transducer differs from its corresponding parameter value in the original table by more than 0.000066. The 11 sentences with different alignments, which account for 0.53% of the corpus, were due to two derivations having the same probability; this was true in Yamada and Knight (2001) as well, and the choice between equal-scoring derivations is arbitrary. Transducer rules that correspond to the parameter tables presented in Figure 8 and a comparison of their learned weights over the three models with the weight learned in Yamada and Knight are in Figure 10. Note that the final perfect model matches the original parameter tables perfectly, indicating we can reproduce complicated models with our transducer formalism.

There are several benefits to this xTs formulation. First, it makes the model very clear, in the same way that Knight and Al-Onaizan (1998) and Kumar and Byrne (2003) elucidate other machine translation models in easily grasped FST terms. Second, the model can be trained with generic, off-the-shelf tools—versus the alternative of working out model-specific re-estimation formulae and implementing custom training software, whose debugging is a significant engineering challenge. Third, we can easily extend the model in interesting ways. For example, we can add rules for multi-level and lexical re-ordering:

r NP(x0:NP, PP(IN("of"), x1:NP)) → q x1, "no", q x0

We can eschew pre-processing that flattens trees prior to training, and instead incorporate flattening rules into the explicit model.

We can add rules for phrasal translations:

r NP(JJ("big"), NN("cars")) → "ooki", "kuruma"

This can include non-constituent phrasal translations:

r S(NP(PRO("there")), VP(VB("are")), x0:NP) → q x0, "ga", "arimasu"

Such non-constituent phrase pairs are commonly used in statistical machine translation (Och, Tillmann, and Ney 1999; Marcu and Wong 2002) and are vital to accuracy (Koehn, Och, and Marcu 2003). We can also eliminate many epsilon word-translation rules in favor of more syntactically-controlled ones, for example:

r NP(DT("the"), x0:NN) → q x0

Removing epsilons serves to reduce practical complexity in training and especially in decoding (Yamada and Knight 2002).

We can make many such changes without modifying the training procedure, as long as we stick to the tree automata.

The implementation of EM training we describe here is part of Tiburon, a generic weighted tree automata toolkit described in May and Knight (2006) and available at `http://www.isi.edu/licensed-sw/tiburon/`.

12. PCFG Modeling Experiment

In this section, we demonstrate another application of the xTs training algorithm. We show its generality by applying it to the standard task of training a probabilistic context-free grammar (PCFG) on string examples. Consider the following grammar:

S → NP VP
NP → DT N
NP → NP PP
PP → P NP
VP → V NP
VP → V NP PP

DT → the	N → the	V → the	P → the
DT → window	N → window	V → window	P → window
DT → father	N → father	V → father	P → father
DT → mother	N → mother	V → mother	P → mother
DT → saw	N → saw	V → saw	P → saw
DT → sees	N → sees	V → sees	P → sees
DT → of	N → of	V → of	P → of
DT → through	N → through	V → through	P → through

Also consider the following observed string data:

```
the father saw the window
the father saw the mother through the window
the mother sees the father of the mother
```

We would like to assign probabilities to the grammar rules such that the probability of the string data is maximized (Baker 1979; Lari and Young 1990). We can exploit the xTs training algorithm by pretending that each string was probabilistically transduced from a tree consisting of the single node $\emptyset$. All we require is to transform the grammar into an xTs transducer:

Start state: qs
qs x0 → qnp x0, qvp x0
qnp x0 $\rightarrow^{0.99}$ qdt x0, qn x0
qnp x0 $\rightarrow^{0.01}$ qnp x0, qpp x0
qpp x0 → qp x0, qnp x0
qvp x0 $\rightarrow^{0.99}$ qv x0, qnp x0
qvp x0 $\rightarrow^{0.01}$ qv x0, qnp x0, qpp x0

qdt $\emptyset$ → the	qn $\emptyset$ → the	qv $\emptyset$ → the	qp $\emptyset$ → the
qdt $\emptyset$ → window	qn $\emptyset$ → window	qv $\emptyset$ → window	qp $\emptyset$ → window
qdt $\emptyset$ → father	qn $\emptyset$ → father	qv $\emptyset$ → father	qp $\emptyset$ → father
qdt $\emptyset$ → mother	qn $\emptyset$ → mother	qv $\emptyset$ → mother	qp $\emptyset$ → mother
qdt $\emptyset$ → saw	qn $\emptyset$ → saw	qv $\emptyset$ → saw	qp $\emptyset$ → saw
qdt $\emptyset$ → sees	qn $\emptyset$ → sees	qv $\emptyset$ → sees	qp $\emptyset$ → sees
qdt $\emptyset$ → of	qn $\emptyset$ → of	qv $\emptyset$ → of	qp $\emptyset$ → of
qdt $\emptyset$ → through	qn $\emptyset$ → through	qv $\emptyset$ → through	qp $\emptyset$ → through

We also transform the observed string data into tree/string pairs:

$\emptyset \rightarrow$ the father saw the window
$\emptyset \rightarrow$ the father saw the mother through the window
$\emptyset \rightarrow$ the mother sees the father of the mother

After running the xTs training algorithm, we obtain maximum likelihood values for the rules. For example, after one iteration, we find the following values for rules that realize verbs:

qv $\emptyset \rightarrow^{0.11}$ of
qv $\emptyset \rightarrow^{0.11}$ through
qv $\emptyset \rightarrow^{0.22}$ sees
qv $\emptyset \rightarrow^{0.56}$ saw

After more iterations, values converge to:

qv $\emptyset \rightarrow^{0.33}$ sees
qv $\emptyset \rightarrow^{0.67}$ saw

Viterbi parses for the strings can also be obtained from the derivations forests computed by the ***SDeriv*** procedure. We note that our use of xTs training relies on copying.[18]

13. Related and Future Work

Concrete xLNT transducers are similar to (weighted) Synchronous TSG (STSG). STSG, like TSG, conflate tree labels with states, and so cannot reproduce all the relations in $\mathcal{L}(xLNT)$ without a subsequent relabeling step, although in some versions the root labels of the STSG rules' input and output trees are allowed to differ. Regular lookahead[19] for deleted input subtrees could be added explicitly to xT. Eisner (2003) briefly discusses training for STSG. For bounded trees, xTs can be represented as an FST (Bangalore and Riccardi 2002).

Our training algorithm is a generalization of forward–backward EM training for finite-state (string) transducers, which is in turn a generalization of the original forward–backward algorithm for Hidden Markov Models. Eisner (2002) describes string-based training under different semirings, and Carmel (Graehl 1997) implements FST string-to-string training. In our tree-based training algorithm, inside–outside weights replace forward–backward, and paths in trees replace positions in strings. Explicit construction and pruning of derivation trees saves time over many EM iterations, and could accelerate string-to-string training as well.

Yamada and Knight (2001) give a training algorithm for a specific tree-to-string machine translation model. Gildea (2003) introduces a variation of tree-to-tree mapping that allows for *cloning* (copying a subtree into an arbitrary position elsewhere), in order

18 Curiously, these rules can have "x0" in place of "$\emptyset$", because the training routine also supports deleting transducers. Such a transducer would transform *any* input tree to the output PCFG.
19 Tree patterns λ of arbitrary regular tree languages, as described in Engelfriet (1977).

to better robustly model the substantial tree transformations found in human language translation data.

Using a similar approach to ***Deriv***, exploiting the independence (except on state) of input-subtree/output-subtree mappings, we can build wRTG for the xT derivation trees matching an observed input tree (*forward application*), or matching an observed output tree (*backward application*).[20] For backward application through concrete transducers, each derivation tree implies a unique input tree, except where deletion occurs (the deleted input subtree could have been anything). For copying transducers, backward application requires wRTG intersection in order to ensure that only input-subtree hypotheses possible for *all* their derived output subtrees are allowed. For noncopying xTs transducers with complete tree patterns, backward application is just exhaustive context-free grammar parsing, generating a wRTG production from the left-hand-side of each xTs rule instance applied in parsing. Training and backward application algorithms for xTs can be extended in the usual way to parse given finite-state output lattices instead of single strings.[21]

14. Conclusion

We have motivated the use of tree transducers for natural language processing, and presented algorithms for training them. The tree-input/tree-output algorithm runs in $O(Gn^2)$ time and space, where G is a grammar constant, n is the total size of the tree pair, and the tree-input/string-output algorithm runs in $O(Gnm^3)$ time and space, where n is the size of the input tree and m is the size of the output string. Training works in both cases by building the derivation forest for each example, pruning it, and then (until convergence) collecting fractional counts for rules from those forests and normalizing. We have also presented an implementation and experimental results.

References

Aho, A. V. and J. D. Ullman. 1971. Translations of a context-free grammar. *Information and Control*, 19:439–475.

Alshawi, Hiyan, Srinivas Bangalore, and Shona Douglas. 2000. Learning dependency translation models as collections of finite state head transducers. *Computational Linguistics*, 26(1):45–60.

Baker, J. K. 1979. Trainable grammars for speech recognition. In *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pages 547–550, Boston, MA.

Bangalore, Srinivas and Owen Rambow. 2000. Exploiting a probabilistic hierarchical model for generation. In *International Conference on Computational Linguistics (COLING 2000)*, pages 42–48, Saarbrucken, Germany.

Bangalore, Srinivas and Giuseppe Riccardi. 2002. Stochastic finite-state models for spoken language machine translation. *Machine Translation*, 17(3):165–184.

Baum, L. E. and J. A. Eagon. 1967. An inequality with application to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bulletin of the American Mathematicians Society*, 73:360–363.

Charniak, Eugene. 2001. Immediate-head parsing for language models. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, pages 116–123, Tolouse, France.

Chelba, C. and F. Jelinek. 2000. Structured language modeling. *Computer Speech and Language*, 14(4):283–332.

Collins, Michael. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the ACL (jointly with the 8th Conference of the EACL)*, pages 16–23, Madrid, Spain.

20 In fact, forward and backward application can also be made to work on wRTG tree sets, with the result still being a wRTG of possible derivations, except in the case of forward application with copying.

21 Instead of pairs of string indices, spans are pairs of lattice states.

Comon, H., M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 1997. Tree automata techniques and applications. Available at `http://www.grappa.univ-lille3.fr/tata`. Release of 12 October 2007.

Corston-Oliver, Simon, Michael Gamon, Eric K. Ringger, and Robert Moore. 2002. An overview of Amalgam: A machine-learned generation module. In *Proceedings of the International Natural Language Generation Conference*, pages 33–40, New York.

Dempster, A. P., N. M. Laird, and D. B. Rubin. 1977. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38.

Doner, J. 1970. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451.

Eisner, Jason. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–8, Philadelphia, PA.

Eisner, Jason. 2003. Learning non-isomorphic tree mappings for machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (companion volume)*, pages 205–208, Sapporo, Japan.

Engelfriet, Joost. 1975. Bottom-up and top-down tree transformations—a comparison. *Mathematical Systems Theory*, 9(3):198–231.

Engelfriet, Joost. 1977. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303.

Engelfriet, Joost, Zoltán Fülöp, and Heiko Vogler. 2004. Bottom-up and top-down tree series transformations. *Journal of Automata, Languages and Combinatorics*, 7(1):11–70.

Fülöp, Zoltán and Heiko Vogler. 2004. Weighted tree transducers. *Journal of Automata, Languages and Combinatorics*, 9(1):31–54.

Gécseg, Ferenc and Magnus Steinby. 1984. *Tree Automata*. Akadémiai Kiadó, Budapest.

Gildea, Daniel. 2003. Loosely tree-based alignment for machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 80–87, Sapporo, Japan.

Graehl, Jonathan. 1997. Carmel finite-state toolkit. Available at `http://www.isi.edu/licensed-sw/carmel/`.

Graehl, Jonathan and Kevin Knight. 2004. Training tree transducers. In *HLT-NAACL 2004: Main Proceedings*, pages 105–112, Boston, MA.

Hopcroft, John and Jeffrey Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, London.

Joshi, Aravind and Yves Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer, Berlin, pages 69–124.

Klein, Dan and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan.

Knight, Kevin and Yaser Al-Onaizan. 1998. Translation with finite-state devices. In *Proceedings of the 3rd Conference of the Association for Machine Translation in the Americas on Machine Translation and the Information Soup (AMTA-98)*, pages 421–437, Berlin.

Knight, Kevin and Daniel Marcu. 2002. Summarization beyond sentence extraction: A probabilistic approach to sentence compression. *Artificial Intelligence*, 139(1):91–107.

Koehn, Phillip, Franz Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *HLT-NAACL 2003: Main Proceedings*, pages 127–133, Edmonton, Alberta, Canada.

Kuich, Werner. 1999. Tree transducers and formal tree series. *Acta Cybernetica*, 14:135–149.

Kumar, Shankar and William Byrne. 2003. A weighted finite state transducer implementation of the alignment template model for statistical machine translation. In *HLT-NAACL 2003: Main Proceedings*, pages 142–149, Edmonton, Alberta, Canada.

Langkilde, Irene. 2000. Forest-based statistical sentence generation. In *Proceedings of the 6th Applied Natural Language Processing Conference*, pages 170–177, Seattle, WA.

Langkilde, Irene and Kevin Knight. 1998. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the Conference of the Association for Computational Linguistics (COLING/ACL)*, pages 704–710, Montreal, Canada.

Lari, K. and S. J. Young. 1990. The estimation of stochastic context-free grammars using

the inside–outside algorithm. *Computer Speech and Language*, 4(1):35–56.
Marcu, Daniel and William Wong. 2002. A phrase-based, joint probability model for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 133–139, Philadelphia, PA.
May, Jonathan and Kevin Knight. 2006. Tiburon: A weighted tree automata toolkit. *Implementation and Application of Automata: 10th International Conference, CIAA 2005*, volume 4094 of *Lecture Notes in Computer Science*, pages 102–113, Taipei, Taiwan.
Nederhof, Mark-Jan and Giorgio Satta. 2002. Parsing non-recursive CFGs. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 112–119, Philadelphia, PA.
Och, Franz, Christoph Tillmann, and Hermann Ney. 1999. Improved alignment models for statistical machine translation. In *Proceedings of the Joint Conference of Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 20–28, College Park, MD.
Pang, Bo, Kevin Knight, and Daniel Marcu. 2003. Syntax-based alignment of multiple translations extracting paraphrases and generating new sentences. In *HLT-NAACL 2003: Main Proceedings*, pages 181–188, Edmonton, Alberta, Canada.
Rounds, William C. 1970. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287.
Schabes, Yves. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars*. Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.
Thatcher, James W. 1970. Generalized^2 sequential machine maps. *Journal of Computer and System Sciences*, 4:339–367.
Viterbi, Andrew. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269.
Wu, Dekai. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23(3):377–404.
Yamada, Kenji and Kevin Knight. 2001. A syntax-based statistical translation model. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, pages 523–530, Tolouse, France.
Yamada, Kenji and Kevin Knight. 2002. A decoder for syntax-based statistical MT. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 303–310, Philadelphia, PA.