In *Proc. of the Sixth International Conference on Intelligent Text Processing and Computational Linguistics (CICLing)*, Lecture Notes in Computer Science, ©Springer Verlag, 2005.

An Overview of Probabilistic Tree Transducers for Natural Language Processing

Kevin Knight and Jonathan Graehl

Information Sciences Institute (ISI) and Computer Science Department
University of Southern California
knight@isi.edu, graehl@isi.edu

Abstract. Probabilistic finite-state string transducers (FSTs) are extremely popular in natural language processing, due to powerful generic methods for applying, composing, and learning them. Unfortunately, FSTs are not a good fit for much of the current work on probabilistic modeling for machine translation, summarization, paraphrasing, and language modeling. These methods operate directly on trees, rather than strings. We show that tree acceptors and tree transducers subsume most of this work, and we discuss algorithms for realizing the same benefits found in probabilistic string transduction.

1 Strings

Many natural language problems have been successfully attacked with finite-state machines. It has been possible to break down very complex problems, both conceptually and literally, into cascades of simpler probabilistic **finite-state transducers** (FSTs). These transducers are bidirectional, and they can be trained on sample input/output string data. By adding a probabilistic **finite-state acceptor** (FSAs) language model to one end of the cascade, we can implement probabilistic **noisy-channel** models.[1] Figure 1 shows a cascade of FSAs and FSTs for the problem of transliterating names and technical terms across languages with different sounds and writing systems [1].

The finite-state framework is popular because it offers powerful, generic operations for statistical reasoning and learning. There are standard algorithms for:

- **intersection** of FSAs
- **forward application** of strings and FSAs through FSTs
- **backward application** of strings and FSAs through FSTs
- **composition** of FSTs
- **k-best** path extraction
- supervised and unsupervised **training** of FST transition probabilities from data

Even better, these generic operations are already implemented and packaged in research software toolkits such as AT&T's FSM toolkit [2], Xerox's finite-state calculus [3,4], van Noord's FSA Utilities [5], the RWTH toolkit [6], and USC/ISI's Carmel [7].

[1] In the noisy-channel framework, we look for the output string that maximizes P(output | input), which is equivalent (by Bayes Rule) to maximizing P(output) · P(input | output). The first term of the product is often captured by a probabilistic FSA, the second term by a probabilistic FST (or a cascade of them).

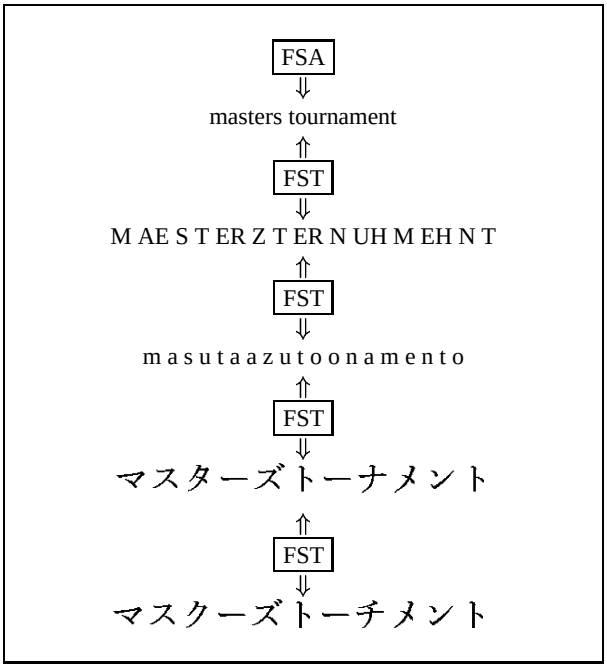


Fig. 1. A cascade of probabilistic finite-state machines for English/Japanese transliteration [1]. At the bottom is an optically-scanned Japanese katakana string. The finite-state machines allow us to compute the most probable English translation (in this case, “Masters Tournament”) by reasoning about how words and phonemes are transformed in the transliteration process.

Indeed, Knight & Al-Onaizan [8] describe how to use generic finite-state tools to implement the statistical machine translation models of [9]. Their scheme is shown in Figure 2, and [8] gives constructions for the transducers. Likewise, Kumar & Byrne [10] do this job for the phrase-based translation model of [11].

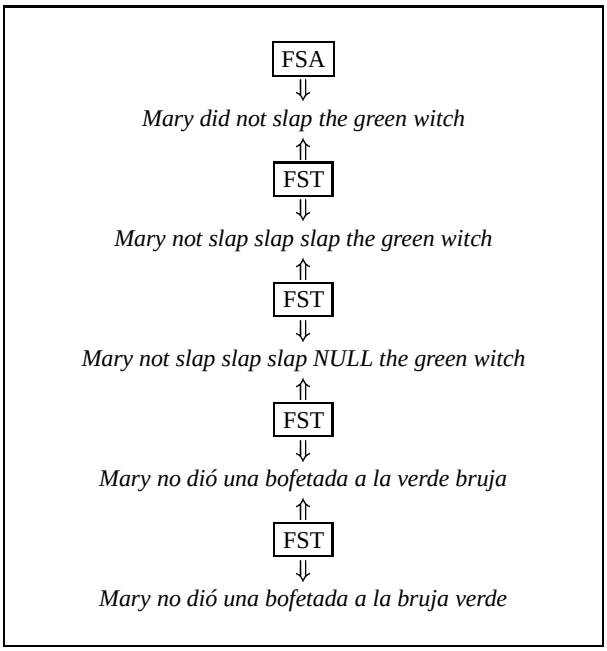


Fig. 2. The statistical machine translation model of [9] implemented with a cascade of standard finite-state transducers [8]. In this model, an observed Spanish sentence is "decoded" back into English through several layers of word substitution, insertion, deletion, and permutation.

2 Trees

Despite the attractive computational properties of finite-state tools, no one is particularly fond of their representational power for natural language. They are not adequate for long-distance reordering of symbols needed for machine translation, and they cannot implement the trees, graphs, and variable bindings that have proven useful for describing natural language processes. So over the past several years, researchers have been developing probabilistic **tree-based** models for

- machine translation (e.g., [13,14,12,15,16,17])
- summarization (e.g., [18])
- paraphrasing (e.g., [19])

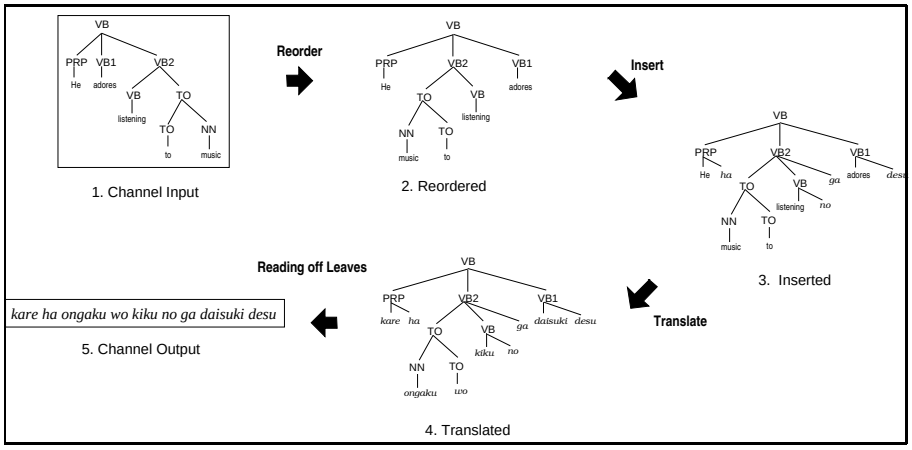


Fig. 3. A syntax-based machine translation model [12].

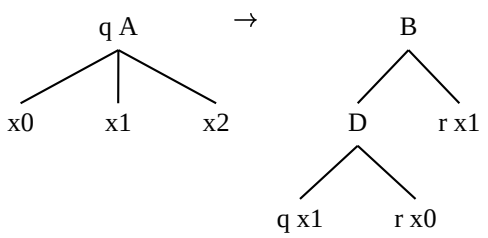
- natural language generation (e.g., [20,21,22])
- question answering (e.g., [23]), and
- language modeling (e.g., [24])

An example of such a tree-based model is shown in Figure 3. Unlike in the previous figures, the probabilistic decisions here are sensitive to syntactic structure.

We have found that most of the current syntax-based NLP models can be neatly captured by probabilistic **tree automata**. Rounds [25] and Thatcher [26] independently introduced top-down **tree transducers** as a generalization of FSTs. Rounds was motivated by problems in 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 ... results should clarify the nature of syntax-directed translations and transformational grammars."[25]

Top-down tree transducers [25,26] are often called **R-transducers**. (R stands for "**R**oot-to-frontier"). An R-transducer walks down an input tree, transforming it and processing branches independently in parallel. It consists of a set of simple transformation **productions** (or **rules**) like this:



This production means:

> *When in state q, facing an input tree with root symbol A and three children* **about which we know nothing**, *replace it with a subtree rooted at B with two children, the left child being a D with two children of its own. To compute D's children, recursively process A's middle child (with state q) and A's left child (with state r). To compute B's right child, recursively process A's middle child (with state r).*

Here, "recursively" means to take the subtree rooted at A's child and look for productions that might in turn match it. R productions are limited to left-hand sides that match only on <state, symbol> pairs, while the right-hand side may have any height. Productions are often written textually, e.g.:

q A(x0, x1, x2) → B(D(q x1, r x0), r x1)

If probabilities are attached to individual productions, then the tree transducer becomes a probabilistic tree transducer.

Figure 4 shows a probabilistic R-transducer based on a machine translation model similar to [12]. This model probabilistically reorders siblings (conditioned on the parent's and siblings' syntactic categories), inserts Japanese function words, and translates English words into Japanese, all in one top-down pass. It defines a conditional probability distribution P(J | E) over all English and Japanese tree pairs. Figure 5 shows an **R-derivation** from one English input tree (with start state q.s) to one possible Japanese output.

Figure 6 shows the corresponding **derivation tree**, which is a record of the rules used in the derivation shown in Figure 4. There are several things worth noting:

- Some rules (like Rule 4) may appear multiple times in the same derivation tree.
- Both the input and output trees can be recovered from the derivation tree, though this takes some work.
- There are usually many derivation trees connecting the same input tree (in this case, English) with the same output tree (in this case, Japanese), i.e., multiple ways of "getting from here to there."

The fact that we can use tree transducers to capture tree-based models proposed in the natural language literature is significant, because extensive work goes into each of these models, for both training and decoding. These are one-off solutions that take a long time to build, and they are difficult to modify. By casting translation models as R-transducers, Graehl & Knight [27] discuss how to add productions for linguistic phenomena not captured well by previous syntax-based translation models, including non-constituent phrase translation, lexicalized reordering, and long-distance wh-movement (Figure 7). Having generic, implemented R operations would allow researchers to focus on modeling rather than coding and specialized algorithm development.

The purpose of this paper is to explore whether the benefits of finite-state string automata can be enjoyed when we work with trees—i.e., what are the implications of trying to reason with tree models like Figure 4, in transducer cascades like those shown

```
/* translate */
```

1. q.s S(x0, x1) $\rightarrow^{0.9}$ S(q.np x0, q.vp x1)
2. q.s S(x0, x1) $\rightarrow^{0.1}$ S(q.vp x1, q.np x0)
3. q.np x $\rightarrow^{0.1}$ r.np x
4. q.np x $\rightarrow^{0.8}$ NP(r.np x, i x)
5. q.np x $\rightarrow^{0.1}$ NP(i x, r.np x)
6. q.pro PRO(x0) $\rightarrow^{1.0}$ PRO(q x0)
7. q.nn NN(x0) $\rightarrow^{1.0}$ NN(q x0)
8. q.vp x $\rightarrow^{0.8}$ r.vp x
9. q.vp x $\rightarrow^{0.1}$ S(r.vp x, i x)
10. q.vp x $\rightarrow^{0.1}$ S(i x, r.vp x)
11. q.vbz x $\rightarrow^{0.4}$ r.vbz x
12. q.vbz x $\rightarrow^{0.5}$ VP(r.vbz x, i x)
13. q.vbz x $\rightarrow^{0.1}$ VP(i x, r.vbz x)
14. q.sbar x $\rightarrow^{0.3}$ r.sbar x
15. q.sbar x $\rightarrow^{0.6}$ SBAR(r.sbar x, i x)
16. q.sbar x $\rightarrow^{0.1}$ SBAR(i x, r.sbar x)
17. q.vbg VBG(x0) $\rightarrow^{1.0}$ VP(VB(q x0))
18. q.pp PP(x0, x1) $\rightarrow^{1.0}$ NP(q.np x1, q.p x0)
19. q.p P(x0) $\rightarrow^{1.0}$ PN(q x0)
20. q he $\rightarrow^{1.0}$ kare
21. q enjoys $\rightarrow^{0.1}$ daisuki
22. q listening $\rightarrow^{0.2}$ kiku
23. q to $\rightarrow^{0.1}$ o
24. q to $\rightarrow^{0.7}$ ni
25. q music $\rightarrow^{0.8}$ ongaku
26. r.vp VP(x0, x1) $\rightarrow^{0.9}$ S(q.vbz x0, q.np x1)
27. r.vp VP(x0, x1) $\rightarrow^{0.1}$ S(q.np x1, q.vbz x0)
28. r.sbar SBAR(x0, x1) $\rightarrow^{0.1}$ S(q.vbg x0, q.pp x1)
29. r.sbar SBAR(x0, x1) $\rightarrow^{0.9}$ S(q.pp x1, q.vbg x0)
30. r.np NP(x0) $\rightarrow^{0.1}$ q.pro x0
31. r.np NP(x0) $\rightarrow^{0.8}$ q.nn x0
32. r.np NP(x0) $\rightarrow^{0.1}$ q.sbar x0
33. r.vbz VBZ(x0) $\rightarrow^{0.7}$ VB(q x0)

```
/* insert */
```

34. i NP(x0) $\rightarrow^{0.3}$ PN(wa)
35. i NP(x0) $\rightarrow^{0.3}$ PN(ga)
36. i NP(x0) $\rightarrow^{0.2}$ PN(o)
37. i NP(x0) $\rightarrow^{0.1}$ PN(ni)
38. i SBAR(x0, x1) $\rightarrow^{0.7}$ PS(no)
39. i VBZ(x0) $\rightarrow^{0.2}$ PV(desu)

Fig. 4. A probabilistic tree transducer.

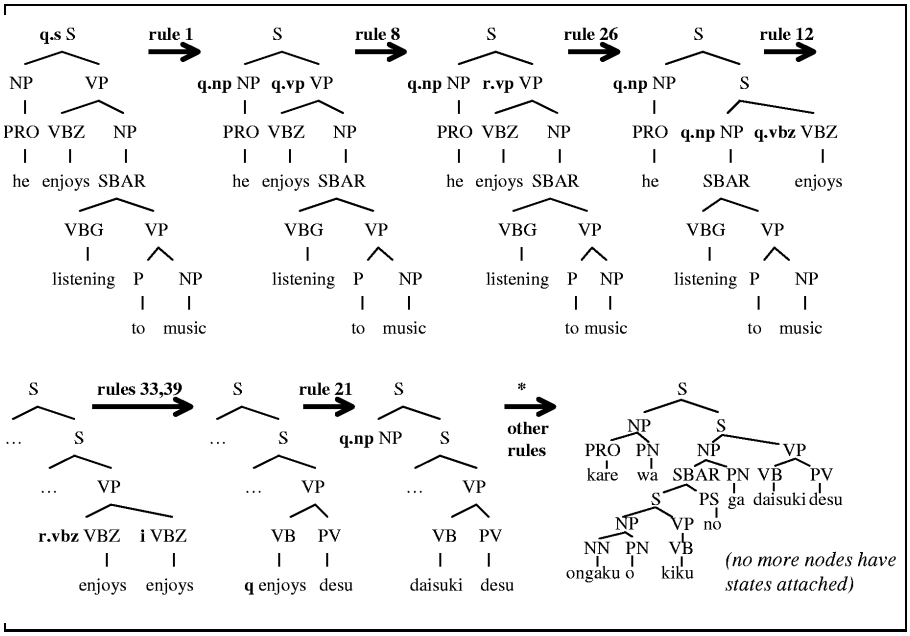


Fig. 5. An English (input) tree being transformed into a Japanese (output) tree by the tree transducer in the previous figure. Because the transducer is non-deterministic, many other output trees are also possible.

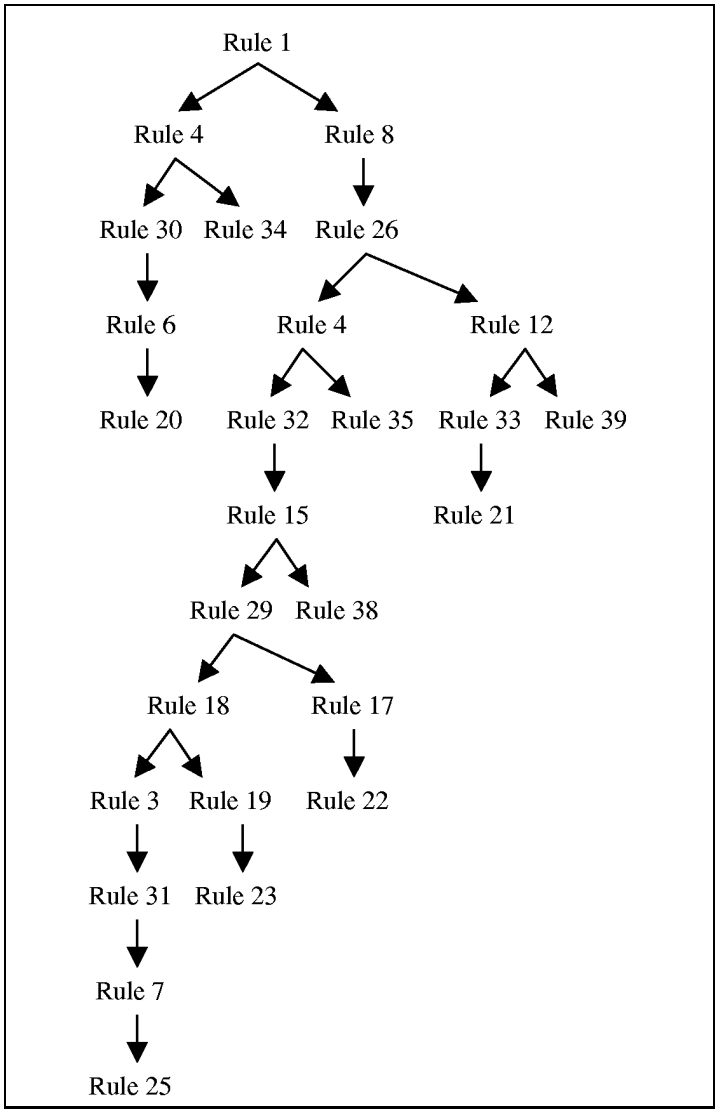


Fig. 6. A derivation tree, or record of transducer rules used to transform a particular input tree into a particular output tree. With careful work, the input and output trees can be recovered from the derivation tree.

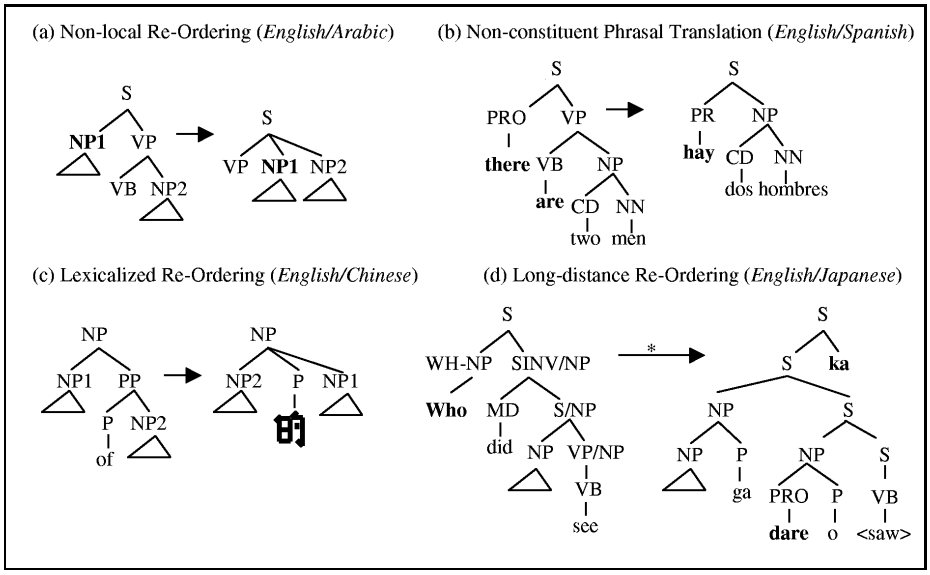


Fig. 7. Tree transducer operations for capturing translation patterns.

in Figures 1 and 2? We survey the extensive literature on tree automata from the focused viewpoint of large-scale statistical modeling/training/decoding for natural language.

In this paper, we give many examples of tree transducers and discuss their consequences, but we will not give formal mathematical definitions and proofs (e.g., "... a tree transducer is a 7-tuple ... "). For readers wishing to dig further into the tree automata literature, we highly recommend the excellent surveys of Gécseg & Steinby [28] and Comon et al [29].

Because tree-automata training is already discussed in [27,30], this paper concentrates on *intersection*, *composition*, forward and backward *application*, and *search*.

3 Intersection

In finite-state string systems, FSAs represent sets of (weighted) strings, such as English sentences. FSAs can capture only regular languages (which we refer to here as **regular string languages** (RSLs) to distinguish them from tree languages). Other ways to capture RSLs include regular grammars and regular expressions. The typical operation on FSAs in a noisy-channel cascade is the intersection of channel-produced candidate strings with an FSA language model. It is very convenient to use the same English language model across different applications that need to map input into English (e.g., translation).

When working with trees, an analog of the RSL is the **regular tree language**, which is a (possibly infinite) set of trees. A probabilistic RTL assigns a P(tree) to every tree. An RTL is usually specified by a **regular tree grammar** (RTG), an example of which is shown in Figure 8. Alternatively, there exists an RTL recognition device [31] that

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 accepted trees:

S(NP(DET(the), N(sons)),
VP(VB(run)))

(*prob = 0.3*)

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

(*prob = 0.036*)

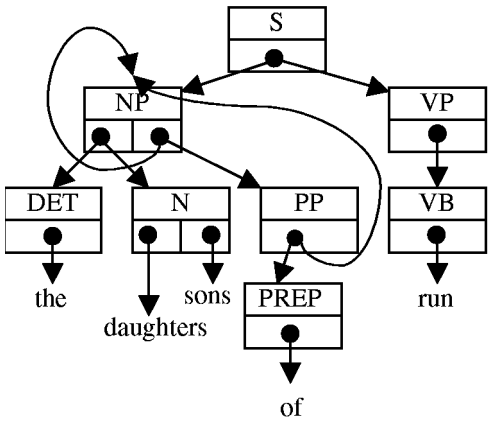


Fig. 8. A sample probabilistic regular tree grammar (RTG). This RTG accepts/generates an infinite number of trees, whose probabilities sum to one.

crawls over an input, R-style, to accept or reject it. This device is the analog of an FSA. In contrast with FSAs, non-deterministic top-down RTL recognizers are strictly more powerful than deterministic ones.

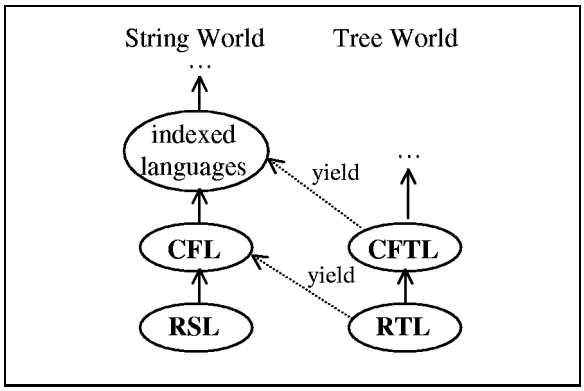


Fig. 9. String language classes and tree language classes. These classes include regular string languages (RSL), regular tree languages (RTL), context-free (string) languages (CFL), context-free tree languages (CFTL), and indexed (string) languages.

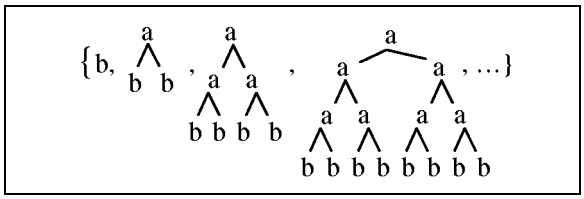
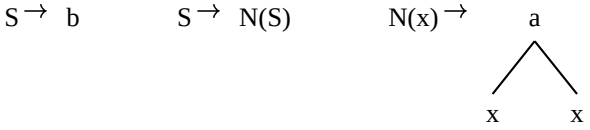


Fig. 10. A tree language that is not a regular tree language (RTL).

There exists a standard hierarchy of tree language classes, summarized in Figure 9. Some sets of trees are not RTL. An example is shown in Figure 10—left and right subtrees are always of equal depth, but there is no way to guarantee this for arbitrary depth, as a regular tree grammar must produce the subtrees independently.

The connection between string and tree languages (Figure 9) was observed by Rounds [25]—if we collect the leaf sequences (**yields**) of the trees in an RTL, we are guaranteed to get a string set that is a **context-free language** (CFL). Moreover, every CFL is the yield language of some RTL, and the derivation trees of a CFG form an RTL. However, an RTL might not be the set of derivation trees from any CFG (unless re-labeling is done).

The tree language in Figure 10 is not RTL, but it is a context-free tree language (CFTL). CFTG allows the left-hand side of a production to have variables—the CFTG for Figure 10 is:



The yield language of this non-RTL is $\{b^{2^n} : n \geq 0\}$, which is not a CFL. We do not pursue CFTG further.

	Strings		Trees
	RSL (FSA, regexp)	CFL (PDA, CFG)	RTL (RTA, RTG)
closed under union	YES ([32] p. 59)	YES ([32] p. 131)	YES ([28] p. 72)
closed under intersection	YES ([32] p. 59)	NO ([32] p. 134)	YES ([28] p. 72)
closed under complement	YES ([32] p. 59)	NO ([32] p. 135)	YES ([28] p. 73)
membership testing	O(n) ([32] p. 281)	$O(n^3)$ ([32] p. 140)	O(n) ([28] p. 110)
emptiness decidable	YES ([32] p. 64)	YES ([32] p. 137)	YES ([28] p. 110)
equality decidable	YES ([32] p. 64)	NO ([32] p. 203)	YES ([28] p. 110)

Fig. 11. Properties of string and tree language classes.

How do the properties of tree languages stack up against string languages? Figure 11 summarizes. The good news is that RTLs have all of the good properties of RSLs. In particular, RTLs are closed under intersection, so there exists an algorithm to intersect two RTGs, which allows probabilistic RTG language models to be intersected with possibly infi nite sets of candidate trees coming through the noisy channel. RTGs are therefore a suitable substitute for FSAs in such probabilistic cascades.

It is also interesting to look at how RTGs can capture probabilistic syntax models proposed in the natural language literature. RTGs easily implement **probabilistic CFG** (PCFG) models initially proposed for tasks like natural language parsing[2] and language modeling.[3] These models include parameters like P(NP → PRO | NP). However, PCFG models do not perform very well on parsing tasks. One useful extension is that of Johnson [33], which further conditions a node's expansion on the node's parent, e.g., P(NP → PRO | NP, parent=S). This captures phenomena such as pronouns appearing in subject position more frequently than in object position, and it leads to better performance. Normally, Johnson's extension is implemented as a straight PCFG with new node types, e.g., P($NP^S \rightarrow PRO^{NP} \mid NP^S$). This is unfortunate, since we are interested in getting

[2] The parsing task is frequently stated as selecting from among the parse trees of a input sentence the one with highest P(tree).

[3] The language modeling task is to assign a high P(tree) only to grammatical, sensible English trees.

out trees with labels like NP and PRO, not NP^S and PRO^S. An RTG elegantly captures the same phenomena without changing the node labels:

{qstart → S(qnp.s, qvp.s)
 qvp.s → VP(qv.vp, qnp.vp)
 qnp.s $\rightarrow^{0.3}$ NP(qpro.np)
 qnp.s $\rightarrow^{0.7}$ NP(qdet.np, qn.np)
 qnp.vp $\rightarrow^{0.05}$ NP(qpro.np)
 qnp.vp $\rightarrow^{0.95}$ NP(qdet.np, qn.np)
 qpro.np $\rightarrow^{0.5}$ PRO(he)
 qpro.np $\rightarrow^{0.5}$ PRO(him)
 ... }

Here we can contrast the 0.3 value (pronouns in subject position) with the 0.05 value (pronouns in object position).

A drawback to the above model is that the decision to generate "he" or "him" is not conditioned on the subject/object position in the sentence. This can also be addressed in an RTG—the more context is useful, the more states can be introduced.

High-performing models of parsing and language modeling [34,24] are actually lexicalized, with probabilistic dependencies between head words and their modifiers, e.g., P(S=sang → NP=boy VP=sang | S=sang). Because they are trained on sparse data, these models include extensive backoff schemes. RTGs can implement lexicalized models with further use of states, though it is open whether good backoff schemes can be encoded by compact RTGs.

Another interesting feature of high-performing syntax models is that they use a **markov** grammar rather than a finite list of rewrite rules. This allows them to recognize or build an infinite number of parent/children combinations, e.g., S → NP VP PP*. Formal machinery for this was introduced by Thatcher [35], in his **extended context-free grammars** (ECFG). ECFGs allow regular expressions on the right-hand side of productions, and they maintain good properties of CFGs. Any string set represented by an ECFG can also be captured by a CFG, though (of course) the derivation tree set may not be captured.

While standard PCFG grammars are parameterized like this

P_{rule}(S → ADV NP VP PP PP | S),

markov grammars [34,24] are parameterized somewhat like this:

P_{head}(VP | S) ·
P_{left}(NP | VP, S) ·
P_{left}(ADV | VP, S) ·
P_{left}(STOP | VP, S) ·
P_{right}(PP | VP, S) ·
P_{right}(PP | VP, S) ·
P_{right}(STOP | VP, S)

We can capture this exactly by replacing the right-hand side of each ECFG production with a probabilistic FSA, with transitions weighted by the parameter values above, re-

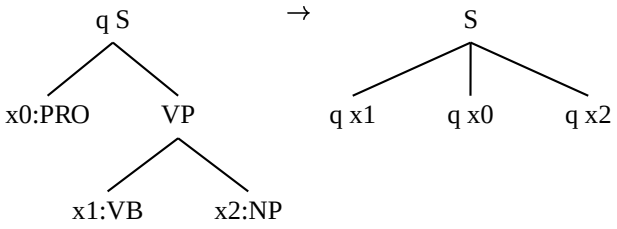
sulting in a **probabilistic ECFG**. The extended CFG notion can then be carried over to make an **extended probabilistic RTG**, which is a good starting point for capturing state-of-the-art syntax models.

4 Tree Transducer Hierarchy

Before turning to tree transducer composition, we first cover the rich class hierarchy of tree transducers. Some of these classes are shown in Figure 12. (This figure was synthesized from many sources in the tree automata literature). Standard acronyms are made of these letters:

- **R**: Top-down transducer, introduced before.
- **F**: Bottom-up transducer ("**F**rontier-to-root"), with similar rules, but transforming the leaves of the input tree first, and working its way up.
- **L**: **Linear** transducer, which prohibits copying subtrees. Rule 4 in Figure 4 is example of a copying production, so this whole transducer is R but not RL.
- **N**: **Non-deleting** transducer, which requires that every left-hand-side variable also appear on the right-hand side. A deleting R-transducer can simply delete a subtree (without inspecting it). The transducer in Figure 4 is the deleting kind, because of rules 34-39. It would also be deleting if it included a rule for dropping English determiners, e.g., q NP(x0, x1) → q x1.
- **D**: **Deterministic** transducer, with a maximum of one production per <state, symbol> pair.
- **T**: **Total** transducer, with a minimum of one production per <state, symbol> pair.
- **PDTT**: Push-down tree transducer, the transducer analog of CFTG [36].
- subscript$_R$: **Regular-lookahead** transducer, which can check to see if an input subtree is tree-regular, i.e., whether it belongs to a specified RTL. Productions only fire when their lookahead conditions are met.

We also introduce the prefix **x** for transducers with **extended left-hand-side** productions[4] that can look finitely deeply into the input, performing tests or grabbing deeper material. xR-transducers are easier to write; for example, we can have a production like



which moves a subject pronoun in between the verb and direct object, as happens in machine translation between certain language pairs.

[4] *Extended* left-hand-side productions are not related to *extended* context-free grammars—the notions unfortunately overwork the same adjective.

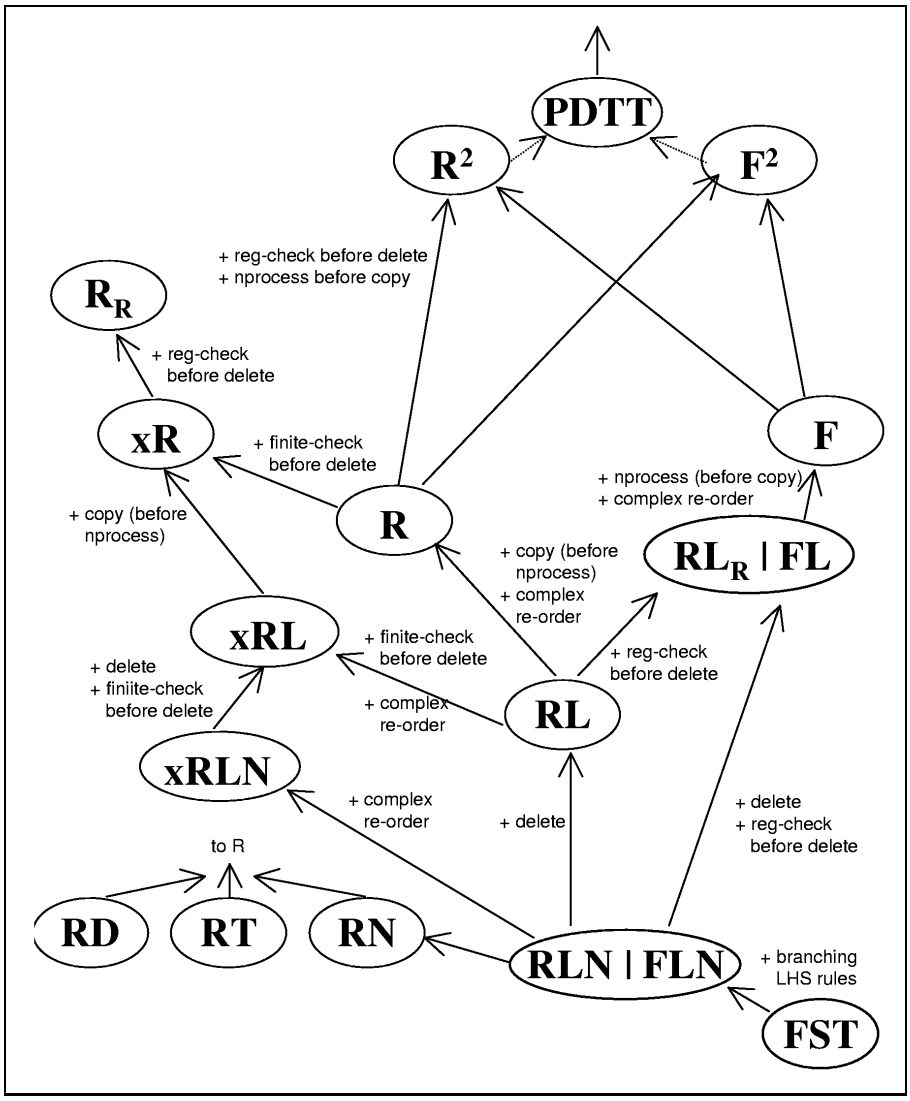


Fig. 12. Some tree transducer classes. Upward arrows indicate increasing levels of transduction power (higher classes can capture more kinds of transductions). Arrows are labeled with a rough description of the power that is added by moving to the higher class. R^2 represents the transduction capability of two R-transducers chained together.

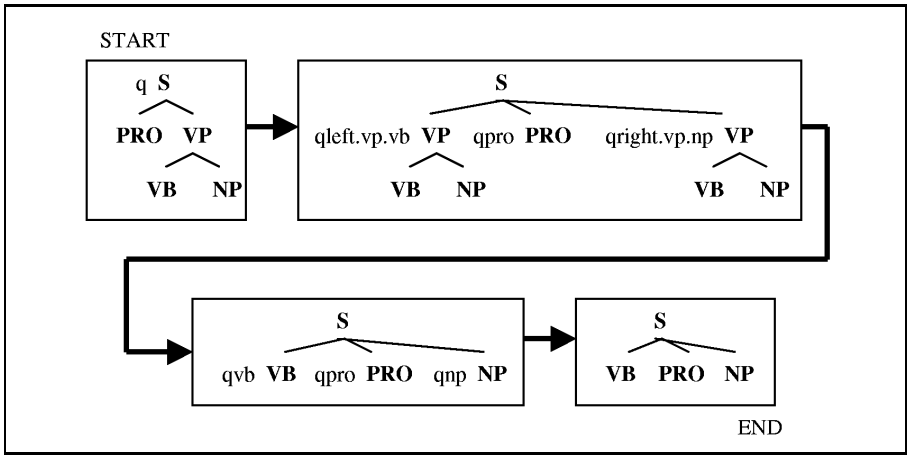


Fig. 13. Complex re-ordering with an R-transducer.

Actually, this case can be captured with R, through the use of states and copying, as demonstrated with these productions:

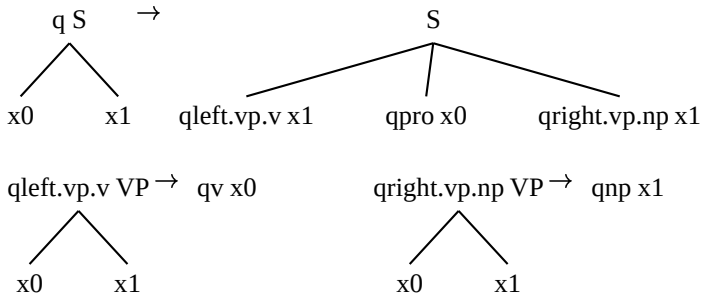


Figure 13 shows how the transformation is carried out. This kind of rule programming is cumbersome, however, and the single xR production above is preferred.

Because of their good fi t with natural language applications, extended left-hand-side productions were briefly touched on already in Section 4 of Rounds' paper [25], though not defi ned. xR cannot quite be simulated by R, because xR has the power to check the root label of a subtree before deleting that subtree, e.g.:

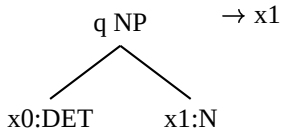


Figure 12 shows that R and F are incomparable. R can delete subtrees without looking at them, while F cannot. F can non-deterministically modify a tree bottom-up, then copy the result, while R has to make the copies first, before modifying them. Since they are modified independently and in parallel, R cannot guarantee that copies are modified in the same way.

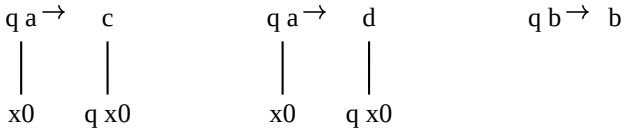
We can also see that non-copying RL and FL transducers have reduced power. Prohibiting both deletion and copying removes the differences between R and F, so that RLN = FLN. Regular lookahead adds power to R [37].

The string-processing FST appears in Figure 12 at the lower right. If we write strings vertically instead of horizontally, then the FST is just an RLN transducer with its left-hand-side productions limited to one child each. Standard FSTs are non-deleting. FSTs can also have transitions with both epsilon input and epsilon output. We show the tree analog of this in Rule 3 of Figure 4; we note that proofs in the tree automata literature do not generally work through epsilon cases.

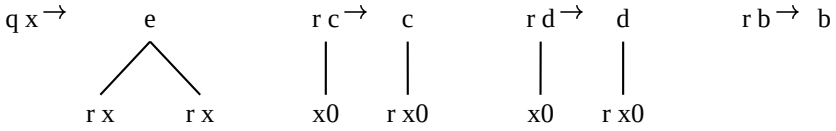
5 Composition

Now we turn to composition. For the string case, FSTs are closed under composition, which means that a long FST cascade can always be composed offline into a single FST before use.

By contrast, R is not closed under composition, as demonstrated by Rounds [25]. The proof is as follows. We set up one R-transducer to non-deterministically modify a **monadic** (non-branching) input tree composed of some number of a's followed by a b; this transducer changes some of the a's to c's and other a's to d's:

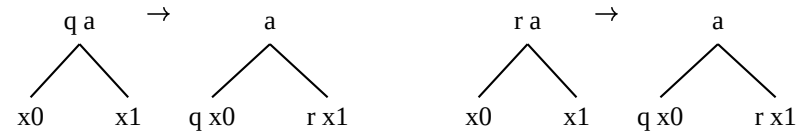


We then set up another R-transducer that simply places two copies of its input under a new root symbol e:



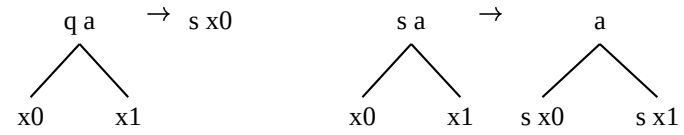
Each of these two transformations is R, but no single R-transducer can do both jobs at once, i.e., non-deterministically modify the input and make a copy of the result. The copy has to be made first, in which case both copies would be processed independently and the branches would diverge. The fact that R is not closed under composition is denoted in Figure 12 by the fact that R^2 (the class of transformations that can be captured with a sequence of two R-transducers) properly contains R.

RL is also not closed under composition [38]. To show this, we set up one RL transducer to pass along its input tree untouched, but only in case the right branch passes a certain test (otherwise it rejects the whole input):



q b → b q f → f r f → f

We set up another RL transducer to simply delete the right branch of its input:



s b → b s f → f

No transducer can do both jobs at once, because no R or RL transducer can check a subtree first, then delete it.

Better news is that RLN (= FLN) is closed under composition, and it is a natural class for many of the probabilistic tree transformations we find in the natural language literature. If we are fond of deleting, then FL is also closed under composition. RTD is also closed, though total deterministic transformations are not of much use in statistical modeling. There are also various other useful compositions and decompositions [38], such as the fact that RL composed with RLN is always RL. These decompositions can help us analyze transducer cascades with mixed transducer types.

We note in passing that none of the tree transducers are closed under intersection, but string FSTs fare no better in this regard (Figure 14). We also note that all the listed transducers are as efficiently trainable as FSTs, given sample tree pairs [27].

6 Forward and Backward Application

Next we turn to transducer application, e.g., what happens if we send an input tree through an R-transducer? What form will the output take?

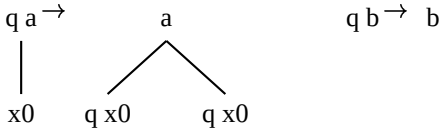
First, the string case: if we feed an input FSA to an FST, the output (**image**) is always an FSA. Likewise, if we present an observed output FSA, we can ask the FST what inputs might have produced any of those strings (**inverse image**), and this is also always an FSA. In practice, the situation is even simpler—we turn input FSAs into identity FSTs, then use FST composition in place of application.

For trees, we can also create an identity tree transducer out of any RTG, but general composition may not be possible, as described in the previous section. However, straight application is a different story—known results are summarized in the remainder of Figure 14.

The bad news is that R does not preserve regularity. For example, consider the transducer

	Strings	Trees				
	FST	R	RL	RLN (= FLN)	F	FL
closed under composition	YES	NO [28] p. 162	NO [28] p. 158	YES from FL	NO [28], p. 162	YES [28], p. 158
closed under intersection	NO	NO	NO	NO	NO	NO
efficiently trainable	YES [39]	YES [27]	YES from R	YES from R	YES	YES
image of tree is:	RSL [28], p. 52	RTL	RTL [28], p. 175	RTL from R	not RTL	RTL [28], p. 174
inverse image of tree is:	RSL [28], p. 52	RTL [28], p. 162	RTL from R	RTL from R	RTL [28], p. 162	RTL from F
image of RTL is:	RSL [28], p. 52	not RTL [28], p. 179	RTL [28], p. 175	RTL from R	not RTL [28], p. 179	RTL [28], p. 174
inverse image of RTL is:	RSL [28], p. 52	RTL [28], p. 162	RTL from R	RTL from R	RTL [28], p. 162	RTL from F

Fig. 14. Properties of string and tree transducers.



If we supply this transducer with an input RTL consisting of all the monadic trees a^*b, then we get the non-RTL output tree set shown back in Figure 10. The same holds for F. In fact, the situation is worse for F, because even a single input tree can produce non-RTL output.

The good news is that sending a single tree (or by extension a finite RTL) through R guarantees RTL output. Moreover, the inverse image of an RTL through any of R, RL, RLN, F, or FL is still RTL. This is relevant to our noisy-channel cascade, in which the observed tree is passed backwards through the transducer cascade until it reaches the language model. This will work even for a cascade of R-transducers, despite the fact that it is not possible to compose those transducers off-line in the general case.

Getting a correct inverse image is tricky when the transducer is a deleting one (e.g., R or RL). A fully-connected RTL, capable of generating any tree in the language, must be inserted at every delete-point going backwards.[5] Copying also complicates inverse imaging, but it can be handled with RTL intersection.

[5] In practice, we may think twice about using delete-before-check to delete an English determiner, e.g., q NP(x0 x1) → x1. In decoding from a foreign language without determiners, the inverse image will contain every imaginable English subtree vying to fill that determiner slot.

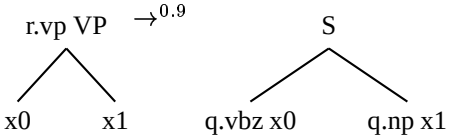
	Tree-to-String		
	Rs	RLs	RLNs
closed under composition	n.a.	n.a.	n.a.
image of tree is:	CFL	CFL	CFL
inverse image of string is:	RTL	RTL	RTL
image of RTL is:	not CFL	CFL	CFL
inverse image of RSL is:	RTL	RTL	RTL

Fig. 15. Properties of tree-to-string transducers. These properties derive from the imaging behaviors of R, RL, and RLN, respectively.

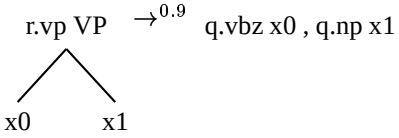
7 Tree-to-String Transducers

Figure 15 shows properties of **tree-to-string** transducers, denoted with suffi x **s**. Tree-to-string transducers were introduced by Aho and Ullman [40] for compilers. Several recent machine translation models are also tree-to-string [12,30].

We can easily turn the R-transducer in Figure 4 into an Rs-transducer by removing the internal structure on the right-hand-sides of productions, yielding a comma-separated list of leaves. For example, instead of rule 26:



we use:



The sample input English tree of Figure 5 would then probabilistically transduce into a flat Japanese string, e.g.:

kare , wa , ongaku , o , kiku , no , ga , daisuki , desu

In contrast with R, the empty string is allowed in the right-hand side of Rs productions.

The inverse image of a string through Rs is guaranteed to be an RTL, as is the inverse image of a whole FSA. This result is similar to results that permit parsing of lattices by CFGs [41]. This means we can put an Rs transducer at the end of a tree transducer cascade, accepting observed inputs in string form (as we usually observe natural language inputs). Or if we have noisy input, as from speech or optical character recognition, then we can accept a whole FSA.

We may also want to use tree-to-string transducers in the forward direction, perhaps passing the result onto a string-based language model. In this case, we note that if the input to an RLs-transducer is an RTL, then the output is guaranteed to be a CFL. However, if we use an Rs-transducer, then the output may be non-CFL, as in the non-context-free yield language of the tree set in Figure 10.

8 Search

To solve problems with string automata, we often send an observed string backwards through a noisy-channel cascade of FSTs and FSAs—this ultimately transforms the string into a probabilistic FSA which encodes and ranks all possible "answer strings." We can then extract the best answer from this FSA with Dijkstra's algorithm [42], and we can extract the k-best paths with the fast algorithm of Eppstein [43], as is done in Carmel [7]. Mohri and Riley [44] also describe how to extract the k-best distinct strings from a weighted automaton.

The situation is similar with tree transducers. Once we manage to send our observed tree (or string, in the case of Rs) back through a noisy-channel cascade of transducers/acceptors, we wind up with a probabilistic RTG of answer trees. We next want to extract the best trees from that RTG. In general, this RTG may represent an infinite number of trees; related problems have been studied for finite forests in [45,46].

Knuth [47] has considered how to extract the highest-probability derivation tree from a probabilistic CFG (what he calls the **grammar problem**).[6] Here, we adapt his algorithm for RTG extraction. Knuth gives an explicit quadratic algorithm and points to an improvement using priority queues, which we work out in Algorithm 1. This is a generalization of Dijkstra's algorithm, greedily finalizing the cost of one RTG nonterminal at a time and remembering the lowest-cost production used to expand it. Each production potentially reduces the cost of its left-hand-side exactly once, when the last nonterminal in its right-hand-side is finalized.

To extract the k-best trees, it is useful to view RTG extraction as a hypergraph shortest path problem [48,49]. It appears that no very efficient k-best hyperpath algorithms exist yet.

9 Conclusion

We have investigated whether the benefits of finite-state string automata (elegant, generic operations in support of probabilistic reasoning for natural language processing) can be carried over into modeling with trees. We have distilled relevant theory and algorithms from the extensive literature on tree automata, and our conclusion is positive.

Many open problems now exist in finding efficient algorithms to support what is theoretically possible, and in implementing these algorithms in software toolkits. Here we list some of the problems of interest:

[6] Knuth's setting is actually more general—he considers weighted CFGs and various weight-combination functions.

Algorithm 1 Adaptation of Knuth's algorithm to extract the best tree from an RTG.

Input: **RTG** with n nonterminals N, and e productions indexed by $1 \leq i \leq e$: with rhs nonterminals T_i and lhs h_i, with a rule cost function of the form $c_i(\alpha : T_i \rightarrow \mathbb{R}) = w_i + \sum_{x \in T_i} \#_i(x)\alpha(x)$, where $\#_i(x)$ is the number of occurrences of nonterminal t in production i, and $w_i = \log P(i|h_i)$.

Output: For all $Y \in N$, $\pi[Y] = i$ is the index of the production that builds the cheapest tree possible from nonterminal $h_i = Y$ (with $\pi[T_i]$ recursively giving the cheapest trees for the child nonterminals), and $\mu[Y]$ is cost of that tree. $\pi[Y] = 0$ if there are no complete derivations from Y. Time complexity is $O(n \lg n + t)$ where (t is the total size of the input) if a Fibonacci heap is used, or $O(e \lg n + t)$ if a binary heap is used.

begin

  for $y \in N$ **do**
    $\mu[y] \longleftarrow \infty$
    $\pi[y] \longleftarrow 0$
    $Adj[y] \longleftarrow \{\}$
  $\mu[\omega] \longleftarrow 0$
  $\pi[\omega] \longleftarrow 0$
  $Adj[\omega] \longleftarrow \{\}$
  $P \longleftarrow$ ***HEAP-CREATE***()
  HEAP-INSERT$(P, \omega, 0)$
  for $1 \leq i \leq e, T_i = \{x_1, \ldots, x_k\}$ **do**
    $w[i] \longleftarrow w_i$
    if $k = 0$ **then**
      `/* fictitious sink nonterminal in rhs: ω */`
      $k \longleftarrow 1, x_1 \longleftarrow \omega$
    $r[i] \longleftarrow k$ `/* r[i] is the number of rhs nonterminals remaining before production i's cost is known. */`
    for $1 \leq j \leq k$ **do** $Adj[x_j] \longleftarrow Adj[x_j] \cup \{i\}$
  while $P \neq \emptyset$ **do**
    $y \longleftarrow$ ***HEAP-EXTRACT-MIN***(P)
    for $i \in Adj[y]$ **do**
      if $w[i] < \mu[h_i]$ **then**
        $w[i] \longleftarrow w[i] + \#_i(y)\mu[y]$
        $r[i] \longleftarrow r[i] - 1$
        if $r[i] = 0$ **then**
          if $w[i] < \mu[h_i]$ **then**
            if $\mu[h_i] = \infty$ **then** ***HEAP-INSERT***$(P, h_i, w[i])$
            else $HEAP - DECREASE - KEY(P, h_i, w[i])$, $\pi[h_i] \longleftarrow i$,
            $\mu[h_i] \longleftarrow w[i]$

end

1. What is the most efficient algorithm for selecting the k-best trees from a probabilistic regular tree grammar (RTG)?
2. How can efficient integrated search be carried out, so that all tree acceptors and transducers in a cascade can simultaneously participate in the best-tree search?
3. What search heuristics (beaming, thresholding, etc.) are necessary for efficient application of tree transducers to large-scale natural language problems?
4. What is the most efficient algorithm for composing probabilistic linear, non-deleting (RLN) tree transducers?
5. What is the most efficient algorithm for intersecting probabilistic RTGs?
6. What are the most efficient algorithms for forward and backward application of tree/tree and tree/string transducers?
7. For large tree transducers, what data structures, indexing strategies, and caching techniques will support efficient algorithms?
8. What is the linguistically most appropriate tree transducer class for machine translation? For summarization? Which classes best handle the most common linguistic constructions, and which classes best handle the most difficult ones?
9. Can compact RTGs encode high-performing tree-based language models with appropriate backoff strategies, in the same way that FSA tools can implement n-gram models?
10. What are the theoretical and computational properties of extended left-hand-side transducers (**x**)? E.g., is xRLN closed under composition?
11. Where do synchronous grammars [50,17] and tree cloning [15] fit into the tree transducer hierarchy?
12. As many syntactic and semantic theories generate acyclic graphs rather than trees, can graph transducers adequately capture the desired transformations?
13. Are there tree transducers that can move unbounded material over unbounded distances, while maintaining efficient computational properties?
14. In analogy with extended context-free grammars [35], are there types of tree transducers that can process tree sets which are not limited to a finite set of rewrites (e.g., $S \rightarrow NP\ VP\ PP^*$)?
15. Can we build tree-transducer models for machine translation that: (1) efficiently train on large amounts of human translation data, (2) accurately model that data by assigning it higher probability than other models, and (3) when combined with search algorithms, yield grammatical and accurate translations?
16. Can we build useful, generic tree-transducer toolkits, and what sorts of programming interfaces will be most effective?

10 Acknowledgements

This work was supported by NSF grant IIS-0428020 and by DARPA contract N66001-00-1-9814.

References

1. Knight, K., Graehl, J.: Machine transliteration. Computational Linguistics **24** (1998)

2. Mohri, M., Pereira, F., Riley, M.: The design principles of a weighted finite-state transducer library. Theor. Comput. Sci. **231** (2000)
3. Kaplan, R., Kay, M.: Regular models of phonological rule systems. Computational Linguistics **20** (1994)
4. Karttunen, L., Gaal, T., Kempe, A.: Xerox finite-state tool. Technical report, Xerox Research Centre Europe (1997)
5. van Noord, G., Gerdemann, D.: An extendible regular expression compiler for finite-state approaches in natural language processing. In Boldt, O., Juergensen, H., eds.: 4th International Workshop on Implementing Automata, Springer Lecture Notes in Computer Science (2000)
6. Kanthak, S., Ney, H.: Fsa: An efficient and flexible C++ toolkit for finite state automata using on-demand computation. In: Proc. ACL. (2004)
7. Graehl, J.: Carmel finite-state toolkit. `http://www.isi.edu/licensed-sw/carmel/` (1997)
8. Knight, K., Al-Onaizan, Y.: Translation with finite-state devices. In Farwell, D., Gerber, L., Hovy, E., eds.: Proceedings of the 3rd Conference of the Association for Machine Translation in the Americas on Machine Translation and the Information Soup (AMTA-98), Berlin, Springer (1998) 421–437
9. Brown, P., Della Pietra, S., Della Pietra, V., Mercer, R.: The mathematics of statistical machine translation: Parameter estimation. Computational Linguistics **19** (1993) 263–311
10. Kumar, S., Byrne, W.: A weighted finite state transducer implementation of the alignment template model for statistical machine translation. In: Proc. NAACL. (2003)
11. Och, F., Tillmann, C., Ney, H.: Improved alignment models for statistical machine translation. In: Proc. ACL. (1999)
12. Yamada, K., Knight, K.: A syntax-based statistical translation model. In: Proc. ACL. (2001) 523–530
13. Wu, D.: Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. Computational Linguistics **23** (1997) 377–404
14. Alshawi, H., Bangalore, S., Douglas, S.: Learning dependency translation models as collections of finite state head transducers. Computational Linguistics **26** (2000) 45–60
15. Gildea, D.: Loosely tree-based alignment for machine translation. In: Proc. ACL, Sapporo, Japan (2003)
16. Eisner, J.: Learning non-isomorphic tree mappings for machine translation. In: Proc. ACL (companion volume). (2003)
17. Melamed, I.D.: Multitext grammars and synchronous parsers. In: Proc. NAACL. (2003)
18. Knight, K., Marcu, D.: Summarization beyond sentence extraction: A probabilistic approach to sentence compression. Artificial Intelligence **139** (2002)
19. Pang, B., Knight, K., Marcu, D.: Syntax-based alignment of multiple translations extracting paraphrases and generating new sentences. In: Proc. NAACL. (2003)
20. Langkilde, I., Knight, K.: Generation that exploits corpus-based statistical knowledge. In: Proc. ACL. (1998)
21. Bangalore, S., Rambow, O.: Exploiting a probabilistic hierarchical model for generation. In: International Conference on Computational Linguistics (COLING 2000), Saarbrucken, Germany (2000)
22. Corston-Oliver, S., Gamon, M., Ringger, E.K., Moore, R.: An overview of Amalgam: A machine-learned generation module. In: Proceedings of the International Natural Language Generation Conference, New York, USA (2002) 33–40
23. Echihabi, A., Marcu, D.: A noisy-channel approach to question answering. In: Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics, Sapporo, Japan (2003)

24. Charniak, E.: Immediate-head parsing for language models. In: Proc. ACL. (2001)
25. Rounds, W.C.: Mappings and grammars on trees. Mathematical Systems Theory **4** (1970) 257—287
26. Thatcher, J.W.: Generalized^2 sequential machine maps. J. Comput. System Sci. **4** (1970) 339–367
27. Graehl, J., Knight, K.: Training tree transducers. In: Proc. NAACL. (2004)
28. Gécseg, F., Steinby, M.: Tree Automata. Akadémiai Kiadó, Budapest (1984)
29. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on `www.grappa.univ-lille3.fr/tata` (1997) release October, 1st 2002.
30. Galley, M., Hopkins, M., Knight, K., Marcu, D.: What's in a translation rule? In: NAACL, Boston, MA (2004)
31. Doner, J.: Tree acceptors and some of their applications. Journal of Computer and System Sciences **4** (1970) 406–451
32. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Series in Computer Science. Addison-Wesley, London (1979)
33. Johnson, M.: PCFG models of linguistic tree representations. Computational Linguistics **24** (1998) 613–632
34. Collins, M.: Three generative, lexicalised models for statistical parsing. In: Proc. ACL. (1997)
35. Thatcher, J.: Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. J. Comput. Syst. Sci. **1** (1967) 317–322
36. Yamasaki, K., Sodeshima, Y.: Fundamental properties of pushdown tree transducers (PDTT) —a top-down case. IEICE Trans. Inf. and Syst. **E76-D** (1993)
37. Engelfriet, J.: Top-down tree transducers with regular look-ahead. Math. Systems Theory **10** (1977) 289–303
38. Engelfriet, J.: Bottom-up and top-down tree transformations —a comparison. Math. Systems Theory **9** (1975) 198–231
39. Baum, L.E., Eagon, J.A.: An inequality with application to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. Bulletin of the American Mathematical Society **73** (1967) 360–363
40. Aho, A.V., Ullman, J.D.: Translations of a context-free grammar. Information and Control **19** (1971) 439–475
41. van Noord, G.: The intersection of finite state automata and definite clause grammars. In: Proc. ACL. (1995)
42. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
43. Eppstein, D.: Finding the k shortest paths. SIAM Journal on Computing **28** (1999) 652–673
44. Mohri, M., Riley, M.: An efficient algorithm for the n-best-strings problem. In: Proc. ICSLP. (2002)
45. Langkilde, I.: Forest-based statistical sentence generation. In: Proc. NAACL. (2000)
46. Nederhof, M.J., Satta, G.: Parsing non-recursive CFGs. In: Proc. ACL. (2002)
47. Knuth, D.: A generalization of Dijkstra's algorithm. Info. Proc. Letters **6** (1977)
48. Klein, D., Manning, C.: Parsing and hypergraphs. In: International Workshop on Parsing Technologies. (2001)
49. Nederhof, M.J.: Weighted deductive parsing and Knuth's algorithm. Computational Linguistics **29** (2003)
50. Shieber, S.M., Schabes, Y.: Synchronous tree-adjoining grammars. In: Proceedings of the 13th International Conference on Computational Linguistics. Volume 3., Helsinki, Finland (1990) 253–258