WEIGHTED TREE AUTOMATA AND TRANSDUCERS FOR SYNTACTIC NATURAL LANGUAGE PROCESSING

by

Jonathan David Louis May

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2010

Dedication

For Lorelei, who made it all worthwhile.

Acknowledgments

As I write these words I am overwhelmed that so many people have provided such a continuous force of encouragement, advice, and unrelenting positivity. Truly, I have been blessed to have them in my life.

My advisor, Kevin Knight, was just about the perfect person to guide me along this path. He was ever tolerant of my irreverent, frequently headstrong style, and gently corrected me when I strayed too far from my course of study, allowing me to make mistakes (such as coding up impossible algorithms) but helping me to learn from them. The other members of my committee have provided valuable feedback and insight—I have especially benefited from working closely with David Chiang and Daniel Marcu, and have welcomed the input from Sven Koenig, Shri Narayanan, and Fernando Pereira. I'm indebted to the continued guidance of Mitch Marcus at Penn, who helped me find BBN way back in 2001, and to that of Scott Miller, Lance Ramshaw, and Ralph Weischedel at BBN, who helped me find ISI.

In 2006, Andreas Maletti, then a Ph.D. student at the Technishe Universität Dresden, contacted Kevin and asked if he could come by ISI and give a presentation on his research in tree automata. Little was I to know at this time what a treasure trove had been opened, and what fruitful work would result from this meeting. This led to collaborations with

him as well as with Johanna Högberg, Heiko Vogler, and Matthias Büchse. The influence of these folks from TCS can be felt throughout this work. In particular, I am indebted to Heiko for extensive help in the drafting of Chapters 2 and 4.

For six years I have been fortunate to have Steve DeNeefe as my office-mate and academic sibling. He has taught me quite a lot about a wide variety of subjects, from computer science and math to Christianity and fatherhood. The number of other truly amazing people at ISI I have had the good fortune of working with is astounding, and I hope I manage to acknowledge them all here: Erika Barragan-Nunez, Rahul Bhagat, Gully Burns, Hal Daumé III, Victoria Fossum, Alex Fraser, Jonathan Graehl, Ulf Hermjakob, Dirk Hovy, Ed Hovy, Liang Huang, Zornitsa Kozareva, Kary Lau, Rutu Mehta, Alma Nava, Oana Postolache, Michael Pust, David Pynadath, Sujith Ravi, Deepak Ravichandran, Jason Riesa, Tom Russ, Radu Soricut, Ashish Vaswani, Jens Vöckler, Wei Wang, and Kenji Yamada. In addition, I am grateful to ISI for its ability to attract visiting professors, graduate students, and other luminaries. This has enabled me to get to know John DeNero, Erica Greene, Carmen Heger, Adam Pauls, Gerald Penn, Bill Rounds, Magnus Steinby, Cătălin Tîrnăucă, and Joseph Turian.

In my times of greatest doubt and despair my friends and family have always been rock-solid, fully supportive of me and confident that I would succeed, even when I didn't believe in myself. I'm so lucky to have their unyielding and undying love. Marco Carbone and Lee Rowland, friends from way back, hover just barely too far away in Nevada. Glenn Østen Anderson, Nausheen Jamal, and Ben Plantan have brightened our door with not-frequent-enough visits and may finally move here some day. My aunt René Block Baird and cousins Kalon, April, and Serena Baird engage me with constant

love and affection. And I literally could not be here without the constant sacrifice, support, and occasional copy-editing of my parents, Howard May and Marlynn Block, and my step-parents, Irena May and Jerry Levine. Finally, though it has been practically effortless, I still consider the greatest accomplishment and benefit that this document represents to be my meeting, falling in love with, and marrying Lorelei Laird.

Table of Contents

List Of Tables

List Of Figures

List of Algorithms

Abstract

Weighted finite-state string transducer cascades are a powerful formalism for models of solutions to many natural language processing problems such as speech recognition, transliteration, and translation. Researchers often directly employ these formalisms to build their systems by using toolkits that provide fundamental algorithms for transducer cascade manipulation, combination, and inference. However, extant transducer toolkits are poorly suited to current research in NLP that makes use of syntax-rich models. More advanced toolkits, particularly those that allow the manipulation, combination, and inference of weighted extended top-down tree transducers, do not exist. In large part, this is because the analogous algorithms needed to perform these operations have not been defined. This thesis solves both these problems, by describing and developing algorithms, by producing an implementation of a functional weighted tree transducer toolkit that uses these algorithms, and by demonstrating the performance and utility of these algorithms in multiple empirical experiments on machine translation data.

Chapter 1

Introduction: Models and Transducers

1.1 A cautionary tale

One of the earliest syntactic parsers was built in 1958 and 1959 to run on the Univac 1 computer at the University of Pennsylvania and attempts were made to recreate the parser some forty years hence on modern machines [60]. Given that computer science was in its infancy at the time of the parser's creation and much had changed in the interim, it is not surprising that the resurrectors relied on the hundreds of pages of flowcharts and program specifications describing the system as guidance, rather than the original assembly code itself. Unfortunately, due to ambiguities in the documentation and damage to the archives, some guesses had to be made in reimplementation, and thus the reincarnation of the parser is only an approximation of the original system at best. As the resurrectors were faithful to the design of the original parser, however, they built the modern incarnation of the parser as a single piece of code designed to do a single task. Of course, this time they wrote the program in C rather than assembly. If, forty years from now, a new generation of computer science archaeologists wishes to re-recreate

the parser, one hopes the C language is still understandable, and that the source code survives. If this source is for some reason unavailable, the team will once again have to re-write the parser from documentation alone. As before, if the documentation is incomplete or imprecise, only an approximation of the original program will be built, and all the work of the resurrectors will be for naught.

1.2 Transducers to the rescue

As it happens, this parser, now called Uniparse, was designed as a cascade of *finite-state transducers*, abstract machines that read an input tape and write an output tape based on a set of state transition rules. Transducers have been widely studied and have numerous attractive properties; among them is the property of *closure under composition*—the transformations accomplished by a sequence of two transducers can be captured by a single transducer. These properties, along with effective algorithms that take advantage of them, such as an algorithm to quickly construct the composition of two transducers, allow any program written in the form of a transducer cascade, as Uniparse was designed, to be easily and effectively handled by a generic program that processes and manipulates transducers.

Rather than writing Uniparse from the original design schematics in custom assembly or C, the resurrectors could have encoded the schematics themselves, which are already written as transducers, into a uniform format that is suitable for reading by a generic finite-state transducer toolkit and used transducer operations such as composition and

projection (the obtaining of either the input or output language of a transducer) to perform the parsing operations, rather than writing new code. By expressing Uniparse as transducer rules, the resurrectors would have been able to combine the transducer design with its code implementation. Any alterations to the original design would have been encoded in the set of transducer rules, preventing any "hacks" from hiding in the code base. A file containing solely transducer rules requires no additional documentation to explain any hidden implementation, as the entire implementation is represented in the rules. Most importantly, aside from formatting issues, a file of transducer rules is immune from the havoc time wreaks on compiled code. Future generations could use their transducer toolkits on these files with a trivial number of changes.

Of course, someone has to build the transducer toolkit itself. And this naturally raises the question: Is implementing a program for finite-state transducers rather than for a syntactic parser simply trading one specific code base for another? Thankfully, transducer cascades are useful for more than light deterministic parsing. In the fields of phonological and morphological analysis Kaplan and Kay [63] realized the analysis rules linguists developed could be encoded as finite-state transducer rules, and this led first to a transducer-based morphological analysis system [80] and eventually to the development of XFST [70], an entire finite-state toolkit complete with the requisite algorithms needed to manipulate transducers and actually get work done. The set of natural language transformation tasks capturable by regular expressions such as date recognition, word segmentation, some simple part-of-speech tagging, and spelling correction, and light parsing such as that done by Uniparse can be expressed as cascades of finite-state transducers [69]. The availability of this toolkit allowed researchers to

simply write their models down as transducer rules and allow the toolkit to do the processing work.

1.3 A transducer model of translation

A concrete example is helpful for demonstrating the usefulness of transducer cascades. Imagine we want to build a small program that translates between Spanish and English. Here is a simple model of how an English sentence becomes a Spanish sentence:

- An English sentence is imagined by someone fluent in English.
- The words of the sentence are rearranged—each word can either remain in its original position or swap places with the subsequent word.
- Each word is translated into a single Spanish word.

Such a model obviously does not capture all translations between Spanish and English, but it does handle some of them, and thus provides a good motivating example. Moreover, it is fairly easy to design software to perform each step of the model, even if it might be hard to design a single piece of software that encodes the entire model, all at once. In this way we are espousing the "conceptual factoring" envisioned by Woods [133].

Having envisioned this model of machine translation, an enterprising student could set about writing a program from scratch that implements the model. However, rather

than enduring lengthy coding and debugging sessions the student could instead represent the model by the following cascade of finite-state transducers which may be composed into a single transducer using a toolkit such as XFST:

- A finite-state *automaton A*, which is an English *language model*, i.e., it recognizes the sentences of English. This is a special case of a finite-state transducer—one where each rule has a single symbol rather than separate reading and writing symbols. A simple automaton for English maintains a state associated with the last word it saw, and only has transitions for valid subsequent words. Let us semantically associate the state q_x with cases where the last recognized word was x. Let q_{START} be the state representing the beginning of a sentence. We write rules like:

  $q_{\text{START}} \xrightarrow{\text{the}} q_{\text{the}}$ $\quad q_{\text{green}} \xrightarrow{\text{ball}} q_{\text{ball}}$

  $q_{\text{the}} \xrightarrow{\text{green}} q_{\text{green}}$ $\quad q_{\text{green}} \xrightarrow{\text{horse}} q_{\text{horse}}$

  $q_{\text{the}} \xrightarrow{\text{ball}} q_{\text{ball}}$ $\quad \ldots$

  and so on. This automaton allows phrases such as "the ball" and "the green horse" but not "green ball the" or "horse green".

- A reordering transducer B with rule patterns of the following form for all words a and b:

  $r \xrightarrow{\text{a:}\epsilon} r_a$ $\quad r \xrightarrow{\text{a:a}} r$ $\quad r_a \xrightarrow{\text{b:b a}} r$

  Here, ϵ on the right side means no symbol is written when the rule is invoked, though the symbol on the left is consumed. The rules are instantiated for each possible pair of words, so given the English vocabulary {the, ball, green} we would have:

$r \xrightarrow{\text{the:}\epsilon} r_{\text{the}}$ $r \xrightarrow{\text{ball:}\epsilon} r_{\text{ball}}$ $r \xrightarrow{\text{green:}\epsilon} r_{\text{green}}$

$r \xrightarrow{\text{the:the}} r$ $r \xrightarrow{\text{ball:ball}} r$ $r \xrightarrow{\text{green:green}} r$

$r_{\text{the}} \xrightarrow{\text{ball:ball the}} r$ $r_{\text{ball}} \xrightarrow{\text{ball:ball ball}} r$ $r_{\text{green}} \xrightarrow{\text{ball:ball green}} r$

$r_{\text{the}} \xrightarrow{\text{green:green the}} r$ $r_{\text{ball}} \xrightarrow{\text{green:green ball}} r$ $r_{\text{green}} \xrightarrow{\text{green:green green}} r$

$r_{\text{the}} \xrightarrow{\text{the:the the}} r$ $r_{\text{ball}} \xrightarrow{\text{the:the ball}} r$ $r_{\text{green}} \xrightarrow{\text{the:the green}} r$

- A one-state transducer C that translates between English and Spanish, e.g.:

  $s \xrightarrow{\text{ball:pelota}} s$ $s \xrightarrow{\text{horse:caballo}} s$

  $s \xrightarrow{\text{the:el}} s$ $s \xrightarrow{\text{green:verde}} s$

  $s \xrightarrow{\text{the:la}} s$ ...

These three transducers can be composed together using classic algorithms to form a single translation machine, D, that reorders valid English sentences and translates them into (possibly invalid) Spanish in a single step. Run in reverse, D translates arbitrary sequences of Spanish words and, if possible, reorders them to form valid English sentences.

Next, a candidate Spanish phrase can be encoded as a simple automaton E, with q_0 as the initial state and the following rules:

$q_0 \xrightarrow{\text{la}} q_1$ $q_1 \xrightarrow{\text{pelota}} q_2$ $q_2 \xrightarrow{\text{verde}} q_3$

This automaton represents exactly the phrase "la pelota verde". It can be composed to the right of D, forming a transducer F that reorders valid English sentences and, if possible, translates them into exactly the phrase "la pelota verde". The domain projection of F, then, is an automaton that represents all valid English translations of the Spanish phrase, which in this case is the single phrase "the green ball." Notice that this translation machine was built from simple transducers: A, which only recognizes valid English

but is completely unaware of Spanish, B, which reorders English without any regard for proper grammar, C, which translates between English and Spanish but is not constrained to the proper grammar of either language, and D, the candidate Spanish sentence. By chaining together several simple transducers that each perform a limited task we can quickly build complicated and powerful systems.

There are many problems with this translation model. One of the most obvious is that we cannot handle cases where the number of English and Spanish words is not the same, so the translation between "I do not have" and "No tengo" is impossible. However, successive refinements and introductions of additional transducers, some with ϵ-rules, can help make the system better. Implementing the same refinements and model changes in a custom code implementation can require many more tedious coding and debugging cycles.

There is a more fundamental problem with this model that cannot easily be resolved by reconfiguring the transducers. If there are multiple valid answers, how do we know which to choose? In this framework a transformation is either correct or incorrect; there is no room for preference. We are faced with the choice of either overproducing and not knowing which of several answers is correct, or underproducing and excluding many valid transformations. Neither of these choices is acceptable.

1.4 Adding weights

While the development of a finite-state transducer toolkit was very helpful for attacking the problems of its age, advances in computation allowed modeling theory to expand

beyond a level conceivable to the previous generation. Specifically, the availability of large corpora and the ability to process these corpora in reasonable time coupled with a move away from prescriptivist approaches to computational linguistics motivated a desire to represent *uncertainty* in processing and to empirically determine the likelihood of a processing decision based on these large corpora. Transducers that make a simple yes-or-no decision were no longer sufficient to represent models that included confidence scores. Researchers at AT&T designed FSM, a toolkit that harnesses *weighted* finite-state transducers—a superset of the formalism supported by XFST [103]. The association of weights with transducer rules affected the previous algorithms for composition and introduced new challenges. Along with the physical toolkit code, new algorithms were developed to cope with these challenges [109, 104, 100]. Carmel [53], a toolkit with similar properties as FSM, but with an algorithm for EM training of weighted finite-state transducers [38], was also useful in this regard. These toolkits and others like them were quite helpful for the community, as the state of NLP models had greatly expanded to include probabilistic models and without a decent weighted toolkit around, the only option to test these models was nose-to-the-grindstone coding of individual systems. Subsequent to their invention and release a number of published results featured the use of these toolkits and transducer formalisms in model design [110, 123, 74, 24, 137, 83, 79, 94].

Returning to our translation example, we can see that some word sequences are more likely than others. Additionally, some translational correspondences are more likely than others. And perhaps we want to encourage the word reordering model to preserve English word order whenever possible. This information can be encoded using

weights on the various rules. Consider the language model A: both "green ball" and "green horse" are valid noun phrases, but the former is more likely than the latter. In the absence of any other evidence, we would expect to see "ball" after "green" more often than we would see "horse". However, we would expect to see "green horse" more often than "green the". By looking at a large amount of real English text, we can collect statistics on how often various words follow "green", and then calculate a probability for each possible word. Such weights are added to the relevant rules as follows:

$$q_{\text{green}} \xrightarrow{\textit{ball}/0.8} q_{\text{ball}} \quad q_{\text{green}} \xrightarrow{\textit{horse}/0.19} q_{\text{horse}} \quad q_{\text{green}} \xrightarrow{\textit{the}/0.01} q_{\text{the}}$$

The product of the weights of each rule used for a given sentence is the weight of the whole sentence. Notice that we did not exclude the very unlikely sequence "green the". However, we gave that rule very low weight, so any sentence with that sequence would be quite unlikely. This represents an increase in power over the unweighted model. Previously, we would not want to allow such a phrase, as it almost certainly would be wrong. Now we can still acknowledge the extreme unlikeliness of the phrase, but allow for the rare situation where it is the most likely choice available after other possibilities have been eliminated.

We demonstrate our preferences in the other transducers in the chain through weights similarly. The reordering transducer, for instance, should favor some reorderings and disfavor others. An English noun phrase that contains an adjective (such as "green ball") would typically be translated with the noun first in Spanish. However, no reordering would be done for a noun phrase without adjectives (such as "the ball"). The following weighted transducer rules reflect these preferences:

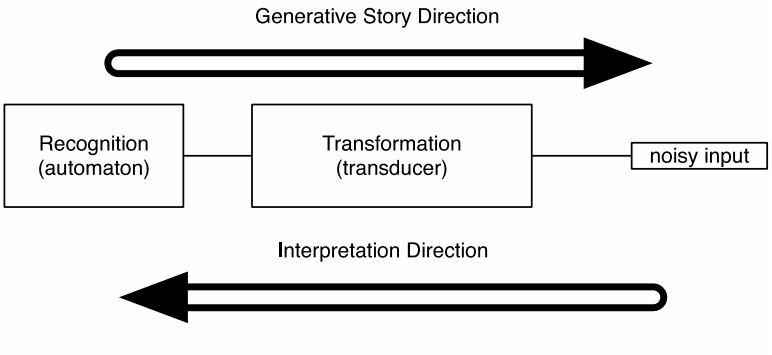


Figure 1.1: The general noisy channel model. The model is proposed in the "story" direction but used in the "interpretation" direction, where a noisy input is transformed into the target domain and then validated against the recognizer.

$$q \xrightarrow{\text{the:}\epsilon/0.1} q_{\text{the}} \qquad q \xrightarrow{\text{green:}\epsilon/0.7} q_{\text{green}}$$

$$q \xrightarrow{\text{the:the}/0.9} q \qquad q \xrightarrow{\text{green:green}/0.3} q$$

$$q_{\text{the}} \xrightarrow{\text{ball:ball the}/1} q \qquad q_{\text{green}} \xrightarrow{\text{ball:ball green}/1} q$$

$$q_{\text{the}} \xrightarrow{\text{green:green the}/1} q \qquad q_{\text{green}} \xrightarrow{\text{green:green green}/1} q$$

$$q_{\text{the}} \xrightarrow{\text{the:the the}/1} q \qquad q_{\text{green}} \xrightarrow{\text{the:the green}/1} q$$

The translation model we have built is getting more and more powerful. It has begun to take on the shape of a very useful and often-applied general model of transformation—the *noisy channel model* [118]. The key principle behind this model, depicted in Figure 1.1, is the separation of a sensible transformation task into the cascade of a transformation task (without regard to sensibility of the output) followed by a recognition task that only permits sensible output. By simply substituting the appropriate transducer or transducers into our chain we can perform diverse tasks without altering the underlying machinery. If we remove the permutation and translation transducers from our model, and instead add a word-to-phoneme transducer followed by a phoneme-to-speech signal transducer, we can perform speech recognition on some given speech signal input. If

we replace the word language model with a part-of-speech language model and the transducer cascade with a tag-to-word transducer we can perform part-of-speech tagging on some word sequence input. In all of these cases the only work required to transform our translation machine into a speech signal-recognition machine or a grammar markup machine is that of rule set construction, which is in its essence pure model design. The underlying mechanics of operation remain constant. This illustrates the wide power of weighted finite-state toolkits and explains why they have been so useful in many research projects.

1.5 Going further

We must acknowledge that our model is still a simplification of "real" human translation, and, for the foreseeable future, this will continue to be the case, as we are limited by practical elements, such as available computational power and data. This has long been a concern of model builders, and thus in every generation, compromises are made. In the 1950s, severe memory limitations precluded anything so general as a finite-state-machine toolkit. In the 1980s, few useful corpora existed and normally available computational power was still too limited to support the millions of words necessary to adequately train a weighted transducer cascade. In the present age we have more computational power and large available corpora. We should consider whether there are yet more limitations imposed by the technology of the previous era that can now be relaxed.

One particular deficiency that arises, particularly in our translation model, is the requirement of linear structure, that is, a sequential left-to-right processing of input.

Human language translation often involves massive movement depending on syntactic structure. In the previous examples we were translating between English and Spanish, two predominately Subject-Verb-Object (SVO) languages, but what if we were translating between English and Japanese? The former is SVO but the latter is SOV. The movement of an arbitrarily long object phrase after an arbitrarily long verb phrase (or vice-versa) is simply not feasible with a formalism that at its fundamental core must process its input in linear order with a finite number of states.

The disconnect between linear processes and the more hierarchical nature of natural language is an issue that has long been raised by linguists [22]. However, practical considerations and a realization that a limited model with sufficient data was good enough for the time being led empirical research away from syntactic models for nearly half a century.

A survey of recent work in NLP shows there is evidence that syntax-rich empirical models may now be practical. Recent work with successful practical results in language modeling [20], syntactic parsing [25], summarization [77], question answering [37], and machine translation [136, 47, 31], to name a few, clearly indicates there are gains to be made in syntactic NLP research. One common thread these papers have, however, is that the results behind the presented models were obtained via custom construction of one-off systems. Weighing the benefits of the syntax translation models of Yamada and Knight [136] and Galley et al. [47], for example, requires access to the projects' respective code bases or re-engineering efforts. Consequently, few if any such comparative studies exist, and the cycle of model improvement is generally only possible for the original model engineers. Such a limitation is harmful to the community.

1.6 Better modeling through tree transducers

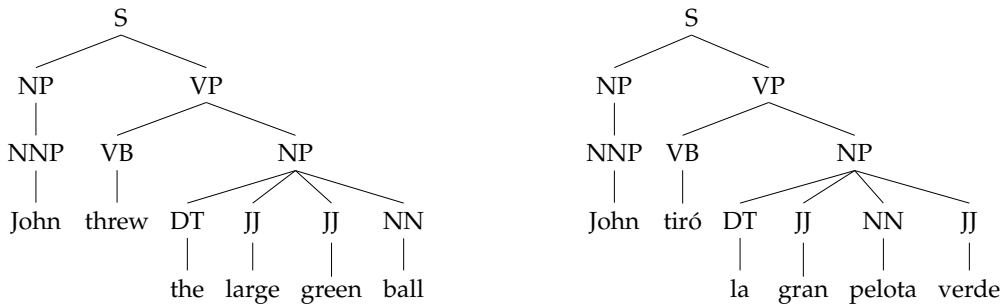


Figure 1.2: An (English, Spanish) tree pair whose transformation we can capture via tree transducers.

The good news is many of these models can be expressed as a cascade of *finite-state tree transducers*. Tree transducers were invented independently by Rounds [116] and Thatcher [128] to support Chomskyan linguistic theories [23]. The theory community then conducted extensive study into their properties [48, 49, 27] without much regard for the original linguistic motivation. Weighted tree automata, the direct analogue of weighted string automata, have been studied as a formal discipline [8, 41], as have weighted regular tree grammars [1]. This latter formalism generates the same class of tree languages as weighted tree automata, and closely resembles weighted context-free grammars, so it is the preferred formalism used in this thesis.

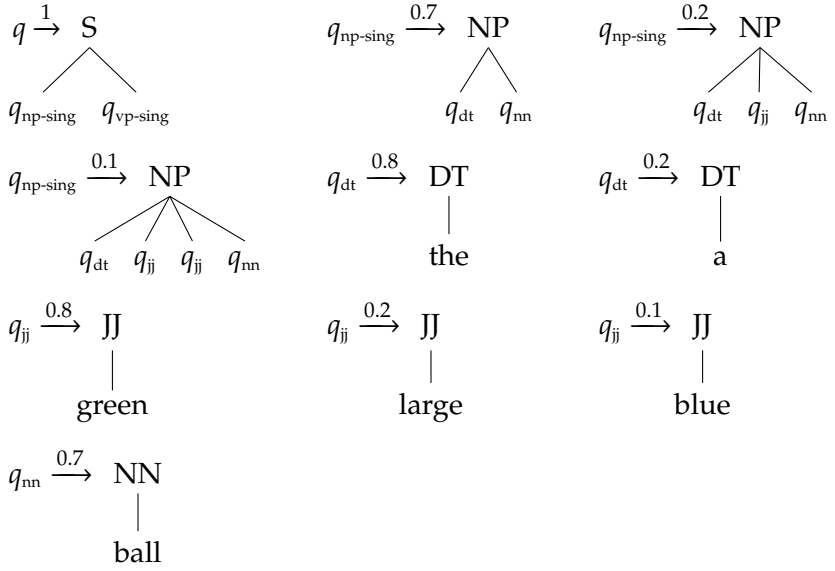
Let us consider how syntax can improve our previous toy translation model. Imagine we are now trying to translate between English and Spanish sentences annotated with syntactic trees, such as those in Figure 1.2. We can accomplish this with weighted regular tree grammars and top-down tree transducers, which are introduced formally in Chapter 2, but which we describe informally now, by way of comparison to the previously described finite-state (string) automata and transducers.

String automaton rules are of the form $q \xrightarrow{a/w} r$, indicating that, with weight w, a machine in state q writes symbol a and changes to state r, where it continues, writing further results to the right of a in the output string. Tree grammar rules, on the other hand, are of the form $q \xrightarrow{w} \tau$. This rule indicates that, with weight w, a machine in state q writes tree τ. Some of the leaves of τ may be states such as r. If they are, further rules are used to write trees at these points in τ.

String transducer rules are of the form $q \xrightarrow{\text{a:b/w}} r$, indicating that, with weight w, a machine in state q reading symbol a writes symbol b and changes to state r, where it continues by processing the symbol to the right of a in the input string and writing the result to the right of b in the output string. Tree transducer rules, on the other hand, are of the form $q.\gamma(x_1 \ldots x_n) \xrightarrow{w} \tau$. Such a rule indicates that, with weight w, a machine in state q reading a tree that has a root label γ and n immediate children writes tree τ. Some of the leaves of τ are of the form $r.x_k$, indicating that the kth immediate child of the tree should be processed in state r and the result written at that location in the output tree.

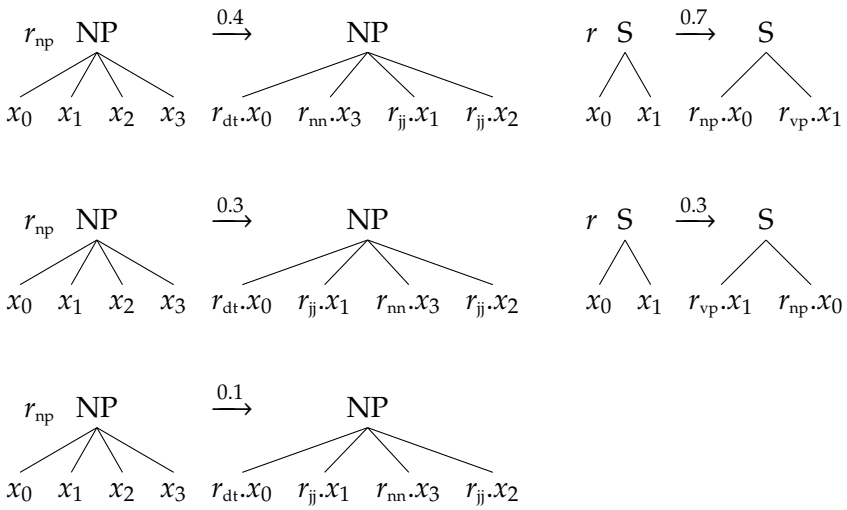
Since we are now considering syntax, we are no longer simply transforming between surface strings in English and Spanish, but between syntactic trees. Consider the power our transducer chain has now that we have introduced this formalism:

- Rather than recognizing valid English phrases, our new tree grammar A', the descendant of the string automaton A, must now recognize valid English trees. Let q be the initial state of A'. Here's what some rules from A' could look like:



Already we can see some ways in which this grammar is more powerful than its string automaton ancestor. Words are conditioned on their parts of speech, so a more appropriate distribution for particular word classes can be defined. The top-down approach enables long-distance dependencies and global requirements. For example, the rule $q \to \text{S}(q_{\text{np-sing}}\ q_{\text{vp-sing}})$ indicates the sentence will have both a noun phrase and verb phrase, and that both will be singular, even though it is not yet known where the singular noun and verb will appear within those phrases.

- Although we can now get quite creative with our permutation model, we can demonstrate the increased power of tree transducers by designing a B' that has the same idea expressed in B, i.e., allow re-ordering one level at a time. The initial state is r and these are some rules:



The reordering at the top of the tree, swapping the positions of arbitrarily large noun phrases and verb phrases, does sentence-wide permutation in a single step. Doing the equivalent with string-based transducers would require a unique state for every possible noun phrase, making such an operation generally impossible. The lower-level inversion could feasibly be accomplished with a string transducer, but it would still require states that encode the sequence of adjectives seen until the noun at the end of the phrase is reached.

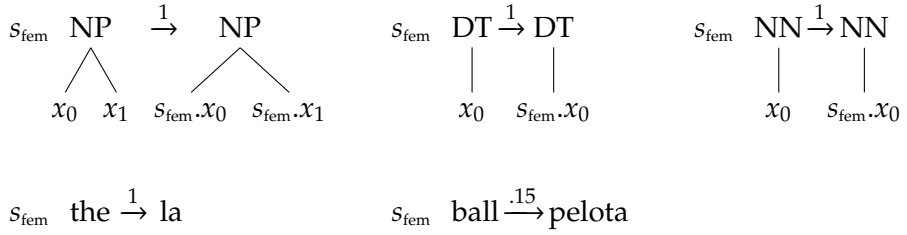
- We can use the power of syntax to our advantage in the design of the translation transducer, C'. Recall that C has a set of word-to-word rules that do not take context into account. We can easily take context into account in C' by our selection of states. The following selection of rules from C' indicate that all words in the noun phrase will be singular and feminine, ultimately constraining "the" to translate as "la" so as to match the translation of "ball" as "pelota."

Operation	String		Tree	
	unweighted	weighted	unweighted	weighted
determinization	Yes	Yes	Yes	No
K-best	N/A	Yes	N/A	Alg
intersection	Yes	Yes	Yes	PoC
EM training	Yes		Alg	

Table 1.1: Availability of algorithms and implementations for various classes of automata. Yes = an algorithm is known and an implementation is publicly available. Alg = an algorithm is known but no implementation is known to be available. PoC = a proof of concept of the viability of an algorithm is known but there is no explicit algorithm. No = no methods have been described.

Operation	String		Tree	
	unweighted	weighted	unweighted	weighted
composition	Yes	Yes	PoC	PoC
domain and range projection	Yes	Yes	PoC	PoC
application	Yes	Yes	PoC	PoC
EM training	Yes		Alg	

Table 1.2: Availability of algorithms and implementations for various classes of transducers, using the key described in Table 1.1.



1.7 Algorithms for tree transducers and grammars

Before deciding to build a new toolkit it is useful to take stock of the availability of needed algorithms. Tables 1.1 and 1.2 identify a set of useful operations for automata and transducers. The operations under consideration all have efficient algorithms that have been implemented for weighted string automata and transducers, primarily in FSM, OpenFst, and Carmel. For tree automata, the situation is more dire—algorithms

for determinization and intersection of unweighted tree automata exist and have been implemented in Timbuk [50] but as there were no previous weighted tree automata toolkits, it is understandable that there would be no implementations of relevant algorithms. No tree transducer software had implementations of composition, projection, or application, and perhaps it is understandable, as these operations generally have not been described in algorithmic manners by practitioners of formal language theory, who do not need such details for their proofs. A good example which illustrates this point of view is the classic construction for unweighted top-down tree transducer composition by Baker [6]. Her construction essentially directs that a transducer rule in a composition of two transducers, M_1 and M_2 be formed by combining a rule $q.\sigma \to u$ from the first, where u is some tree, with a state p from the second, to form$(q, p).\sigma \to t$, for every t that is a transformation of u starting in p by M_2.[1] Such a construction is certainly correct, and it is known that under certain restrictions on M_1 and M_2 a finite set of t can be found. Furthermore, in the weighted extension of this construction by Maletti [89] the additional constraint that the weight of the transformation from u to t by M_2 be calculable is known to be determinable for the closure cases. But while this is sufficient for proving theorems it does not suffice for building software. The means of finding every t for u is not described; the construction demonstrates it *can* be built, but does not describe a practical algorithm for *how* it should be built. The status of such operations is indicated in Tables 1.1 and 1.2 as "proofs of concept", in that no concrete, implementable algorithms have been shown for their operation. I have thus delved into these declarative constructs and exposed algorithms useful to the software-writing community that implement these operations.

[1] This is a significant paraphrase from page 195 of [6], as we don't wish to introduce detailed terminology just yet.

And, in some cases, I have designed new algorithms, or extended existing algorithms or declarative constructs to the weighted tree automaton and weighted extended top-down tree transducer case.

1.8 Building a new toolkit

Weighted tree machines are a powerful formalism for natural language processing models, and as the example above indicates, constructing useful models is not difficult. However, prior to this thesis there was one main roadblock preventing progress in weighted tree machine modeling of the scale seen for weighted string machines: no appropriate toolkit existed. There were some existing tree automata and transducer toolkits [15, 50, 58, 34] but these are unweighted, a crucial omission in the age of data-driven modeling. Additionally, they are chiefly aimed at the logic and automata theory community and are not suited for the needs of the NLP community.

To guide me in this work I followed the lead of previous toolkits. I have chosen simple design semantics and a small set of desired operations. As noted above, some algorithms for these operations already existed and could more or less be directly implemented, some had to be inferred from declarative constructs, and some required novel algorithm development. The resulting toolkit, Tiburon, can read, write, and manipulate large weighted tree-to-tree and tree-to-string transducers, regular tree grammars, context-free grammars, and training corpora.

To summarize, these are the contributions provided in my thesis:

- I present algorithms for intersection of weighted regular tree grammars, composition and projection of weighted tree transducers, and application of weighted tree transducers to grammars that were previously only declarative proofs of concept. I also demonstrate the connection between classic parsing algorithms and one particular kind of application; that of tree-to-string transducers to strings. I provide these algorithms for weighted extended tree transducers, a formalism that is more useful to the NLP community than classical weighted tree transducers, though somewhat neglected by formal language theory.

- I present a novel algorithm for practical determinization of acyclic weighted regular tree grammars and show the algorithm's empirical benefits on syntax machine translation and parsing tasks. In joint work with colleagues from formal language theory we prove correctness and examine the applicability of the algorithm to some classes of cyclic grammars.

- I present novel algorithms for application of tree transducer *cascades* to grammar input, as well as more efficient *on-the-fly* versions of these grammars that take advantage of lazy evaluation to avoid unnecessary exploration. I demonstrate the performance advantages of these on-the-fly algorithms.

- I demonstrate the use of weighted tree transducers as formal models in the improvement of the state of the art in syntax machine translation by using an EM training algorithm for weighted tree transducers to improve automatically induced word alignments in bilingual training corpora, leading to significant BLEU score increases in Arabic-English and Chinese-English MT evaluations.

- I provide Tiburon, a tree automaton and transducer toolkit that provides these fundamental operations such that complicated tree-based models may easily be represented. The toolkit has already been used on several research projects and theses [56, 113, 126, 12, 125].

Here is an outline of the remainder of this thesis:

- Chapter 2 provides a formal basis for the remainder of the work. It defines key structures such as trees, grammars, and transducers, and presents algorithms for basic tree transducer and grammar operations.

- Chapter 3 presents the first practical determinization algorithm for weighted regular tree grammars. I outline the development of this algorithm, show, in joint work, a proof of correction, and demonstrate its effectiveness on two real-world experiments.

- Chapter 4 presents novel methods of efficient inference through tree transducers and transducer cascades. It also contains a detailed description of on-the-fly inference algorithms which can be faster and use less memory than traditional inference algorithms, as demonstrated in empirical experiments on a machine translation cascade.

- Chapter 5 presents a method of using tree transducer training algorithms to accomplish significant improvements in state-of-the-art syntax-based machine translation.

- Chapter 6 presents Tiburon, a toolkit for manipulating tree transducers and grammars. Tiburon contains implementations of many of the algorithms presented in previous sections.
- Chapter 7 concludes this work with a high-level view of what has been presented in the previous chapters and outlines useful future directions.

Chapter 2

Weighted Regular Tree Grammars and Weighted Tree Transducers

In this chapter we introduce basic terminology used throughout this thesis. In particular we define the formal tree grammars and tree transducers that are the fundamental structures this thesis is concerned with. We also make note of basic algorithms for combining, transforming, and manipulating these machines. Although essentially all of the algorithms in this chapter were known previous to this work, much of it has not been presented in this way, designed for those seeking to provide an implementation. Additionally, some algorithms known to the community as "folklore" and various extensions to more general formalisms are presented.

Because a good deal of formal notions are presented in this chapter, particularly early on, it may behoove the reader to read this chapter lightly, and return to relevant sections when the precise definition of a term or piece of symbology is needed.

2.1 Preliminaries

Much of the notation we use is adapted from Section 2 of Fülöp and Vogler [45], some quite extensively. Additional notation is adapted from Section 3.1.1 of Mohri [101].

2.1.1 Trees

A *ranked alphabet* is a tuple (Σ, rk) consisting of a finite set Σ and a mapping $rk : \Sigma \to \mathbb{N}$ which assigns a *rank* to every member of Σ. We refer to a ranked alphabet by its carrier set, Σ. Frequently used ranked alphabets are Σ, Δ, and Γ. For every $k \in \mathbb{N}$, $\Sigma^{(k)} \subseteq \Sigma$ is the set of those $\sigma \in \Sigma$ such that $rk(\sigma) = k$. When it is clear, we write $\sigma^{(k)}$ as shorthand for $\sigma \in \Sigma^{(k)}$. We let $X = \{x_1, x_2, \ldots\}$ be a set of *variables*; $X_k = \{x_1, \ldots, x_k\}, k \in \mathbb{N}$. We assume that X is disjoint from any ranked alphabet used in this work. A *tree* $t \in T_\Sigma$ is denoted $\sigma(t_1, \ldots, t_k)$ where $k \in \mathbb{N}$, $\sigma \in \Sigma^{(k)}$, and $t_1, \ldots, t_k \in T_\Sigma$.[1] For $\sigma \in \Sigma^{(0)}$ we write $\sigma \in T_\Sigma$ as shorthand for $\sigma()$. For every set A disjoint from Σ, let $T_\Sigma(A) = T_{\Sigma \cup A}$, where for all $a \in A$, $rk(a) = 0$. We define the *positions* of a tree $t = \sigma(t_1, \ldots, t_k)$, for $k \geq 0$, $\sigma \in \Sigma^{(k)}$, and $t_1, \ldots, t_k \in T_\Sigma$, as a set $pos(t) \subset \mathbb{N}^*$ such that $pos(t) = \{\varepsilon\} \cup \{iv \mid 1 \leq i \leq k, v \in pos(t_i)\}$. The set of leaf positions $leaves(t) \subseteq pos(t)$ are those positions $v \in pos(t)$ such that for no $i \in \mathbb{N}$, $vi \in pos(t)$. We presume standard lexicographic orderings $<$ and $\leq$ on pos. Let $t, s \in T_\Sigma$ and $v \in pos(t)$. The *label* of t at position v, denoted by $t(v)$, the *subtree* of t at v, denoted by $t|_v$, and the *replacement* at v by s, denoted by $t[s]_v$, are defined as follows:

[1] Note that these are *ranked* trees, but in natural language examples we may see the same symbol with more than one rank. For example, both the phrases "boys" and "the boys" are noun phrases, and one frequently sees their tree representations as, respectively, NP(boys) and NP(the boys). From a formal perspective we should distinguish the parent nonterminals, writing, e.g., NP_1(boys) and NP_2(the boys), where $NP_1 \in \Sigma^{(1)}$ and $NP_2 \in \Sigma^{(2)}$, and indeed these two parent symbols are interpreted differently, but to simplify notation and remain consistent with common practices, they will both appear on the page as NP.

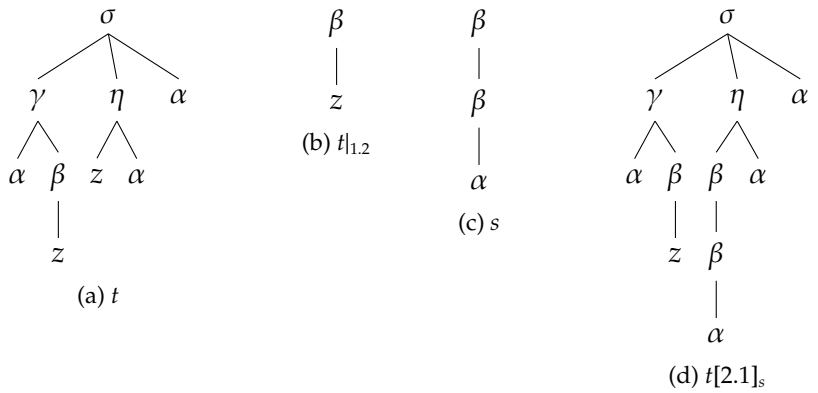


Figure 2.1: Trees from Example 2.1.1.

1. For every $\sigma \in \Sigma^{(0)}$, $\sigma(\varepsilon) = \sigma$, $\sigma|_\varepsilon = \sigma$, and $\sigma[s]_\varepsilon = s$.

2. For every $t = \sigma(t_1, \ldots, t_k)$ such that $k = rk(\sigma)$ and $k \geq 1$, $t(\varepsilon) = \sigma$, $t|_\varepsilon = t$, and $t[s]_\varepsilon = s$. For every $1 \leq i \leq k$ and $v \in pos(t_i)$, $t(iv) = t_i(v)$, $t|_{iv} = t_i|_v$, and $t[s]_{iv} = \sigma(t_1, \ldots, t_{i-1}, t_i[s]_v, t_{i+1}, \ldots, t_k)$.

The *size* of a tree t, $size(t)$, is $|pos(t)|$, the cardinality of its position set. The *height* of a tree t, $height(t)$, is 1 if its size is 1; else, $height(t) = 1 + \max(height(t|_i) \mid 1 \leq i \leq rk(t(\varepsilon)))$. The *yield set* of a tree t, $ydset(t)$, is the set of labels of its leaves, thus $ydset(t) = \{t(v) \mid v \in leaves(t)\}$.

Example 2.1.1 Let $\Sigma = \{\alpha^{(0)}, \beta^{(1)}, \gamma^{(2)}, \eta^{(2)}, \sigma^{(3)}\}$. Let $A = \{z\}$. Let $t = \sigma(\gamma(\alpha, \beta(z)), \eta(z, \alpha), \alpha)$. Then, $t \in T_\Sigma(A)$, $pos(t) = \{\varepsilon, 1, 2, 3, 1.1, 1.2, 2.1, 2.2, 1.2.1\}$, $leaves(t) = \{1.1, 1.2.1, 2.1, 2.2, 3\}$. $t(2) = \eta$, and $t|_{1.2} = \beta(z)$. Let $s = \beta(\beta(\alpha))$. Then, $t[2.1]_s = \sigma(\gamma(\alpha, \beta(z)), \eta(\beta(\beta(\alpha)), \alpha), \alpha)$, $size(t) = 9$, $height(t) = 4$, $size(t[2.1]_s) = 11$, $height(t[2.1]_s) = 5$, and $ydset(t) = \{\alpha, z\}$. For greater clarity, t, s, and $t[2.1]_s$ are reproduced in a more "treelike" fashion in Figure 2.1.

2.1.2 Semirings

A *semiring* $(\mathbb{W}, +, \cdot, 0, 1)$ is an algebra consisting of a commutative monoid $(\mathbb{W}, +, 0)$ and a monoid $(\mathbb{W}, \cdot, 1)$ where $\cdot$ distributes over $+$, $0 \neq 1$, and 0 absorbs $\cdot$, that is, $w \cdot 0 = 0 \cdot w = 0$ for any $w \in \mathbb{W}$. A semiring is *commutative* if $\cdot$ is commutative. A semiring is *complete* if there is an additional operator $\bigoplus$ that extends the addition operator $+$ such that for any countable index set I and family $(w_i)_{i \in I}$ of elements of $\mathbb{W}$, $\bigoplus$ calculates the possibly infinite summation of $(w_i)_{i \in I}$. We write $\bigoplus_{i \in I} w_i$ rather than $\bigoplus (w_i)_{i \in I}$. $\bigoplus$ has the following properties:

$\bigoplus$ extends $+$: $\bigoplus_{i \in I} w_i = 0$ if $|I| = 0$ and

$\bigoplus_{i \in I} w_i = w_i$ if $|I| = 1$.

$\bigoplus$ is associative and commutative: $\bigoplus_{i \in I} w_i = \bigoplus_{j \in J} \left(\bigoplus_{i \in I_j} w_i \right)$

for any disjoint partition $I = \bigcup_{j \in J} I_j$.

$\cdot$ distributes over $\bigoplus$ from both sides: $w \cdot \left(\bigoplus_{i \in I} w_i \right) = \bigoplus_{i \in I} (w \cdot w_i)$ and

$\left(\bigoplus_{i \in I} w_i \right) \cdot w = \bigoplus_{i \in I} (w_i \cdot w)$

for any $w \in \mathbb{W}$.

A complete semiring can be augmented with the unary closure operator $*$, defined by $w^* = \bigoplus_{i=0}^{\infty} w^i$ for any $w \in \mathbb{W}$, where $w^0 = 1$ and $w^{n+1} = w^n \cdot w$. Unless otherwise noted, henceforth semirings are presumed to be commutative and complete. We refer to a semiring by its carrier set $\mathbb{W}$. Some common commutative and complete semirings, are the:

- *Boolean semiring*: $(\{0, 1\}, \vee, \wedge, 0, 1)$

- *probability semiring*: $(\mathbb{R}_+ \cup \{+\infty\}, +, \cdot, 0, 1)$
- *tropical semiring*: $(\mathbb{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$

In the Boolean semiring, $0^* = 1^* = 1$. In the tropical semiring, $w^* = 0$ for $w \in \mathbb{R}_+$ and $w^* = -\infty$ otherwise. In the probability semiring, $w^* = \frac{1}{1-w}$ for $0 \leq w < 1$ and $w^* = +\infty$ otherwise. Further material on semirings may be found in [35, 57, 52].

Example 2.1.2 In the probability semiring, $0.3 + 0.5 = 0.8$ and $0.3 \cdot 0.5 = 0.15$. In the tropical semiring, $0.3 + 0.5 = 0.3$ and $0.3 \cdot 0.5 = 0.8$. In the Boolean semiring, $0 + 1 = 1$ and $0 \cdot 1 = 0$.

2.1.3 Tree series and weighted tree transformations

A *tree series* over Σ and $\mathbb{W}$ is a mapping $L : T_\Sigma \to \mathbb{W}$. For $t \in T_\Sigma$, the element $L(t) \in \mathbb{W}$ is called the *coefficient* of t. The *support* of a tree series L is the set $supp(L) \subseteq T_\Sigma$ where $t \in supp(L)$ iff $L(t)$ is nonzero. A *weighted tree transformation* over Σ, Δ, and $\mathbb{W}$ is a mapping $\tau : T_\Sigma \times T_\Delta \to \mathbb{W}$. The *inverse* of a weighted tree transformation $\tau : T_\Sigma \times T_\Delta \to \mathbb{W}$ is the weighted tree transformation $\tau^{-1} : T_\Delta \times T_\Sigma \to \mathbb{W}$ where, for every $t \in T_\Sigma$ and $s \in T_\Delta$, $\tau^{-1}(s,t) = \tau(t,s)$. The *domain* of a weighted tree transformation $\tau : T_\Sigma \times T_\Delta \to \mathbb{W}$ is the tree series $\mathrm{dom}(\tau) : T_\Sigma \to \mathbb{W}$ where, for every $t \in T_\Sigma$, $\mathrm{dom}(\tau)(t) = \bigoplus_{s \in T_\Delta} \tau(t,s)$. The *range* of τ is the tree series $\mathrm{range}(\tau) : T_\Delta \to \mathbb{W}$ where, for every $s \in T_\Delta$, $\mathrm{range}(\tau)(s) = \bigoplus_{t \in T_\Sigma} \tau(t,s)$. The *identity* of a tree series $L : T_\Sigma \to \mathbb{W}$ is the weighted tree transformation $\imath_L : T_\Sigma \times T_\Sigma \to \mathbb{W}$ where, for every $s, t \in T_\Sigma$, $\imath_L(s,t) = L(s)$ if $s = t$ and 0 otherwise. The *composition* of a weighted tree transformation $\tau : T_\Sigma \times T_\Delta \to \mathbb{W}$

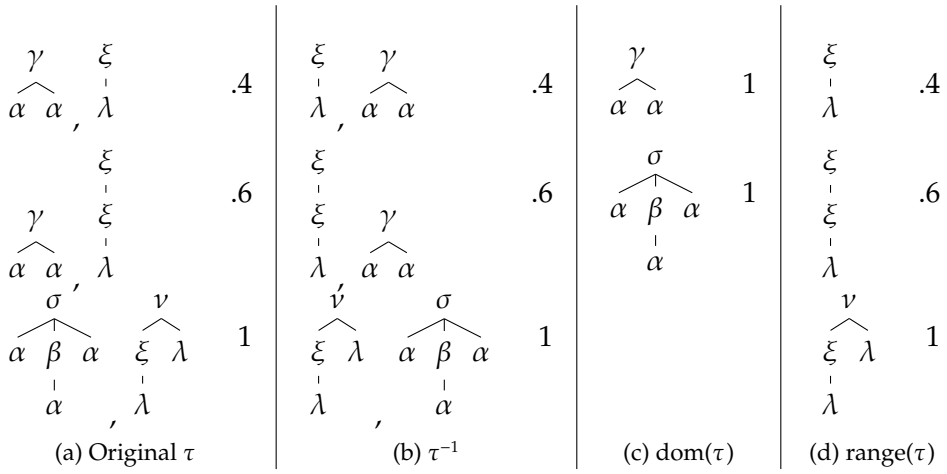


Figure 2.2: Non-zero entries of weighted tree transformations and tree series of Example 2.1.3. For each row, the element(s) on the left maps to the value on the right, and all other elements map to 0.

and a weighted tree transformation $\mu : T_\Delta \times T_\Gamma \to \mathbb{W}$ is the weighted tree transformation $\tau;\mu : T_\Sigma \times T_\Gamma \to \mathbb{W}$ where for every $t \in T_\Sigma$ and $u \in T_\Gamma$, $\tau;\mu(t,u) = \bigoplus_{s \in T_\Delta} \tau(t,s) \cdot \mu(s,u)$.

Example 2.1.3 Let Σ be the ranked alphabet defined in Example 2.1.1. Let $\mathbb{W}$ be the probability semiring. Let $L : T_\Sigma \to \mathbb{W}$ be a tree series such that $L(\gamma(\alpha,\alpha)) = .3$, $L(\sigma(\alpha,\beta(\alpha),\alpha)) = .5$, $L(\alpha) = .2$ and $L(t) = 0$ for all other $t \in T_\Sigma$. Then, $supp(L) = \{\gamma(\alpha,\alpha),\ \sigma(\alpha,\beta(\alpha),\alpha),\ \alpha\}$ and $\iota_L : T_\Sigma \times T_\Sigma \to \mathbb{W}$ is a weighted tree transformation such that $\iota_L(\gamma(\alpha,\alpha),\gamma(\alpha,\alpha)) = .3$, $\iota_L(\sigma(\alpha,\beta(\alpha),\alpha),\sigma(\alpha,\beta(\alpha),\alpha)) = .5$, $\iota_L(\alpha,\alpha) = .2$, and $\iota_L(t,s) = 0$ for all other $(t,s) \in T_\Sigma \times T_\Sigma$.

Let $\Delta = \{\lambda^{(0)}, \xi^{(1)}, \nu^{(2)}\}$ be a ranked alphabet. Let $\tau : T_\Sigma \times T_\Delta \to \mathbb{W}$ be a weighted tree transformation such that $\tau(\gamma(\alpha,\alpha),\xi(\lambda)) = .4$, $\tau(\gamma(\alpha,\alpha),\xi(\xi(\lambda))) = .6$, $\tau(\sigma(\alpha,\beta(\alpha),\alpha),\nu(\xi(\lambda),\lambda)) = 1$, and $\tau(t,s) = 0$ for all other $(t,s) \in T_\Sigma \times T_\Delta$. Then, the non-zero members of the weighted tree transformation $\tau^{-1} : T_\Delta \times T_\Sigma \to \mathbb{W}$ and tree series $\mathrm{dom}(\tau) : T_\Sigma \to \mathbb{W}$ and $\mathrm{range}(\tau) : T_\Delta \to \mathbb{W}$ are those presented in Figure 2.2.

2.1.4 Substitution

Let A and B be sets. Let $\varphi : A \to T_\Sigma(B)$ be a mapping. φ may be extended to the mapping $\overline{\varphi} : T_\Sigma(A) \to T_\Sigma(B)$ such that for $a \in A$, $\overline{\varphi}(a) = \varphi(a)$ and for $k \geq 0$, $\sigma \in \Sigma^{(k)}$, and $t_1, \ldots, t_k \in T_\Sigma(A)$, $\overline{\varphi}(\sigma(t_1, \ldots, t_k)) = \sigma(\overline{\varphi}(t_1), \ldots, \overline{\varphi}(t_k))$. We indicate such extensions by describing φ as a *substitution mapping* and then abuse notation, conflating, e.g., φ and $\overline{\varphi}$ both to φ where there is no confusion.

Example 2.1.4 Let Σ be the ranked alphabet defined in Example 2.1.1. Let $A = \{z, y\}$ and $B = \{v, w\}$. Let $\varphi : A \to T_\Sigma(B)$ be a substitution mapping such that $\varphi(z) = v$ and $\varphi(y) = w$. Let $t = \eta(\gamma(\alpha, z), \beta(y))$. Then, $\varphi(t) = \eta(\gamma(\alpha, v), \beta(w))$.

2.2 Weighted regular tree grammars

In much of the text that follows we refer to previous work, but note that our constructions are somewhat different. We will cite that work, note differences, and note the implications of these differences as we come to them. The algorithms we present are intended to be close to pseudocode and directly implementable; exceptions are noted.

Definition 2.2.1 (cf. Alexandrakis and Bozapalidis [1]) A weighted regular tree grammar (wrtg) over semiring $\mathbb{W}$ is a 4-tuple $G = (N, \Sigma, P, n_0)$ where:

1. N is a finite set of *nonterminals*, with $n_0 \in N$ the start nonterminal
2. Σ is the input ranked alphabet.
3. P is a tuple (P', π), where P' is a finite set of *productions*, each production p of the form $n \to u$, $n \in N$, $u \in T_\Sigma(N)$, and $\pi : P' \to \mathbb{W}$ is a weight function of the

productions. Within these constraints we may (and usually do) refer to P as a finite set of weighted productions, each production p of the form $n \xrightarrow{\pi(p)} u$. We denote subsets of P as follows: $P_n = \{p \in P \mid p \text{ is of the form } n \to u\}$. We extend all definitions of operations on trees from Section 2.1.1 to productions such that, e.g., $size(p) = size(u)$. We associate P with G, such that, e.g., $p \in G$ is interpreted to mean $p \in P$.

Unlike the definition by Alexandrakis and Bozapalidis [1] we in general allow *chain productions* in a wrtg, that is, productions of the form $n_i \xrightarrow{w} n_j$, where $n_i, n_j \in N$.

For wrtg $G = (N, \Sigma, P, n_0)$, s, t, $u \in T_\Sigma(N)$, $n \in N$, and $p \in P$ of the form $n \xrightarrow{w} u \in P$, we obtain a *derivation step* from s to t by replacing some leaf nonterminal in s labeled n with u. Formally, $s \Rightarrow_G^p t$ if there exists some $v \in pos(s)$ such that $s(v) = n$ and $s[u]_v = t$. We say this derivation step is *leftmost* if, for all $v' \in leaves(s)$ where $v' < v$, $s(v') \in \Sigma$. Except where noted and needed, we henceforth assume all derivation steps are leftmost and drop the subscript G. If, for some $m \in \mathbb{N}$, $p_i \in P$, and $t_i \in T_\Sigma(N)$ for all $1 \leq i \leq m$, $n_0 \Rightarrow^{p_1} t_1 \ldots \Rightarrow^{p_m} t_m$, we say the sequence $d = (p_1, \ldots, p_m)$ is a *derivation* of t_m in G and that $n_0 \Rightarrow^* t_m$. The weight of d is $wt(d) = \pi(p_1) \cdot \ldots \cdot \pi(p_m)$, the product of the weights of all occurrences of its productions. We may loosen the definition of a derivation and speak of, for example, *a derivation from n using* P', where $P' \subseteq P$, or assert that this derivation exists by saying that $n \Rightarrow^* t_m$ *using* P'. In such cases one may imagine this to be equivalent to a derivation in some wrtg $G' = (N, \Sigma, P', n)$.

The tree series represented by G is denoted L_G. For $t \in T_\Sigma$ and $n \in N$, the tree series $L_G(t)_n$ is defined as follows:

$$L_G(t)_n = \bigoplus_{\text{derivation } d \text{ of } t \text{ from } n \text{ in } G} wt(d)$$

.

Then $L_G(t) = L_G(t)_{n_0}$. Note that this tree series is well defined, even though this summation may be infinite (since chain productions are allowed), because $\mathbb{W}$ is presumed complete. We call a tree series L *recognizable* if there is a wrtg G such that $L_G = L$; in such cases we then call G a *wrtg representing* L_G. Two wrtgs G_1 and G_2 are *equivalent* if $L_{G_1} = L_{G_2}$.

Example 2.2.2 Figure 2.3 depicts P_1 for a wrtg $G_1 = (N_1, \Sigma, P_1, n_s)$ over the probability semiring with production id numbers. N_1 and Σ may be inferred from P_1. Note that production 12 is a chain production. A derivation of the tree S(NP(some students) VP(eat NP(red meat))) is (1, 2, 7, 5, 8, 11, 13, 15) and the weight of this derivation is .00084.

2.2.1 Normal form

A wrtg G is in *normal form* if each production $p \in P$ is in normal form. A production p of the form $n \xrightarrow{w} u$ is in normal form if u has one of the following forms:

1. $u \in \Sigma^{(0)}$
2. $u \in N$
3. $u = \sigma(n_1, \ldots, n_k)$ where $k \geq 1$, $\sigma \in \Sigma^{(k)}$, and $n_1, \ldots, n_k \in N$.

1. $n_{\text{S}} \xrightarrow{1} \text{S}\ (n_{\text{NP-SUBJ}}\ n_{\text{VP}})$

2. $n_{\text{NP-SUBJ}} \xrightarrow{.4} \text{NP}\ (n_{\text{DT}}\ n_{\text{NNS-SUBJ}})$

3. $n_{\text{NP-SUBJ}} \xrightarrow{.6} \text{NP}\ (n_{\text{NNS-SUBJ}})$

4. $n_{\text{NNS-SUBJ}} \xrightarrow{.7} \text{dogs}$

5. $n_{\text{NNS-SUBJ}} \xrightarrow{.3} \text{students}$

6. $n_{\text{DT}} \xrightarrow{.8} \text{the}$

7. $n_{\text{DT}} \xrightarrow{.2} \text{some}$

8. $n_{\text{VP}} \xrightarrow{.7} \text{VP}\ (\text{eat}\ n_{\text{NP-OBJ}})$

9. $n_{\text{VP}} \xrightarrow{.25} \text{VP}\ (\text{chase}\ n_{\text{NP-OBJ}})$

10. $n_{\text{VP}} \xrightarrow{.05} \text{VP}\ (\text{lie})$

11. $n_{\text{NP-OBJ}} \xrightarrow{.25} \text{NP}\ (n_{\text{JJ}}\ n_{\text{NP-OBJ}})$

12. $n_{\text{NP-OBJ}} \xrightarrow{.75} n_{\text{NNS-OBJ}}$

13. $n_{\text{JJ}} \xrightarrow{.4} \text{red}$

14. $n_{\text{JJ}} \xrightarrow{.6} \text{smelly}$

15. $n_{\text{NNS-OBJ}} \xrightarrow{.5} \text{meat}$

16. $n_{\text{NNS-OBJ}} \xrightarrow{.5} \text{cars}$

Figure 2.3: Production set P_1 from example wrtg G_1 used in Examples 2.2.2, 2.2.3, 2.2.4, and 2.2.7.

17. $n_{\text{VP}} \xrightarrow{.7} \text{VP}\ (n_1\ n_{\text{NP-OBJ}})$

18. $n_{\text{VP}} \xrightarrow{.25} \text{VP}\ (n_2\ n_{\text{NP-OBJ}})$

19. $n_{\text{VP}} \xrightarrow{.05} \text{VP}\ (n_3)$

20. $n_1 \xrightarrow{1} \text{eat}$

21. $n_2 \xrightarrow{1} \text{chase}$

22. $n_3 \xrightarrow{1} \text{lie}$

Figure 2.4: Normal-form productions inserted in P_1 to replace productions 8, 9, and 10 of Figure 2.3 in normalization of G_1, as described in Example 2.2.3.

23. $n_{\text{NP-OBJ}} \xrightarrow{.375}$ meat

24. $n_{\text{NP-OBJ}} \xrightarrow{.375}$ cars

Figure 2.5: Productions inserted in P_1 to compensate for the removal of chain production 12 of Figure 2.3 in chain production removal of G_1, as described in Example 2.2.4.

For every wrtg G we can form the wrtg G' such that G and G' are equivalent and G' is in normal form. This is achieved by Algorithm 1, which follows the first half of the construction in Prop. 1.2 of Alexandrakis and Bozapalidis [1] and preserves chain productions.

Example 2.2.3 The wrtg G from Example 2.2.2 is not in normal form. Algorithm 1 produces a normal form equivalent by replacing productions 8, 9, and 10 from Figure 2.3 with the productions in Figure 2.4.

Algorithm 1 NORMAL-FORM

inputs
  wrtg $G_{\text{in}} = (N_{\text{in}}, \Sigma, P_{\text{in}}, n_0)$ over $\mathbb{W}$
outputs
  wrtg $G_{\text{out}} = (N_{\text{out}}, \Sigma, P_{\text{out}}, n_0)$ over $\mathbb{W}$ in normal form such that $L_{G_{\text{in}}} = L_{G_{\text{out}}}$
complexity
  $O(size(\tilde{p})|P_{\text{in}}|)$, where $\tilde{p}$ is the production of largest size in P_{in}

$N_{\text{out}} \leftarrow N_{\text{in}}$
$P_{\text{out}} \leftarrow \emptyset$
for all $p \in P_{\text{in}}$ **do**
  for all $p' \in \text{NORMALIZE}(p, N_{\text{in}})$ **do**
    Let p' be of the form $n \xrightarrow{w} u$.
    $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{p'\}$
    $N_{\text{out}} \leftarrow N_{\text{out}} \cup \{n\}$
return G_{out}

Algorithm 2 NORMALIZE

inputs
  production p_{in} of the form $n \xrightarrow{w} u$
  nonterminal set N
outputs
  $P = \{p_1, \ldots, p_n\}$, the set of productions in normal form such that $n \Rightarrow^{p_1} \ldots \Rightarrow^{p_n} u$ and $wt(p_1, \ldots, p_n) = w$
complexity
  $O(size(p_{in}))$

$P \leftarrow \emptyset$; $\Psi \leftarrow \{p_{in}\}$
while $\Psi \neq \emptyset$ **do**
  $p \leftarrow$any element of Ψ
  $\Psi \leftarrow \Psi \setminus \{p\}$
  Let p be of the form $n \xrightarrow{w} u$.
  if $u \in \Sigma^{(0)}$ **or** $u \in N$ **then** {already in normal form}
    $P \leftarrow P \cup \{p\}$
  else
    Let u be of the form $\sigma(u_1, \ldots, u_k)$.
    $(n_1, \ldots, n_k) \leftarrow (u_1, \ldots, u_k)$
    for $i = 1$ **to** k **do**
      if $u_i \notin N$ **then**
        $n_i \leftarrow$ new nonterminal n_x
        $\Psi \leftarrow \Psi \cup \{n_x \xrightarrow{1} u_i\}$
    $P \leftarrow P \cup \{n \xrightarrow{w} \sigma(n_1, \ldots, n_k)\}$
return P

2.2.2 Chain production removal

Although we have defined algorithms that take chain productions into account, it is nevertheless sometimes useful, as in the string case [101], to remove chain productions[2] from a wrtg. Although chain productions are very helpful in the design of grammars and may be produced by NLP systems, they delay computation and their presence in a wrtg can make certain algorithms cumbersome. Fortunately, removing chain productions in a wrtg is conceptually equivalent to removing them in a weighted finite-state (string) automaton. Chain production removal for weighted string automata was described in Theorem 3.2 of Ésik and Kuich [41], Theorem 3.2 of Kuich [81], and by Mohri [101]. Algorithm 3 reproduces the chain production removal algorithm described by Mohri [101] but does so in the terminology of wrtgs.

Example 2.2.4 The wrtg G_1 from Example 2.2.2 has chain productions, specifically production 12. Algorithm 3 produces an equivalent wrtg without chain productions by removing production 12 (making productions 15 and 16 no longer useful) and adding the productions in Figure 2.5.

Because Example 2.2.4 does not make significant use of Algorithm 4, we provide a more complicated example of chain production removal.

Example 2.2.5 Consider the wrtg $G_2 = (\{q, r, s\}, \Sigma, P_2, q)$ where Σ is defined in Example 2.1.1 and P_2 is depicted in Figure 2.6a. Algorithm 4 operates on the chain productions to form the map represented in Table 2.6b. This map is then used by Algorithm 3 to produce wrtg $G_3 = (\{q, r, s\}, \Sigma, P_3, q)$ where P_3 is depicted in Figure 2.6c.

[2]For finite-state (string) automata chain productions are often called *epsilon transitions*.

1. $q \xrightarrow{.1} \gamma(q, s)$
2. $q \xrightarrow{.4} \alpha$
3. $q \xrightarrow{.2} r$
4. $q \xrightarrow{.3} s$
5. $r \xrightarrow{.8} \alpha$
6. $r \xrightarrow{.05} q$
7. $r \xrightarrow{.15} s$
8. $s \xrightarrow{.9} \alpha$
9. $s \xrightarrow{.1} s$

(a) Production set P_2 from example wrtg G_2 used in Example 2.2.5 to demonstrate chain production cycle removal.

	q	r	s
q	$1.\overline{01}$	$.\overline{20}$	$.\overline{370}$
r	$.\overline{05}$	$1.\overline{01}$	$.\overline{185}$
s	0	0	$1.\overline{1}$

(b) Closure table for chain production removal of G_2. Rows and columns list source and destination nonterminals, respectively.

1. $q \xrightarrow{.\overline{10}} \gamma(q, s)$
2. $q \xrightarrow{.\overline{89}} \alpha$
3. $s \xrightarrow{1} \alpha$

(c) Production set P_3, obtained by chain production removal of G_2.

Figure 2.6: Illustration of Algorithms 3 and 4, as described in Example 2.2.5. Algorithm 4 builds the table in Figure 2.6b from the productions in Figure 2.6a and then Algorithm 3 uses this table to generate the productions in Figure 2.6c.

2.2.3 Determinization

A normal-form wrtg $G = (N, \Sigma, P, n_0)$ over $\mathbb{W}$ is *deterministic* if, for each $k \in \mathbb{N}$, $\sigma \in \Sigma^{(k)}$, and $n_1, \ldots, n_k \in N^k$ there is at most one production of the form $n \xrightarrow{w} \sigma(n_1, \ldots, n_k)$ in P, where $n_1, \ldots, n_k \in N$.[3] Non-deterministic and deterministic wrtg over the Boolean semiring (which we sometimes refer to as rtg, as they are equivalent to unweighted regular tree grammars) represent the same tree series ([33], Thm. 1.10); thus we may define an algorithm which takes an arbitrary rtg in normal form and produces a language-equivalent deterministic one. The naive algorithm generalizes the classical determinization algorithm for fsas [112]; for each word $\vec{\rho} = \rho_1 \rho_2 \ldots \rho_k$ in $(\mathcal{P}(N))^k$ and $\sigma \in \Sigma^{(k)}$, find all m

[3]Thus we are describing a wrtg equivalent to a *bottom-up* deterministic weighted tree automaton. We do not consider top-down deterministic properties, as top-down deterministic tree automata are strictly weaker than their bottom-up counterparts ([48], Ex. II.2.11).

Algorithm 3 CHAIN-PRODUCTION-REMOVAL

inputs
  wrtg $G_{\text{in}} = (N, \Sigma, P_{\text{in}}, n_0)$ over $\mathbb{W}$
outputs
  wrtg $G_{\text{out}} = (N, \Sigma, P_{\text{out}}, n_0)$ over $\mathbb{W}$ such that $L_{G_{\text{in}}} = L_{G_{\text{out}}}$. Additionally, no $p \in P_{\text{out}}$ is of the form $n_{\text{src}} \xrightarrow{w} n_{\text{dst}}$, where n_{src} and $n_{\text{dst}} \in N$.
complexity
  $O(|N|^3 + |N||P_{\text{in}}|)$

$P_{\text{chain}} \leftarrow \{n \xrightarrow{w} u \in P_{\text{in}} \mid u \in N\}$
Form a mapping $\phi : N \times N \to \mathbb{W}$.
$\phi \leftarrow$ COMPUTE-CLOSURE(N, P_{chain})
Form a mapping $\theta : N \times T_\Sigma(N) \to \mathbb{W}$.
$\overline{P_{\text{chain}}} \leftarrow P_{\text{in}} \setminus P_{\text{chain}}$
for all $n_{\text{dst}} \in N$ **do**
  for all $n_{\text{src}} \in N$ **do**
    for all $n_{\text{dst}} \xrightarrow{w} u \in \overline{P_{\text{chain}}}$ **do**
      $\theta(n_{\text{src}}, u) \leftarrow \theta(n_{\text{src}}, u) + (w \cdot \phi(n_{\text{src}}, n_{\text{dst}}))$
$P_{\text{out}} \leftarrow \{n \xrightarrow{\theta(n,u)} u \mid \theta(n, u) \neq 0\}$
return G_{out}

Algorithm 4 COMPUTE-CLOSURE

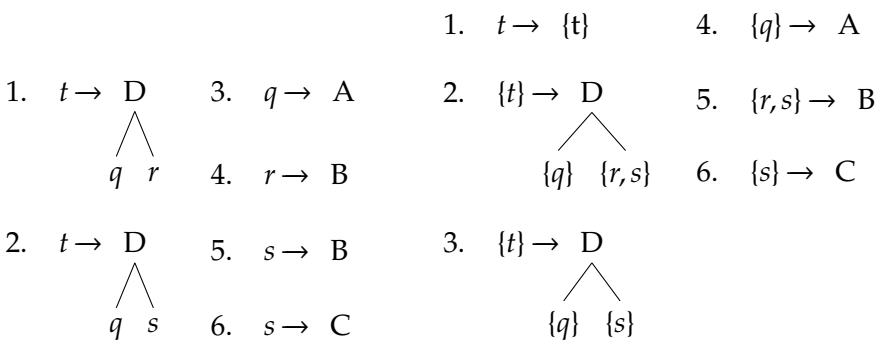
inputs
  nonterminals N
  production set P, where each $p \in P$ is of the form $n_{\text{src}} \xrightarrow{w} n_{\text{dst}}$, n_{src} and $n_{\text{dst}} \in N$
outputs
  mapping $\phi : N \times N \to \mathbb{W}$ such that $\phi(n_{\text{src}}, n_{\text{dst}})$ is the sum of weights of all derivations from n_{src} to n_{dst} using P.
complexity
  $O(|N|^3)$

$\phi(n_{\text{src}}, n_{\text{dst}}) \leftarrow 0$ for each $n_{\text{src}}, n_{\text{dst}} \in N$
for all $n_{\text{src}} \xrightarrow{w} n_{\text{dst}} \in P$ **do**
  $\phi(n_{\text{src}}, n_{\text{dst}}) \leftarrow \phi(n_{\text{src}}, n_{\text{dst}}) + w$
for all $n_{\text{mid}} \in N$ **do**
  for all $n_{\text{src}} \in N, n_{\text{src}} \neq n_{\text{mid}}$ **do**
    for all $n_{\text{dst}} \in N, n_{\text{dst}} \neq n_{\text{mid}}$ **do**
      $\phi(n_{\text{src}}, n_{\text{dst}}) \leftarrow \phi(n_{\text{src}}, n_{\text{dst}}) + (\phi(n_{\text{src}}, n_{\text{mid}}) \cdot \phi(n_{\text{mid}}, n_{\text{mid}})^* \cdot \phi(n_{\text{mid}}, n_{\text{dst}}))$
  for all $n_{\text{src}} \in N, n_{\text{src}} \neq n_{\text{mid}}$ **do**
    $\phi(n_{\text{mid}}, n_{\text{src}}) \leftarrow \phi(n_{\text{mid}}, n_{\text{mid}})^* \cdot \phi(n_{\text{mid}}, n_{\text{src}})$
    $\phi(n_{\text{src}}, n_{\text{mid}}) \leftarrow \phi(n_{\text{src}}, n_{\text{mid}}) \cdot \phi(n_{\text{mid}}, n_{\text{mid}})^*$
  $\phi(n_{\text{mid}}, n_{\text{mid}}) \leftarrow \phi(n_{\text{mid}}, n_{\text{mid}})^*$
return ϕ

Algorithm 5 DETERMINIZE

inputs
  wrtg $G_{\text{in}} = (N, \Sigma, P_{\text{in}}, n_0)$ over Boolean semiring in normal form with no chain productions
outputs
  deterministic wrtg $G_{\text{out}} = (\mathcal{P}(N) \cup \{n_0\}, \Sigma, P_{\text{out}}, n_0)$ over Boolean semiring in normal form such that $L_{G_{\text{in}}} = L_{G_{\text{out}}}$.
complexity
  $O(|P_{\text{in}}| 2^{|N|^{\max_{\sigma \in \Sigma} rk(\sigma)}})$

$P_{\text{out}} \leftarrow \emptyset$
$\Xi \leftarrow \emptyset$ {Seen nonterminals}
$\Psi \leftarrow \emptyset$ {New nonterminals}
for all $\alpha \in \Sigma^{(0)}$ **do**
  $\rho_{dst} \leftarrow \{n \mid n \to \alpha \in P_{\text{in}}\}$
  $\Psi \leftarrow \Psi \cup \{\rho_{dst}\}$
  $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{\rho_{dst} \to \alpha\}$
  if $n_0 \in \rho_{dst}$ **then**
    $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{n_0 \to \rho_{dst}\}$
while $\Psi \neq \emptyset$ **do**
  $\rho_{new} \leftarrow$ any element of Ψ
  $\Xi \leftarrow \Xi \cup \{\rho_{new}\}$
  $\Psi \leftarrow \Psi \setminus \{\rho_{new}\}$
  for all $\sigma^{(k)} \in \Sigma \setminus \Sigma^{(0)}$ **do**
    for all $\vec{\rho} = \rho_1 \dots \rho_k \mid \rho_1 \dots \rho_k \in \Xi^k, \rho_i = \rho_{new}$ for some $1 \leq i \leq k$ **do**
      $\rho_{dst} \leftarrow \{n \mid n \to \sigma(n_1, \dots, n_k) \in P_{\text{in}}, n_1 \in \rho_1, \dots, n_k \in \rho_k\}$
      if $\rho_{dst} \neq \emptyset$ **then**
        if $\rho_{dst} \notin \Xi$ **then**
          $\Psi \leftarrow \Psi \cup \{\rho_{dst}\}$
        $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{\rho_{dst} \to \sigma(\vec{\rho})\}$
        if $n_0 \in \rho_{dst}$ **then**
          $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{n_0 \to \rho_{dst}\}$
return G_{out}



(a) P_4, productions of nondeterministic rtg G_4.

(b) P_5, productions of deterministic rtg G_5 obtained by applying Algorithm 5 to G_4.

Figure 2.7: rtg productions before and after determinization, as described in Example 2.2.6. Note that (a) is not deterministic because productions 4 and 5 have the same right side, while no productions in (b) have the same right side.

productions $p_1, \ldots, p_m$ where, for $1 \leq j \leq m$, $p_j = n_j \to \sigma(n_{j_1}, \ldots, n_{j_k})$ such that for $1 \leq i \leq k$, $n_{j_i} \in \rho_i$. Then, $\{n_j \mid 1 \leq j \leq m\} \to \sigma(\rho_1, \ldots, \rho_k)$ is in the determinized rtg. Additionally, if $n_j = n_0$ for some $1 \leq j \leq m$, $n_0 \to \{n_j \mid 1 \leq j \leq m\}$ is in the determinized rtg. This is frequently overexhaustive, though. Algorithm 5 is more appropriate for actual implementation, as it only bothers to build productions for nonterminals that can be reached. This algorithm first creates the nonterminals used to produce leaves in lines 10–15, then uses those nonterminals to produce more nonterminals in lines 20–26. until no more can be produced. To ensure a single start nonterminal, chain productions are added from a new unique start nonterminal in lines 15 and 28.

Example 2.2.6 Consider the rtg $G_4 = (N_4, \Sigma, P_4, t)$ where $N_4 = \{t, q, r, s\}$, $\Sigma = \{A^{(0)}, B^{(0)}, C^{(0)}, D^{(2)}\}$, and P_4 is depicted in Figure 2.7a. The result of Algorithm 5 on input G_4 is $G_5 = (\mathcal{P}(N_4), \Sigma, P_5, t)$, where P_5 is depicted in Figure 2.7b.

We discuss our contribution to algorithms for determinization of a wider class of wrtg in Chapter 3.

2.2.4 Intersection

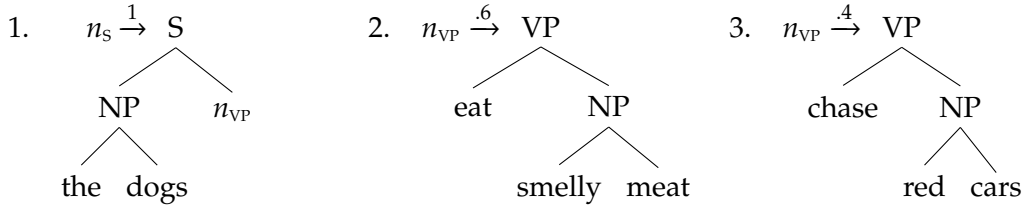
Algorithm 6 INTERSECT

1: **inputs**
2: wrtg $G_A = (N_A, \Sigma, P_A, n_{A_0})$ over $\mathbb{W}$ in normal form with no chain productions
3: wrtg $G_B = (N_B, \Sigma, P_B, n_{B_0})$ over $\mathbb{W}$ in normal form with no chain productions
4: **outputs**
5: wrtg $G_C = ((N_A \times N_B), \Sigma, P_C, (n_{A_0}, n_{B_0}))$ over $\mathbb{W}$ such that for every $t \in T_\Sigma$, $L_{G_C}(t) = L_{G_A}(t) \cdot L_{G_B}(t)$
6: **complexity**
7: $O(|P_A||P_B|)$

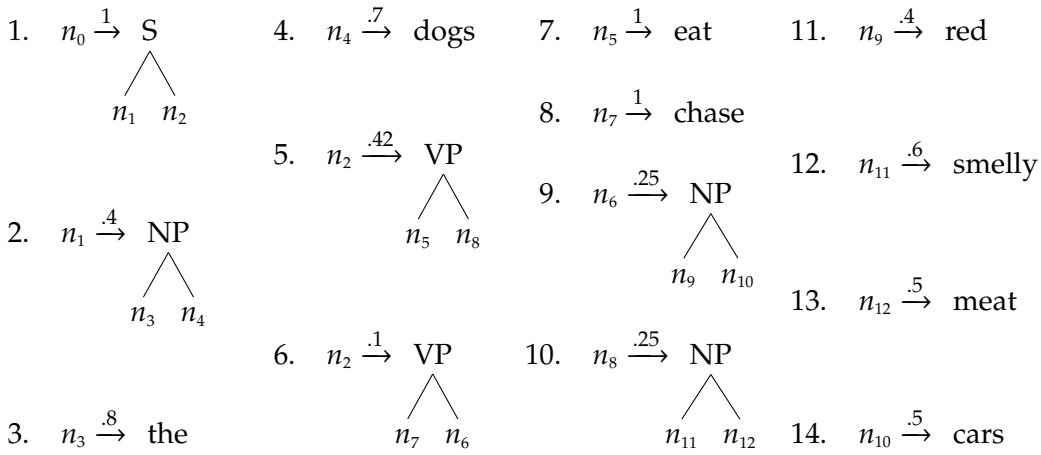
8: $P_C \leftarrow \emptyset$
9: **for all** $(n_A, n_B) \in N_A \times N_B$ **do**
10: **for all** $\sigma^{(k)} \in \Sigma$ **do**
11: **for all** p_A of the form $n_A \xrightarrow{w_A} \sigma(n_{A_1}, \ldots, n_{A_k}) \in P_A$ **do**
12: **for all** p_B of the form $n_B \xrightarrow{w_B} \sigma(n_{B_1}, \ldots, n_{B_k}) \in P_B$ **do**
13: $P_C \leftarrow P_C \cup (n_A, n_B) \xrightarrow{w_A \cdot w_B} \sigma((n_{A_1}, n_{B_1}), \ldots, (n_{A_k}, n_{B_k}))$
14: **return** G_C

It is frequently useful to find the weighted intersection between two tree series L_A and L_B. This intersection is of course also a tree series L_C where for every $t \in T_\Sigma$, $L_C(t) = L_A(t) \cdot L_B(t)$. For the case of recognizable tree series, if we consider two chain production-free, normal-form wrtgs G_A and G_B representing these tree series, then we would like to find a third wrtg G_C such that $L_{G_C}(t) = L_{G_A}(t) \cdot L_{G_B}(t)$. Algorithm 6 is a very simple algorithm that finds this intersection wrtg.[4] Note that in practice, rather than iterating over all members of $N_A \times N_B$ as the algorithm specifies at line 9, an actual implementation should begin by considering (n_{A_0}, n_{B_0}), and then proceed by considering nonterminals that appear in the right sides of newly generated productions as they are discovered, at line 13. We have omitted this detail from the presentation of the algorithm

[4]This algorithm is derived from a composition of identity transducers; see Section 2.3.3.



(a) P_6, productions for the wrtg G_6 used in Example 2.2.7.



(b) P_7, productions for the result of intersecting normal-form, chain-production free transformations of Figures 2.3 and 2.8a.

Figure 2.8: Productions for a wrtg and intersection result, as described in Example 2.2.7.

for clarity's sake and will continue to do so throughout the remainder of this work, but will indicate when such an optimization is appropriate in running text.

Example 2.2.7 Consider the wrtg $G_1 = (N_1, \Sigma, P_1, n_S)$ described in Example 2.2.2 with P_1 depicted in Figure 2.3 and the wrtg $G_6 = (\{n_S, n_{VP}\}, \Sigma, P_6, n_S)$, with P_6 depicted in Figure 2.8a. Normal form and chain production removal of G_1 is described in Examples 2.2.3 and 2.2.4; similar transformations of G_6 are left as an exercise. The result of intersecting these transformed wrtgs (after nonterminals have been renamed) is $G_7 = (\{n_i \mid 0 \leq i \leq 12\}, \Sigma, P_7, n_0)$, where P_7 is depicted in Figure 2.8b.

2.2.5 K-best

Algorithms for determining the k highest weighted paths in a hypergraph have been described by Huang and Chiang [59] and Pauls and Klein [108] and are easily adaptable to the wrtg domain. We refer the reader to those works, which contain clearly presented algorithms.

2.3 Weighted top-down tree transducers

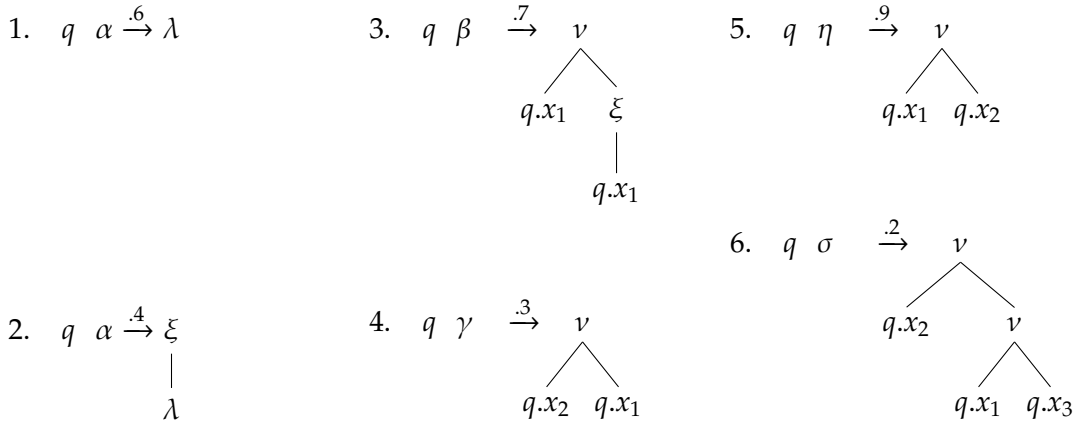
Definition 2.3.1 (cf. Sec. 5.3 of Fülöp and Vogler [45]) A weighted top-down tree transducer (wtt) is a 5-tuple $M = (Q, \Sigma, \Delta, R, q_0)$ where:

1. Q is a finite set of *states*, with $q_0 \in Q$ the start state,
2. Σ and Δ are the input and output ranked alphabets,
3. R is a tuple (R', π) where R' is a finite set of *rules*, each rule r of the form $q.\sigma \to u$ for $q \in Q$, $\sigma \in \Sigma^{(k)}$, and $u \in T_\Delta(Q \times X_k)$, and $\pi : R' \to \mathbb{W}$ is a weight function of the rules. We frequently refer to R as a finite set of weighted rules, each rule r of the form $q.\sigma \xrightarrow{\pi(r)} u$. We denote subsets of R as follows: $R_{q,\sigma} = \{r \in R \mid r \text{ is of the form } q.\sigma \to u\}$. We extend all definitions of operations on trees from Section 2.1.1 to rules, such that, e.g., $size(r) = size(u)$. We associate R with M, such that, e.g., $r \in M$ is interpreted to mean $r \in R$.

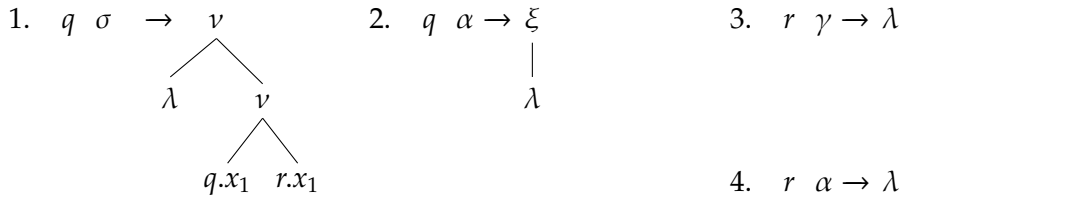
The *multiplicity* of a variable x_i in a rule r of the form $q.\sigma \xrightarrow{w} u$, denoted $\text{mult}(r, i)$, is the number of times x_i appears in u. A wtt is *linear* if for each rule r of the form $q.\sigma \xrightarrow{w} u$ where $\sigma \in \Sigma^{(k)}$ and $k \geq 1$, $\max_{i=1}^{k} \text{mult}(r, i) = 1$. If, for each rule r, $\min_{i=1}^{k} \text{mult}(r, i) = 1$,

the wtt is *nondeleting*. We denote the class of all wtt as wT and add the letters L and N to signify intersections of the classes of linear and nondeleting wtt, respectively. We also remove the letter "w" to signify those wtt over the Boolean semiring. For example, wNT is the class of nondeleting wtt over arbitrary semiring, and LNT is the class of nondeleting and linear wtt over the Boolean semiring. We use class names as a generic descriptor of individual wtts. For example, the phrase "a wLT M" means "a wtt M of class wLT." We also define the following properties of wtt used in special circumstances (and not part of the "core"): a wtt is *deterministic* if, for each $q \in Q$ and $\sigma \in \Sigma$ there is at most one rule of the form $q.\sigma \xrightarrow{w} u$; it is *total* if there is at least one such rule. It is *height-1* if each rule is of the form $q.\sigma \xrightarrow{w} \delta(d_1, \ldots, d_k)$ where $\delta \in \Delta^{(k)}$ and $d_i \in Q \times X$ for $1 \leq i \leq k$.

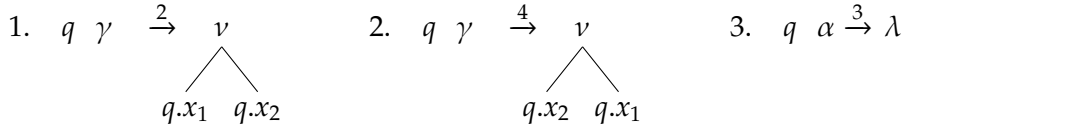
For wtt $M = (Q, \Sigma, \Delta, R, q_0)$, $s, t \in T_\Delta(Q \times T_\Sigma)$, $q \in Q$, and $r \in R$ of the form $q.\sigma \xrightarrow{w} u$, we obtain a *derivation step* from s to t by replacing some leaf of s labeled with q and a tree beginning with σ by a transformation of the right side of r, where each instance of a variable has been replaced by a corresponding child of the σ-headed tree. Formally, $s \Rightarrow_M^r t$ if there exists some $v \in pos(s)$ such that $s(v) = (q, \sigma(s_1, \ldots, s_k))$ and $s[\varphi(u)]_v = t$, where φ is a substitution mapping $Q \times X \to T_\Delta(Q \times T_\Sigma)$, such that $\varphi((q', x_i)) = (q', s_i)$ for all $q' \in Q, 1 \leq i \leq k$. We say this derivation step is *leftmost* if, for all $v' \in leaves(s)$ where $v' < v$, $s(v') \in \Delta$. Except where noted and needed, we henceforth assume all derivation steps are leftmost and drop the subscript M. If, for some $q \in Q$, $s \in T_\Sigma$, $m \in \mathbb{N}$, $r_i \in R$, and $t_i \in T_\Delta(Q \times T_\Sigma)$ for all $1 \leq i \leq m$, $(q, s) \Rightarrow^{r_1} t_1 \ldots \Rightarrow^{r_m} t_m$, we say the sequence $(r_1, \ldots, r_m)$ is a *derivation* of (s, t_m) in M from q. The weight of a derivation d, $wt(d)$, is the product of the weights of all occurrences of its rules.



(a) R_1: total wNT



(b) R_2: deterministic T



(c) R_3: height-1 wLNT

Figure 2.9: Rule sets for three wtts.

The weighted tree transformation represented by M is the mapping $\tau_M : T_\Sigma \times T_\Delta \times Q \to \mathbb{W}$ defined, for all $s \in T_\Sigma, t \in T_\Delta$, and $q \in Q$, as follows:

$$\tau_M(s,t)_q = \bigoplus_{\text{derivation } d \text{ of } (s,t) \text{ in } M \text{ from } q} wt(d).$$

If $q = q_0$, we may leave off "from q_0" in the definition of a derivation and use $\tau_M(s,t)$ as shorthand for $\tau_M(s,t)_{q_0}$.

Example 2.3.2 Let Σ and Δ be the ranked alphabets defined in Examples 2.1.1 and 2.1.3. For reference, $\Sigma = \{\alpha^{(0)}, \beta^{(1)}, \gamma^{(2)}, \eta^{(2)}, \sigma^{(3)}\}$ and $\Delta = \{\lambda^{(0)}, \xi^{(1)}, \nu^{(2)}\}$. Let $M_1 = (\{q\}, \Sigma, \Delta, R_1, q)$, $M_2 = (\{q, r\}, \Sigma, \Delta, R_2, q)$, and $M_3 = (\{q\}, \Sigma, \Delta, R_3, q)$ be wtts, with R_1, R_2, and R_3 depicted in Figures 2.9a, 2.9b, and 2.9c, respectively. M_1 is a total wNT, M_2 is a deterministic T, and M_3 is a height-1 wLNT. The sequence (3, 1, 2) is a derivation of $(\beta(\alpha), \nu(\lambda, \xi(\xi(\lambda))))$ in M_1 and if M_1 is taken to be over the probability semiring, the weight of the derivation is .168. The value of $\tau_{M_3}(\gamma(\alpha, \alpha), \nu(\lambda, \lambda))$ is 8 if M_3 is taken to be over the tropical semiring, but is 54 if taken over the probability semiring.

We now define weighted extended top-down tree transducers [56], a generalization of wtt where the left-hand side may contain an arbitrary pattern. This formalism is frequently more useful than "traditional" wtt in NLP applications as it captures at least the same set of weighted tree transformations as the commonly used synchronous tree substitution grammars [90, 119].[5] We can immediately see problems with wtt before even considering real-world applications. Consider that there is no wtt that captures the (finite!) transformation depicted in Figure 2.2a.

[5] In fact, xLNT (which are defined further down the page) have precisely the same power as STSG , if STSG are given states [90].

Definition 2.3.3 (cf. Def. 1 of Maletti [90]) A weighted extended top-down tree transducer (wxtt) is a 5-tuple $M = (Q, \Sigma, \Delta, R, q_0)$ where:

1. Q, Σ, and Δ are defined as for wtt.

2. R is a tuple (R', π). R' is a finite set of *rules*, each rule r of the form $q.y \xrightarrow{w} u$ for $q \in Q$, $y \in T_\Sigma(X)$, and $u \in T_\Delta(Q \times X)$. We further require that y is linear in X, i.e., no variable $x \in X$ appears more than once in y, and that each variable appearing in u is also in y. $\pi : R' \to \mathbb{W}$ is a weight function of the rules. As for wrtgs and wtts, we refer to R as a finite set of weighted rules, each rule r of the form $q.y \xrightarrow{\pi(r)} u$.

For wxtt $M = (Q, \Sigma, \Delta, R, q_0)$, $s, t \in T_\Delta(Q \times T_\Sigma)$, $q \in Q$, and $r \in R$ of the form $q.y \xrightarrow{w} u$, we obtain a *derivation step* from s to t by replacing some leaf of s labeled with q and a tree matching y, by a transformation of u, where each instance of a variable has been replaced by a corresponding subtree of the y-matching tree. Formally, $s \Rightarrow_M^r t$ if there is a position $v \in pos(s)$, a substitution mapping $\varphi : X \to T_\Sigma$, and a rule $q.y \xrightarrow{w} u \in R$ such that $s(v) = (q, \varphi(y))$ and $t = s[\varphi'(u)]_v$, where φ' is a substitution mapping $Q \times X \to T_\Delta(Q \times T_\Sigma)$ defined such that $\varphi'(q', x) = (q', \varphi(x))$ for all $q' \in Q$ and $x \in X$. We define *leftmost*, *derivation*, and *wt* for wxtt as we do for wtt. We also define the weighted tree transformation $\tau_M(s, t)_q$ for all $s \in T_\Sigma$, $t \in T_\Delta$, and $q \in Q$ as we do for wtt, but additionally note that the assumption that $\mathbb{W}$ is complete ensures weighted tree transformation is well defined for wxtt, even though the summation of derivations may be infinite due to "chain" rules such as $q.x \xrightarrow{w} q.x$.

We extend the properties *linear* and *nondeleting* to wxtt. We use the letter "x" to denote classes of wxtt and thus incorporate them into our class naming convention. Thus, xLT

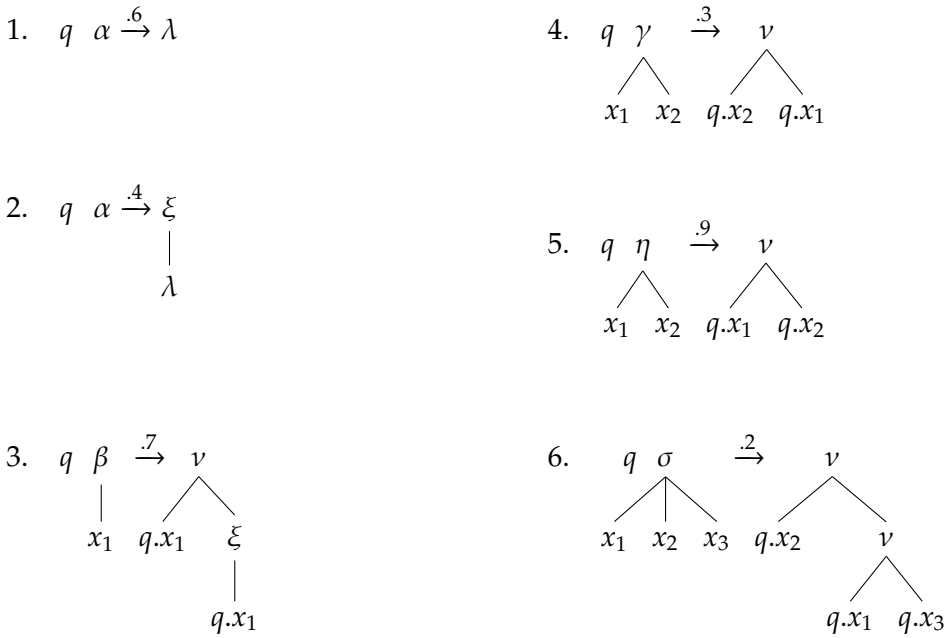
is the class of linear wxtt over the Boolean semiring, and wxLNT is the class of linear and nondeleting wxtt over an arbitrary semiring. A wxtt is *ϵ-free* if there is no rule $q.x \xrightarrow{w} u \in R$ where $x \in X$.

Example 2.3.4 M_1, M_2, and M_3 from Example 2.3.2 are wtts, so they are also wxtts. However, the form of their rules when presented as wxtts is slightly different; Figure 2.10a demonstrates this for M_1 and similar "transformations" for M_2 and M_3 should be fairly obvious. $M_4 = (\{q_1, q_2, q_3\}, \Sigma, \Delta, R_4, q_1)$, where R_4 is depicted in Figure 2.10b, is a wxLNT over the probability semiring that recognizes τ from Figure 2.2a.

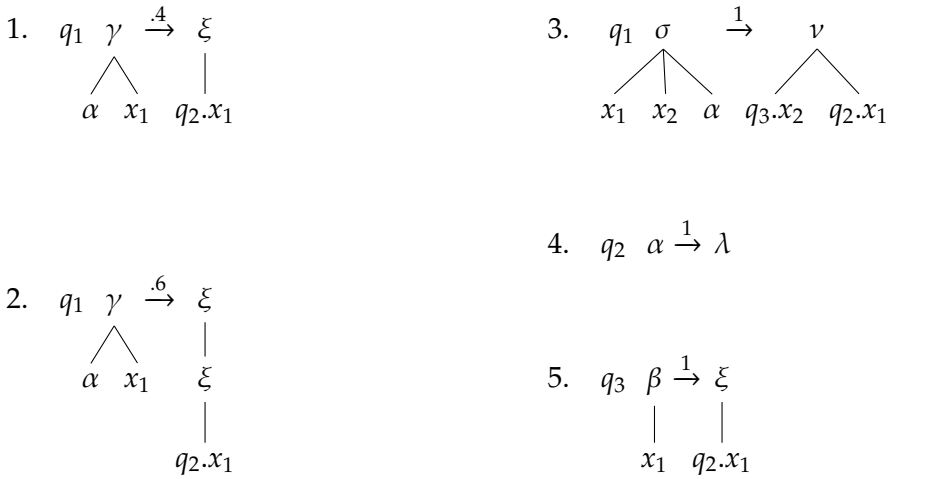
When certain properties apply to every weighted tree transformation represented by some class of wtt, we elevate those properties to the class itself. For example, we can say that T has recognizable domain, because all weighted tree transformations represented by a wtt of class T have recognizable domain ([48], cor. IV.3.17). We now discuss some algorithms on wtts and wxtts.

2.3.1 Projection

The *projection* of a transducer M is the construction of a syntactic structure that represents either $\mathrm{dom}(\tau_M)$ (called the *domain projection*) or $\mathrm{range}(\tau_M)$ (the *range projection*). Since the only syntactic structures under discussion here capture recognizable tree series, projection is only possible once recognizability is ensured. As we just mentioned, T has recognizable domain ([48], cor. IV.3.17). Algorithm 7 is a "folklore" algorithm that obtains the domain projection from a wtt of class T. As in the case of Algorithm 6, an



(a) Recasting of R_1 from Figure 2.9a as wxtt rules.



(b) R_4, for representing τ from Figure 2.2a.

Figure 2.10: Rule sets for wxtts presented in Example 2.3.4.

implementation of this algorithm should consider nonterminals as they are encountered, and not iterate over all possible nonterminals.

wNT does not have recognizable domain [91], but wLT does [43]. A much simpler algorithm, Algorithm 8, obtains domain projection from a wtt of class wLT. The implementation optimization previously discussed for Algorithms 6 and 7 applies here as well. The key difference between the algorithms, aside from the preservation of weights, is that the linearity constraint for Algorithm 8 does not require the "merging" of rules done in lines 17–22 of Algorithm 7.

Algorithm 7 BOOL-DOM-PROJ

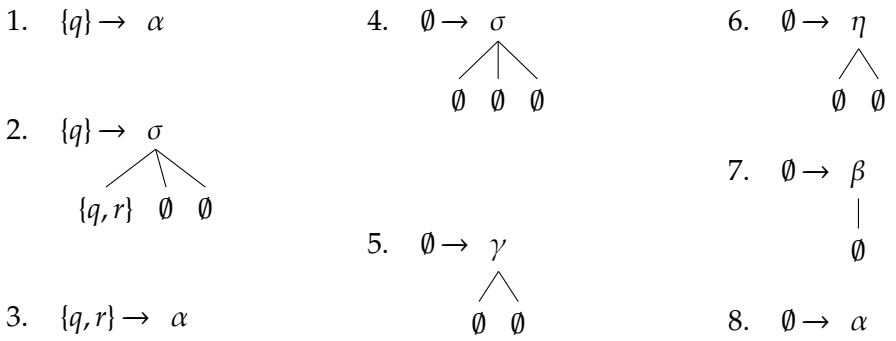
inputs
  wtt $M = (Q, \Sigma, \Delta, R, q_0)$ over Boolean semiring
outputs
  wrtg $G = (N, \Sigma, P, n_0)$ over Boolean semiring such that $L_G = \mathrm{dom}(\tau_M)$
complexity
  $O((\frac{2|R|}{|Q|})^{|Q|})$

$N \leftarrow \mathcal{P}(Q)$
$n_0 \leftarrow \{q_0\}$
$P \leftarrow \emptyset$
for all $n \in N$ **do**
  if $n = \emptyset$ **then**
    for all $\sigma \in \Sigma$ with rank k **do**
      $P \leftarrow P \cup \{\emptyset \rightarrow \sigma(\overbrace{\emptyset, \ldots, \emptyset}^{k})\}$
  else
    $n \in \mathcal{P}(Q)$ is of the form $\{q_1, \ldots, q_m\}$ for some $m \leq |Q|$.
    for all $\sigma^{(k)} \in \Sigma$ **do**
      for all $(r_1, \ldots, r_m) \in R_{q_1,\sigma} \times \ldots \times R_{q_m,\sigma}$ **do**
        Let ϕ be a mapping $X_k \rightarrow N$ where $\phi(x_i) = \emptyset$ for all $1 \leq i \leq k$.
        for $i = 1$ **to** m **do**
          r_i has the form $q_i.\sigma \rightarrow t$.
          for all $(q', x) \in ydset(t) \cap (Q \times X_k)$ **do**
            $\phi(x) \leftarrow \phi(x) \cup \{q'\}$
        $P \leftarrow P \cup \{n \rightarrow \sigma(\phi(x_1), \ldots, \phi(x_k))\}$
return G

Algorithm 8 LIN-DOM-PROJ

inputs
  wLT $M = (Q, \Sigma, \Delta, R, q_0)$ over $\mathbb{W}$
outputs
  wrtg $G = (Q \cup \{\bot\}, \Sigma, P, q_0)$ over $\mathbb{W}$ such that $L_G = \mathrm{dom}(\tau_M)$
complexity
  $O(|R|)$

$P = (P', \pi) \leftarrow (\emptyset, \emptyset)$
for all $\sigma^{(k)} \in \Sigma$ **do**
  $P' \leftarrow P' \cup \{\bot \to \sigma(\overbrace{\bot, \ldots, \bot}^{k})\}$
  $\pi(\bot \to \sigma(\overbrace{\bot, \ldots, \bot}^{k})) \leftarrow 1$
for all $q \in Q$ **do**
  for all $\sigma \in \Sigma$ with rank k **do**
    for all $r \in R_{q,\sigma}$ **do**
      r has the form $q.\sigma \xrightarrow{w} u$.
      Let ϕ be a mapping $X_k \to Q \cup \{\bot\}$ where $\phi(x_i) = \bot$ for all $1 \leq i \leq k$.
      for all $(q', x) \in ydset(u) \cap (Q \times X_k)$ **do**
        $\phi(x) \leftarrow q'$
      $p_{new} \leftarrow q \to \sigma(\phi(x_1), \ldots, \phi(x_k))$
      $P' \leftarrow P' \cup \{p_{new}\}$
      $\pi(p_{new}) \leftarrow \pi(p_{new}) + w$
return G



(a) P_7 formed from domain projection of M_2 from Example 2.3.2 with rules in Figure 2.9b.

(b) P_8 formed from domain projection of M_3 from Example 2.3.2 with rules in Figure 2.9c.

Figure 2.11: Production sets formed from domain projection using Algorithms 7 and 8, as described in Example 2.3.5.

Example 2.3.5 We can use Algorithms 7 and 8 to obtain the domain projections of, respectively, M_2 and M_3, from Example 2.3.2, where M_2 is taken to be over the Boolean semiring and M_3 is taken to be over the tropical semiring. Let G_7 and G_8 be wrtgs such that $L_{G_7} = \text{dom}(M_2)$ and $L_{G_8} = \text{dom}(M_3)$. $G_7 = (\mathcal{P}(\{q, r\}), \Sigma, P_7, \{q\})$ and $G_8 = (\{q, \perp\}, \Sigma, P_8, q)$, where P_7 is in Figure 2.11a and P_8 is in Figure 2.11b.

Transducer classes xT and wxLT also have recognizable domain, as the proofs for their non-extended brethren are not dependent on particulars of the left side, but Algorithms 7 and 8 are not appropriate for these classes. We can, however, transform a wxT into a wT with equivalent domain using Algorithm 9. This algorithm calls Algorithm 10, which separates multi-height left sides of rules and preserves state transition information from the original rules, but discards syntactic rule right side information. If we then augment $\Sigma^{(1)}$ with an additional symbol ϵ with the understanding that $R_{q,\epsilon}$ signifies ϵ

rules beginning with q, and that $q.\epsilon \xrightarrow{w} u$ is equivalent to $q.x_1 \xrightarrow{w} u$, we may use Algorithms 7 and 8 on wxtts transformed from Algorithm 9 to obtain domain projections.

Algorithm 9 PRE-DOM-PROJ

inputs
  wxtt $M_{\text{in}} = (Q_{\text{in}}, \Sigma, \Delta, R_{\text{in}}, q_0)$ over $\mathbb{W}$, where $l = \max_{\sigma \in \Sigma} rk(\sigma)$ and $m = \max_{r \in R_{\text{in}}} \max_{i=1}^{l} \text{mult}(r, i)$.
outputs
  wtt $M_{\text{out}} = (Q_{\text{out}} \supseteq Q_{\text{in}}, \Sigma, \Gamma, R_{\text{out}}, q_0)$ over $\mathbb{W}$, where $\Gamma = \{v^{(0)}, \omega^{(lm)}\}$, such that if M_{in} is linear or $\mathbb{W}$ is Boolean, $\text{dom}(\tau_{M_{\text{in}}}) = \text{dom}(\tau_{M_{\text{out}}})$
complexity
  $O(|R_{\text{in}}|)$

$Q_{\text{out}} \leftarrow Q_{\text{in}}$
$R_{\text{out}} \leftarrow \emptyset$
for all $r \in R_{\text{in}}$ **do**
  r is of the form $q.y \xrightarrow{w} u$
  for all $r' \in$ PRE-DOM-PROJ-PROCESS$(\Sigma, \Delta, Q_{\text{out}}, lm, v, \omega, q, y, u, w)$ **do**
    r' is of the form $q'.\sigma \xrightarrow{w'} u'$.
    $R_{\text{out}} \leftarrow R_{\text{out}} \cup \{r'\}$
    $Q_{\text{out}} \leftarrow Q_{\text{out}} \cup \{q'\}$
return M_{out}

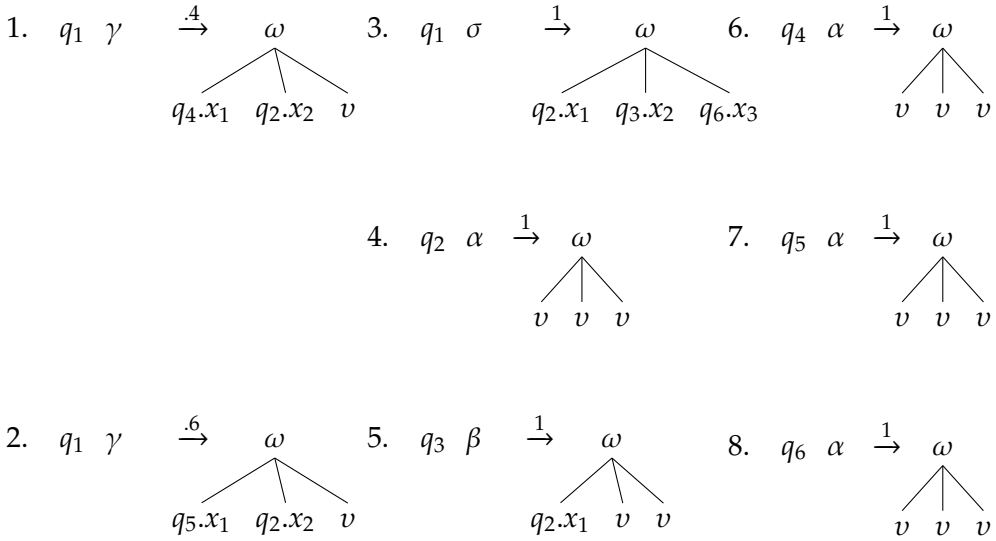
Example 2.3.6 Recall M_4 from Example 2.3.4. The result of Algorithm 9 on M_4 is $M_5 = (Q_5 = \{q_i \mid 1 \leq i \leq 6\}, \Sigma, \Gamma, R_5, q_1)$, where $\Gamma = \{v^{(0)}, \omega^{(3)}\}$, and R_5 is depicted in Figure 2.12a. The result of Algorithm 8 on M_5 is $G_9 = (Q_5, \Sigma, P_9, q_1)$, where P_9 is depicted in Figure 2.12b. Note that L_{G_9} is depicted in Figure 2.2c.

Transducer classes xLT (from [48], Thm. IV.6.5) and wxLNT (from [43]) have regular range. Algorithm 11 describes how to obtain range projections from wxtts of these classes. Note that the algorithm is defined for wxLT but is only applicable to xLT and wxLNT; should the input be, for example, in wxLT over some non-Boolean semiring

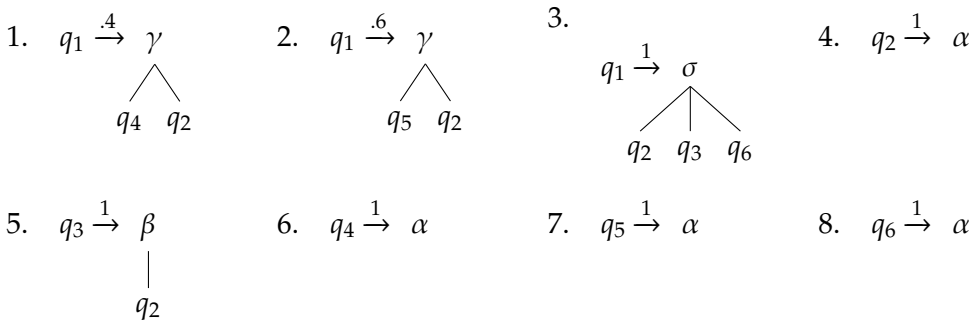
Algorithm 10 PRE-DOM-PROJ-PROCESS

inputs
  ranked alphabets Σ, Δ
  state set Q
  maximum new right side rank $r \in \mathbb{N}$
  rank-0 symbol v
  rank-r symbol ω
  state $q \in Q$
  tree $y \in T_\Sigma(X)$
  tree $u \in T_{\Delta \cup \{\chi\}}(Q \times X)$ where χ is a rank-0 "placeholder" symbol not in Σ or Δ
  weight w
outputs
  set of rules R' and states $Q' \supseteq Q$ such that for any wxtt $M = (Q, \Sigma, \Delta \cup \{v, \omega\}, R \cup \{q.y \xrightarrow{w} u\}, q_0)$, $\mathrm{dom}(\tau_M) = \mathrm{dom}(\tau_{M'})$, where $M' = (Q \cup Q', \Sigma, \Delta \cup \{v, \omega\}, R \cup R', q_0)$
complexity
  $O(size(y))$

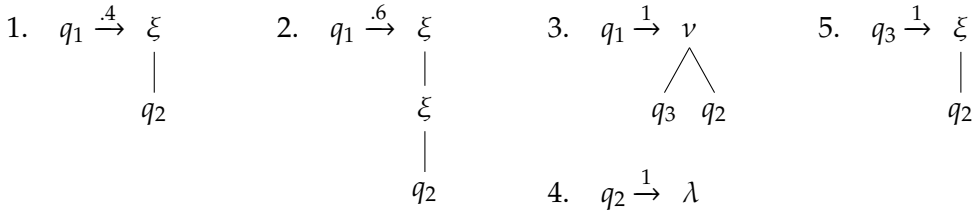
$R' \leftarrow \emptyset$
$Q' \leftarrow \emptyset$
$\Psi \leftarrow \emptyset$
Let $b_1 = \ldots = b_r = v$
$m \leftarrow 1$
Let y be of the form $\sigma^{(k)}(y_1, \ldots, y_k)$.
Form substitution map $\varphi : Q \times X \to T_{\Delta \cup \{\chi\}}(Q \times X)$.
for $i = 1$ **to** k **do**
  if $y_i \in X$ **then**
    for all $(q_i, y_i) \in yset(u) \cap (Q \times \{y_i\})$ **do**
      $\varphi(q_i, y_i) \leftarrow \chi$
      $b_m \leftarrow (q_i, x_m)$
      $m \leftarrow m + 1$
  else
    Let q_x be a new state such that $Q \cap \{q_x\} = \emptyset$.
    $Q' \leftarrow Q' \cup \{q_x\}$
    $\Psi \leftarrow \Psi \cup \{(q_x, y_i)\}$
    $b_m \leftarrow (q_x, x_m)$
    $m \leftarrow m + 1$
  for all $(q_x, y_x) \in \Psi$ **do**
    $R', Q' \leftarrow R', Q' \cup$ PRE-DOM-PROJ-PROCESS$(\Sigma, \Delta, Q, r, v, \omega, q_x, y_x, \varphi(u), 1)$
$R' \leftarrow R' \cup \{q.\sigma \xrightarrow{w} \omega(b_1, \ldots, b_r)\}$
return R', Q'



(a) Rule set R_5 formed as the result of pre-domain conversion of M_4, Algorithm 9, as described in Example 2.3.6.



(b) Production set P_9, formed as the result of domain projection, Algorithm 8, on M_5, which has rules depicted in Figure 2.12a, as described in Example 2.3.6.



(c) Production set P_{10}, formed as the result of range projection, Algorithm 11, on M_4, as described in Example 2.3.7.

Figure 2.12: Transformations of M_4 from Example 2.3.4, depicted in Figure 2.10b, for use in domain and range projection Examples 2.3.6 and 2.3.7.

the result of the algorithm is not guaranteed to be meaningful. More specifically, the coefficients of the represented tree series may be wrong.

Algorithm 11 RANGE-PROJ

inputs
  wxLT $M = (Q, \Sigma, \Delta, R, q_0)$ over semiring $\mathbb{W}$.
outputs
  wrtg $G = (Q, \Delta, P, q_0)$ over semiring $\mathbb{W}$ such that, if M is nondeleting or $\mathbb{W}$ is Boolean, $L_G = \text{range}(\tau_M)$
complexity
  $O(|R| \max_{r \in R} size(r))$

$P = (P', \pi) \leftarrow (\emptyset, \emptyset)$
Let φ be a substitution mapping $Q \times X \to T_\Delta(Q)$ such that for all $q \in Q$ and $x \in X$, $\varphi((q, x)) = q$.
for all r of the form $q.y \xrightarrow{w} z$ in R **do**
  $p_{new} \leftarrow q \to \varphi(z)$
  $P' \leftarrow P' \cup \{p_{new}\}$
  $\pi(p_{new}) \leftarrow \pi(p_{new}) + w$
return G

Example 2.3.7 Recall $M_4 = (Q_4, \Sigma, \Delta, R_4, q_1)$ from Example 2.3.4. The result of Algorithm 11 on M_4 is $G_{10} = (Q_4, \Delta, P_{10}, q_1)$, where P_{10} is depicted in Figure 2.12c. Note that $L_{G_{10}}$ is depicted in Figure 2.2d.

2.3.2 Embedding

It is sometimes useful to *embed* a wrtg in a wtt, that is, given a wrtg G, to form a wtt M such that $\tau_M = \iota_{L_G}$. Algorithm 12 is a very simple algorithm for forming this embedding from a normal form and chain-production-free wrtg; this can be done with an arbitrary wrtg in an analogous manner to that of the algorithm, but the resulting embedding will be a wxtt.

Algorithm 12 EMBED

inputs
  wrtg $G = (N, \Sigma, P, n_0)$ over $\mathbb{W}$ in normal form with no chain productions
outputs
  wtt $M = (N, \Sigma, \Sigma, R, n_0)$ over $\mathbb{W}$ such that $\tau_M = \iota_{L_G}$
complexity
  $O(|P|)$

for all p of the form $n \xrightarrow{w} \sigma(n_1, \ldots, n_k) \in P$ **do**
  $R \leftarrow R \cup \{n.\sigma \xrightarrow{w} \sigma(n_1.x_1, \ldots, n_k.x_k)\}$
return M

1. n_0 S $\xrightarrow{1}$ S
   $n_1.x_1$ $n_2.x_2$
2. n_1 NP $\xrightarrow{.4}$ NP
   $n_3.x_1$ $n_4.x_2$
3. n_3 the $\xrightarrow{.8}$ the
4. n_4 dogs $\xrightarrow{.7}$ dogs
5. n_2 VP $\xrightarrow{.42}$ VP
   $n_5.x_1$ $n_6.x_2$
6. n_2 VP $\xrightarrow{.1}$ VP
   $n_7.x_1$ $n_8.x_2$
7. n_5 eat $\xrightarrow{1}$ eat
8. n_7 chase $\xrightarrow{1}$ chase
9. n_6 NP $\xrightarrow{.25}$ NP
   $n_9.x_1$ $n_{10}.x_2$
10. n_8 NP $\xrightarrow{.25}$ NP
    $n_{11}.x_1$ $n_{12}.x_2$
11. n_9 red $\xrightarrow{.4}$ red
12. n_{11} smelly $\xrightarrow{.6}$ smelly
13. n_{12} meat $\xrightarrow{.5}$ meat
14. n_{10} cars $\xrightarrow{.5}$ cars

Figure 2.13: Rule set R_6, formed from embedding of wrtg G_7 from Example 2.2.7, as described in Example 2.3.8.

Example 2.3.8 Recall the wrtg $G_7 = (\{n_i \mid 0 \leq i \leq 12\}, \Sigma, P_7, n_0)$, where P_7 is depicted in Figure 2.8b. Then $M_6 = (\{n_i \mid 0 \leq i \leq 12\}, \Sigma, \Sigma, R_6, n_0)$, where R_6 is depicted in Figure 2.13, is the result of Algorithm 12, an embedding of G_7.

2.3.3 Composition

In Section 2.1.3 we described the composition of two weighted tree transformations τ and μ. Here we consider composition of *transducers* (sometimes referred to as *syntactic* composition). In other words, given two wtts M_A and M_B, we want to construct a wtt $M_A \circ M_B$ such that $\tau_{M_A \circ M_B} = \tau_{M_A}; \tau_{M_B}$.

A construction was given in declarative terms for syntactic composition of two unweighted top-down tree transducers M_A and M_B by Baker [6]. This construction was shown to be correct for the cases where (M_B is linear or M_A is deterministic) and (M_B is nondeleting or M_A is total). The construction was extended to the weighted case by Maletti [88]. An inspection of Baker's construction is enough to satisfy that it may be generalized to allow M_A to be a wxtt without altering matters. Note, though, that the deterministic and total properties are not defined for wxtt, so any composition construction that involves a wxT as M_A will require a wLNT as M_B. We re-state Baker's construction, with the additional generalization and modification to handle weights provided by Maletti, as follows:

Let $M_A = (Q_A, \Sigma, \Delta, R_A, q_{0_A})$ and $M_B = (Q_B, \Delta, \Gamma, R_B, q_{0_B})$ be wtts. Then define $M_C = ((Q_A \times Q_B), \Sigma, \Gamma, R_C, (q_{0_A}, q_{0_B}))$, where members of R_C are found by the following process:

1. Augment M_B such that it represents transformations from $T_{\Delta \cup (Q_A \times X)}$ to $T_{\Gamma \cup (Q_A \times Q_B \times X)}$ by adding, for all $q_A \in Q_A$, $x \in X$, and $q_B \in Q_B$, the rule $q_B.(q_A, x) \xrightarrow{1} ((q_A, q_B), x)$ to R_B.

2. Using the augmented M_B, for all rules $q_A.y \xrightarrow{w_1} u$ in R_A, all states q_B in Q_B, and all z such that $\tau_{M_B}(u,z)_{q_B}$ is non-zero, add the rule $(q_A,q_B).y \xrightarrow{w_1 \cdot \tau_{M_B}(u,z)_{q_B}} z$ to R_C.

3. If R_C contains rules that only differ by their weight, replace these rules with a single rule that has as a weight the sum of the weights of the replaced rules.

This process, essentially a reformulation of the construction of Baker [6], succinctly and intuitively describes how the composition is constructed. However, as discussed in Section 1.7, this text does not provide an actual, implementable algorithm for obtaining the composition transducer. For one, the first step requires adding an infinite number of rules. This problem is easily solved by only adding such rules as may be necessitated by the second step. Of more concern is the method by which the second step is performed—the description above gives no hint as to how one may actually obtain the specified z.

Algorithm 13, COMPOSE, seeks to correct precisely this omission. It is an algorithmic description of the aforementioned composition construction of weighted tree transducers. The algorithm takes as input a wxT M_A and a wLNT M_B and produces a wxtt M_C such that $M_C = M_A \circ M_B$. As in the declarative presentation, the main algorithm and general idea is rather similar to composition algorithms for wst. Like in the wst case, for each state of the composition transducer (i.e., for each pair of states, one from M_A and one from M_B), rules from M_A and M_B are combined.[6] The key difference is that while for strings, one rule from M_A is paired with one rule from M_B, here multiple rules from M_B may be necessary to match a single rule from M_A. Specifically, every *tiling* of rules from M_B with left sides that "cover" a potentially large right side of a rule from M_A must

[6] As previously discussed for Algorithms 6, 7, and 8, an implementation of this algorithm should consider states as they are encountered, and not iterate over all possible states.

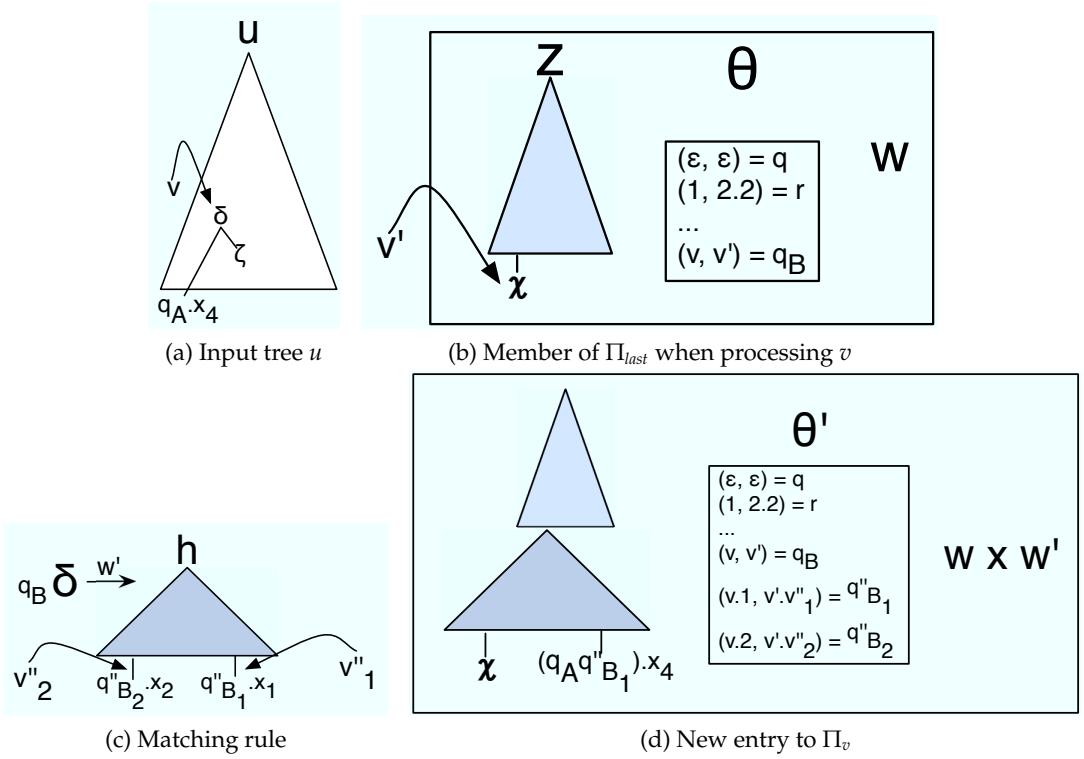
be chosen. The restriction that M_B not be extended ensures that there are only a finite number of such tilings (see Arnold and Dauchet [5] for more details).

The act of forming all tilings of a tree by a transducer is handled by the sub-algorithm COVER, presented as algorithm 14. The input to the algorithm is a tree, u, a transducer, M_B, and a state q_B. The desired output is the set of all trees that are formed as a consequence of tiling u with left hand sides of rules from M_B and joining the right hand sides of those rules in the natural way. Additionally, the product of the weights of the rules used to form these trees is desired. COVER proceeds in a top-down manner, finding all rules that match the root of u, and building an incomplete result tree for each matching rule. Then, for each incomplete result tree, another step down the input tree is taken, forming more partial results, and so on until all the possible full coverings and complete result trees are formed. Figure 2.14 graphically explains how COVER works. Additionally, the following example provides a walkthrough of COMPOSE and COVER.

Algorithm 13 COMPOSE

inputs
  wxT $M_A = (Q_A, \Sigma, \Delta, R_A, q_{A_0})$ over $\mathbb{W}$
  wLNT $M_B = (Q_B, \Delta, \Gamma, R_B, q_{B_0})$ over $\mathbb{W}$
outputs
  wxT $M_C = ((Q_A \times Q_B), \Sigma, \Gamma, R_C, (q_{A_0}, q_{B_0}))$ over $\mathbb{W}$ such that $M_C = M_A \circ M_B$.
complexity
  $O(|R_A| \max(|R_B|^{size(\tilde{u})}, |Q_B|))$ where $\tilde{u}$ is the largest right side tree in any rule in R_A

Let R_C be of the form (R'_C, π)
$R_C \leftarrow (\emptyset, \emptyset)$
for all $(q_A, q_B) \in Q_A \times Q_B$ **do**
  for all r of the form $q_A.y \xrightarrow{w_1} u$ in R_A **do**
    for all $(z, w_2) \mid (z, \theta, w_2) \in \text{COVER}(u, M_B, q_B)$ **do**
      $r_{new} \leftarrow (q_A, q_B).y \rightarrow z$
      $R'_C \leftarrow R'_C \cup \{r_{new}\}$
      $\pi(r_{new}) \leftarrow \pi(r_{new}) + (w_1 \cdot w_2)$
return M_C



(a) Input tree u

(b) Member of Π_{last} when processing v

(c) Matching rule

(d) New entry to Π_v

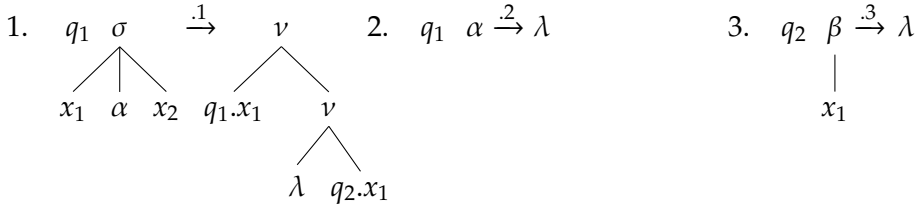
Figure 2.14: Graphical representation of COVER, Algorithm 14. At line 13, position v of tree u is chosen. As depicted in Figure 2.14a, in this case, u(v) is δ and has two children. One member (z, θ, w) of Π_{last} is depicted in Figure 2.14b. The tree z has a leaf position v' with label χ and there is an entry for (v, v') in θ, so as indicated on lines 16 and 17, we look for a rule with state $\theta(v, v') = q_{\text{B}}$ and left symbol δ. One such rule is depicted in Figure 2.14c. Given the tree u, the triple (z, θ, w), and the matching rule, we can build the new member of Π_v depicted in Figure 2.14d as follows: The new tree is built by first transforming the (state, variable) leaves of h; if the ith child of v is a (state, variable) symbol, say, (q, x), then leaves in h of the form (q'', x_i) are transformed to (q, q'', x) symbols, otherwise they become χ. The former case, which is indicated on line 24, accounts for the transformation from $q''_{\text{B}_1}.x_1$ to $(q_{\text{A}}, q''_{\text{B}_1}).x_4$. The latter case, which is indicated on line 26, accounts for the transformation from $q''_{\text{B}_2}.x_2$ to χ. The result of that transformation is attached to the original z at position v'; this is indicated on line 27. The new θ' is extended from the old θ, as indicated on line 18. For each immediate child vi of v that has a corresponding leaf symbol in h marked with x_i at position v'', the position in the newly built tree will be $v'v''$. The pair $(vi, v'v'')$ is mapped to the state originally at v'', as indicated on line 22. Finally, the new weight is obtained by multiplying the original weight, w with the weight of the rule, w'.

Algorithm 14 COVER

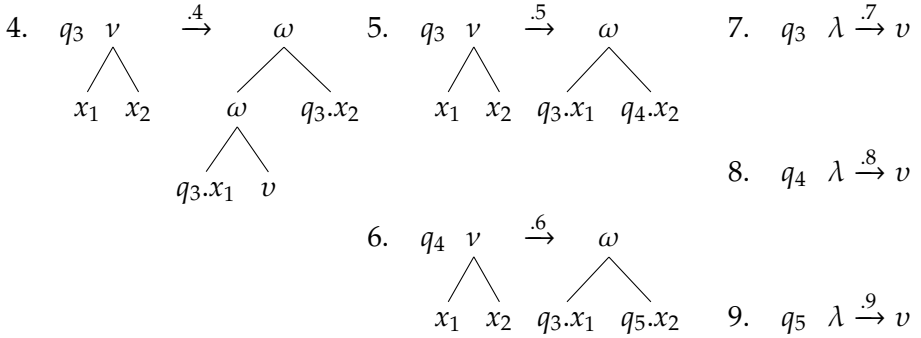
inputs
  $u \in T_\Delta(Q_A \times X)$
  wT $M_B = (Q_B, \Delta, \Gamma, R_B, q_{2_0})$ over $\mathbb{W}$
  state $q_B \in Q_B$
outputs
  set Π of triples $\{(z, \theta, w) : z \in T_\Gamma((Q_A \times Q_B) \times X)$, θ a partial mapping $pos(u) \times pos(z) \to Q_B$, and $w \in \mathbb{W}\}$, each triple indicating a successful run on u by rules in R_B, starting from q_B, forming z, and w, the weight of the run.
complexity
  $O(|R_B|^{size(u)})$

if $u(\varepsilon)$ is of the form $(q_A, x) \in Q_A \times X$ **then**
  $\Pi_{last} \leftarrow \{(((q_A, q_B), x), \{((\varepsilon, \varepsilon), q_B)\}, 1)\}$
else
  $\Pi_{last} \leftarrow \{(\chi, \{((\varepsilon, \varepsilon), q_B)\}, 1)\}$
for all $v \in pos(u)$ such that $u(v) \in \Delta^{(k)}$ for some $k \geq 0$ in prefix order **do**
  $\Pi_v \leftarrow \emptyset$
  for all $(z, \theta, w) \in \Pi_{last}$ **do**
    for all $v' \in leaves(z)$ such that $z(v') = \chi$ **do**
      for all $\theta(v, v').u(v) \xrightarrow{w'} h \in R_B$ **do**
        $\theta' \leftarrow \theta$
        Form substitution mapping $\varphi : (Q_B \times X) \to T_\Gamma((Q_A \times Q_B \times X) \cup \{\chi\})$.
        for $i = 1$ **to** k **do**
          for all $v'' \in pos(h)$ such that $h(v'') = (q_B'', x_i)$ for some $q_B'' \in Q_B$ **do**
            $\theta'(vi, v'v'') \leftarrow q_B''$
            if $u(vi)$ is of the form $(q_A, x) \in Q_A \times X$ **then**
              $\varphi(q_B'', x_i) \leftarrow ((q_A, q_B''), x)$
            else
              $\varphi(q_B'', x_i) \leftarrow \chi$
        $\Pi_v \leftarrow \Pi_v \cup \{(z[\varphi(h)]_{v'}, \theta', w \cdot w')\}$
    $\Pi_{last} \leftarrow \Pi_v$
return Π_{last}

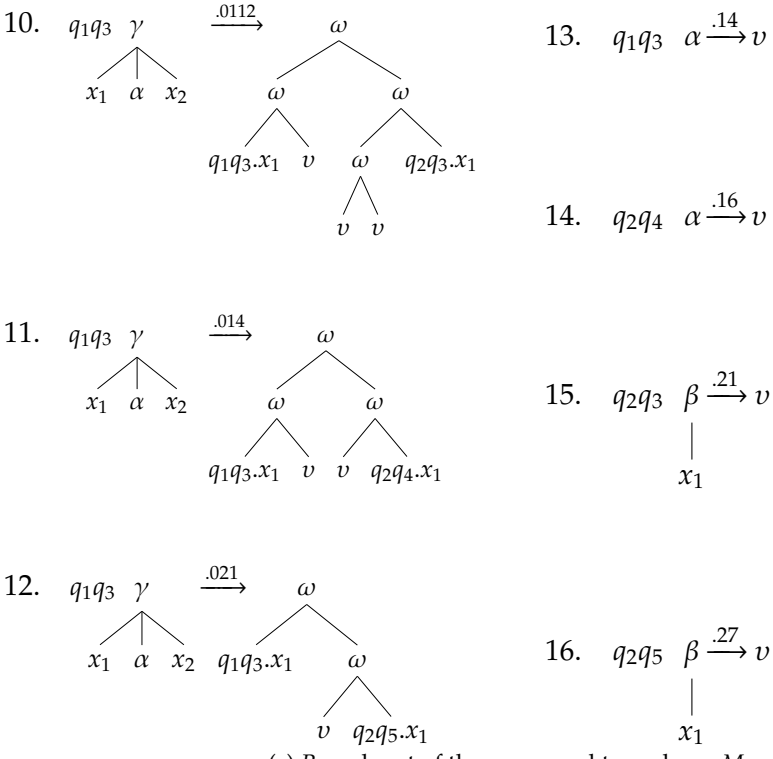
Example 2.3.9 Let $M_7 = (\{q_1, q_2\}, \Sigma, \Delta, R_7, q_1)$ and $M_8 = (\{q_3, q_4, q_5\}, \Delta, \Gamma, R_8, q_3)$ be wtts where Σ and Δ are those ranked alphabets defined in Examples 2.3.2, 2.1.1, and 2.1.3, $\Gamma = \{\upsilon^{(0)}, \psi^{(1)}, \omega^{(2)}\}$, and R_7 and R_8 are depicted in Figures 2.15a and 2.15b, respectively. Then $M_9 = (\{q_1q_3, q_2q_3, q_2q_4, q_2q_5\}, \Sigma, \Gamma, R_9, q_1q_3)$, where R_9 is depicted in Figure 2.15c, is formed by Algorithm 13 applied to M_7 and M_8.



(a) R_7, rule set of M_7.



(b) R_8, rule set M_8.



(c) R_9, rule set of the composed transducer M_9.

Figure 2.15: Rule sets for transducers described in Example 2.3.9.

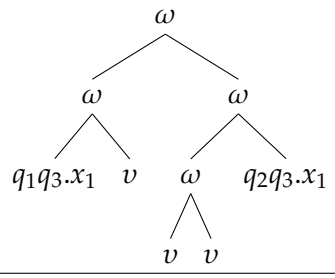
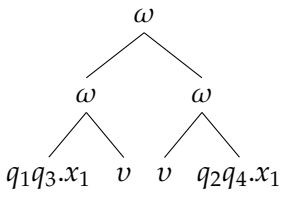
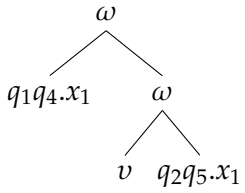
θ	z	w
$(\epsilon, \epsilon) \to q_3$ $(1, 1.1) \to q_3$ $(2, 2) \to q_3$ $(2.1, 2.1.1) \to q_3$ $(2.2, 2.2) \to q_3$		.112
$(\epsilon, \epsilon) \to q_3$ $(1, 1.1) \to q_3$ $(2, 2) \to q_3$ $(2.1, 2.1) \to q_3$ $(2.2, 2.2) \to q_4$		.14
$(\epsilon, \epsilon) \to q_3$ $(1, 1) \to q_3$ $(2, 2) \to q_4$ $(2.1, 2.1) \to q_3$ $(2.2, 2.2) \to q_5$		.21

Figure 2.16: Π formed in Example 2.3.9 as a result of applying Algorithm 14 to $\nu(q_1.x_1, \nu(\lambda, q_2.x_1))$, M_8, and q_3.

Let us describe in more detail how the rules of Figure 2.15c are formed, particularly rules 10, 11, and 12. The first state pair chosen in line 10 is (q_1, q_3). Of particular interest to us is what happens when rule 1 is chosen at line 11. We now turn to Algorithm 14, which finds all coverings of $\nu(q_1.x_1, \nu(\lambda, q_2.x_1))$ with rules from M_8, starting from q_3. A covering is denoted by the output tree z that is formed from assembling right sides of rules from M_8 (and that will become the newly produced rule's right side), the mapping θ, which indicates how z was built, and the derivation weight w corresponding to the product of the weights in the rules used to form the covering. Figure 2.16 shows the three entries for z, θ, and w corresponding to the three coverings of $\nu(q_1.x_1, \nu(\lambda, q_2.x_1))$ with rules from M_8, starting from q_3. The entries correspond to the derivations (4, 4, 7), (4, 5, 7), and (5, 6, 7), respectively.

Variations on COVER are presented in Chapter 4. These variations are different from the algorithm presented here in that the "tiling" device is a wrtg, not a transducer, a result tree is not explicitly built, and the algorithms allow on-the-fly discovery of the tiling wrtg's productions.

2.4 Tree-to-string and string machines

We have thus far focused our discussion on tree-generating automata (wrtgs) and tree-transforming transducers (wxtts and wtts). However, it is frequently the case that we wish to transform between tree and string objects. Parsing, where a tree structure is placed on top of a given string, and recent work in syntactic machine translation are good examples of this. This motivates the description of two more formal machines, the

weighted context-free grammar, a well studied string analogue to wrtgs, and the *weighted, extended tree-to-string transducer*, a minor variation on wxtts.

Note in the definitions below that ϵ is a special symbol not contained in any terminal, nonterminal, or state set. It denotes the empty word.

Definition 2.4.1 (cf. Salomaa and Soittola [117]) A weighted context-free grammar (wcfg) over semiring $\mathbb{W}$ is a 4-tuple $G = (N, \Delta, P, n_0)$ where:

1. N is a finite set of *nonterminals*, with $n_0 \in N$ the start nonterminal

2. Δ is the terminal alphabet.

3. P is a tuple (P', π), where P' is a finite set of *productions*, each production p of the form $n \to g$, $n \in N$, $g \in (\Delta \cup N)^*$, and $\pi : P' \to \mathbb{W}$ is a weight function of the productions. Within these constraints we may (and usually do) refer to P as a finite set of weighted productions, each production p of the form $n \xrightarrow{\pi(p)} g$. We associate P with G, such that, e.g., $p \in G$ is interpreted to mean $p \in P$.

For wcfg $G = (N, \Delta, P, n_0)$, $e, f, g \in (\Delta \cup N)^*$, $n \in N$, and $p \in P$ of the form $n \xrightarrow{w} g \in P$, we obtain a *derivation step* from e to f by replacing an instance of n in e with g. Formally, $e \Rightarrow_G^p f$ if there exist $e', e'' \in (\Delta \cup N)^*$ such that $e = e'ne''$ and $f = e'ge''$. We say this derivation step is *leftmost* if $e' \in \Delta^*$. Except where noted and needed, we henceforth assume all derivation steps are leftmost and drop the subscript G. If, for some $m \in \mathbb{N}$, $p_i \in P$, and $e_i \in (\Delta \cup N)^*$ for all $1 \leq i \leq m$, $n_0 \Rightarrow^{p_1} e_1 \ldots \Rightarrow^{p_m} e_m$, we say the sequence $(p_1, \ldots, p_m)$ is a *derivation* of e_m in G and that $n_0 \Rightarrow^* e_m$. The weight of a derivation d, $wt(d)$, is the product of the weights of all occurrences of its productions.

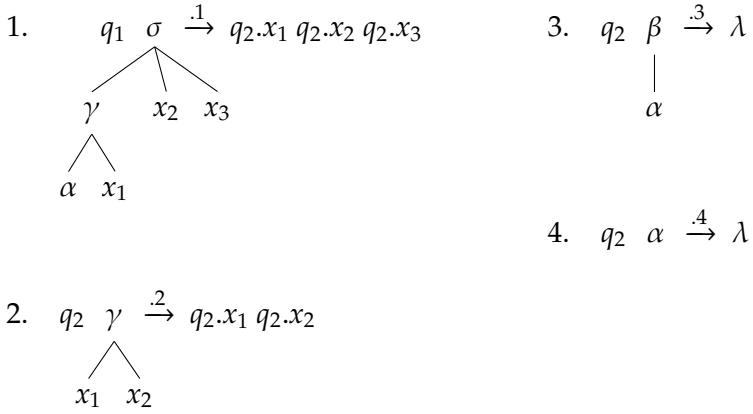
Definition 2.4.2 A weighted extended top-down tree-to-string transducer (wxtst) is a 5-tuple $M = (Q, \Sigma, \Delta, R, q_0)$ where:

1. Q and Σ are defined as for wtts, and Δ is defined as for wcfgs.

2. R is a tuple (R', π). R' is a finite set of *rules*, each rule r of the form $q.y \xrightarrow{w} g$ for $q \in Q$, $y \in T_\Sigma(X)$, and $g \in (\Delta \cup (Q \times X))^*$. We further require that y is linear in X, i.e., no variable $x \in X$ appears more than once in y, and that each variable appearing in g is also in y. $\pi : R' \to \mathbb{W}$ is a weight function of the rules. As for wrtgs, wtts, and wcfgs, we refer to R as a finite set of weighted rules, each rule r of the form $q.y \xrightarrow{\pi(r)} g$.

For wxtst $M = (Q, \Sigma, \Delta, R, q_0)$, $e, f \in (\Delta \cup (Q \times T_\Sigma))^*$, $q \in Q$, and $r \in R$ of the form $q.y \xrightarrow{w} g$ where $g = g_1 g_2 \dots g_k$ for some $k \in \mathbb{N}$, and $g_i \in \Delta \cup (Q \times X)$, $1 \leq i \leq k$, we obtain a *derivation step* from e to f by replacing some substring of e of the form (q, t), where $q \in Q$, $t \in T_\Sigma$, and t matches y, by a transformation of g, where each instance of a variable has been replaced by a corresponding subtree of the y-matching tree. Formally, $e \Rightarrow^r_M f$ if there exist $e', e'' \in (\Delta \cup (Q \times T_\Sigma))^*$ such that $e = e'(q, t)e''$, a substitution mapping $\varphi : X \to T_\Sigma$, and a rule $q.y \xrightarrow{w} g \in R$ such that $t = \varphi(y)$ and $f = e'\theta(g_1) \dots \theta(g_k)e''$, where θ is a mapping $\Delta \cup (Q \times X) \to \Delta \cup (Q \times T_\Sigma)$ defined such that $\theta(\lambda) = \lambda$ for all $\lambda \in \Delta$ and $\theta(q, x) = (q, \varphi(x))$ for all $q \in Q$ and $x \in X$. We define *leftmost*, *derivation*, and *wt* for wxtst as we do for wcfg. We also define a *weighted tree-string transformation* $\tau_M(s, f)$ for all $s \in T_\Sigma$ and $f \in \Delta^*$ in an analogous way to that for weighted tree transformations.

We extend the properties *linear* and *nondeleting* to wxtst. We differentiate wxtst from wxtt by appending the letter "s" and omit the letter "x" to denote wxtst without extended



(a) rule set R_{10} from example xLNTs M_{10}.

1. $q_1 \xrightarrow{.1} q_2\ q_2\ q_2$ 2. $q_2 \xrightarrow{.2} q_2\ q_2$ 3. $q_2 \xrightarrow{.7} \lambda$

(b) production set P_{11} from from example wcfg G_{11}.

Figure 2.17: Rules and productions for an example wxtst and wcfg, respectively, as described in Example 2.4.3.

left sides (i.e., wtsts). Thus, xNTs is the class of nondeleting wxtsts over the Boolean semiring, and wLNTs is the class of linear and nondeleting wtsts over an arbitrary semiring. Note that as for wcfgs, the right side of a wxtst rule can be ϵ, but we retain the original nomenclature for an ϵ-free wxtst.

We do not provide an extensive recap of algorithms for these string-based structures as we did for wrtgs and wxtts, as most of the algorithms previously presented are intuitively extendable, or inapplicable to the string or tree-string case. In Chapters 4 and 5 we will discuss algorithms that are interestingly defined for wxtsts.

Example 2.4.3 Let $M_{10} = (\{q_1, q_2\}, \Sigma, \Delta, R_{10}, q_1)$ be a wxtst where Σ is the ranked alphabet defined in Examples 2.3.2, 2.1.1, 2.1.3, and 2.3.9, $\Delta = \{\lambda\}$, and R_{10} is depicted in Figure 2.17a. Additionally, let $G_{11} = (\{q_1, q_2\}, \Delta, P_1, q_1)$ be a wcfg where P_{11} is depicted in Figure 2.17b. Note that G_{11} is the range projection of M_{10}.

2.5 Useful classes for NLP

Knight [73] laid out four properties tree transducers should have if they are to be of much use in practical NLP systems:

- They should be *expressive* enough to capture complicated transformations seen in observed natural language data.
- They should be *inclusive* enough to generalize other well-known transducer classes.
- They should be *teachable* so that a model of real-world transformations informed by observed data can be built such that the model's parameters coincide with the transducers' transformations.
- They should be *modular*, so that complicated transformations can be broken down into several discrete, easy-to-build transducers and then used together.

Knight instantiated these general properties in four concrete ways:

- An expressive transducer can capture the local rotation demonstrated in Figure 2.18.
- A teachable transducer can use EM and a training corpus to assign weights to the transducer's rules.
- An inclusive transducer generalizes wfst.
- A modular transducer is closed under composition.

Knight analyzed several classes of tree transducer, including many of those discussed in this chapter, as well as classes beyond the scope of this thesis such as bottom-up tree

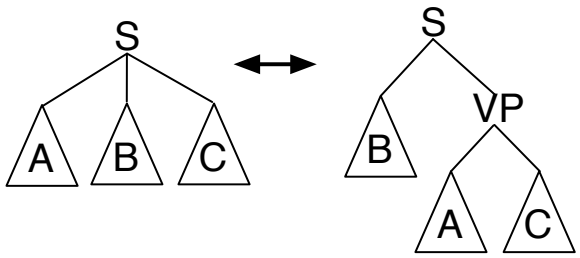


Figure 2.18: Example of the tree transformation power an expressive tree transducer should have, according to Knight [73]. The re-ordering expressed by this transformation is widely observed in practice, over many sentences in many languages.

transducers, and determined that none of them had all four of the desired properties. One of the classes analyzed in [73], wxLNT, is of particular interest because it is the basis of a state-of-the-art machine translation system [47, 46]. From the perspective of the four desired properties it is also quite promising, as wxLNT is expressive, teachable, and inclusive, though not modular according to Knight's definitions, because it is not closed under composition. However, one may reasonably consider a transducer as modular if it and its inverse *preserve recognizability*. A transducer (or its inverse) preserves recognizability if the transformation of all members of a recognizable language by the transducer is itself a recognizable language.[7] Transducer classes with this property satisfy the idea of modularity since they can then be used in a pipelined cascade in a forward or backward direction, with each transducer processing the output of its neighbor.

Figure 2.19, which is analogous to a similar figure in [73], depicts the generalization relationship between the classes of transducer discussed in this thesis as well as the desired properties they satisfy, substituting preservation of recognizability for closure under composition. As can be seen from the figure, wxLNT possesses all the desired

[7]We will discuss preservation of recognizability formally and in greater detail in Chapter 4.

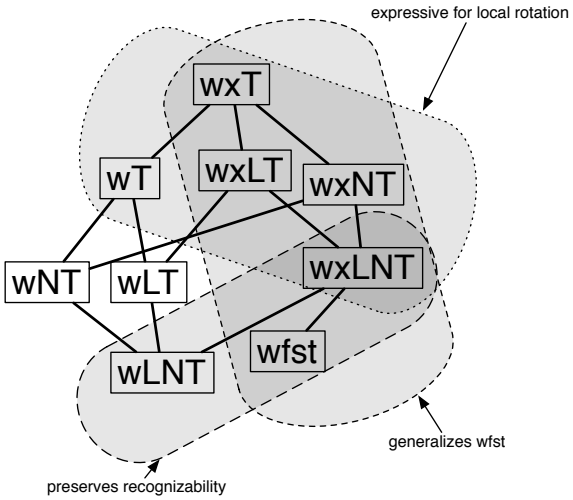


Figure 2.19: Tree transducers and their properties, inspired by a similar diagram by Knight [73]. Solid lines indicate a generalization relationship. Shaded regions indicate a transducer class has one or more of the useful properties described in Section 2.5. All transducers in this figure have an EM training algorithm.

properties and is thus a good choice for further research. With the exception of composition, all the tree transducer-related algorithms discussed in this thesis are appropriate for wxLNT, and even the composition algorithm can be used to compose a wxLNT with a wLNT. The tree-to-string variant of wxLNT, wxLNTs, also satisfies these properties, if we consider modularity for a wxLNTs to mean it produces a recognizable *string* language given a recognizable tree language as input.

2.6 Summary

We have discussed the principal formal structures of interest in this thesis and presented useful algorithms for them, some of which are presented in implementable algorithmic form for the first time. We defer discussion of some algorithms, such as weighted

determinization of wrtgs, training of wrtgs and wxtts, and application of a wxtt to a wrtg to subsequent chapters.

Chapter 3

Determinization of Weighted Tree Automata

In this chapter we present the first practical determinization algorithm for chain production-free wrtgs in normal form.[1] This work, which was first presented in [95], elevates a determinization algorithm for wsas to the tree case and demonstrates its effectiveness on two real-world experiments using acyclic wrtgs. We additionally present joint work with Matthias Büchse and Heiko Vogler that proves the correctness of our determinization algorithm for acyclic wrtgs and lays out the conditions for which this algorithm is appropriate for cyclic wrtgs. That work was first presented in [17].

3.1 Motivation

A useful tool in natural language processing tasks such as translation, speech recognition, parsing, etc., is the ranked list of results. Modern systems typically produce competing partial results internally and return only the top-scoring complete result to the user. They

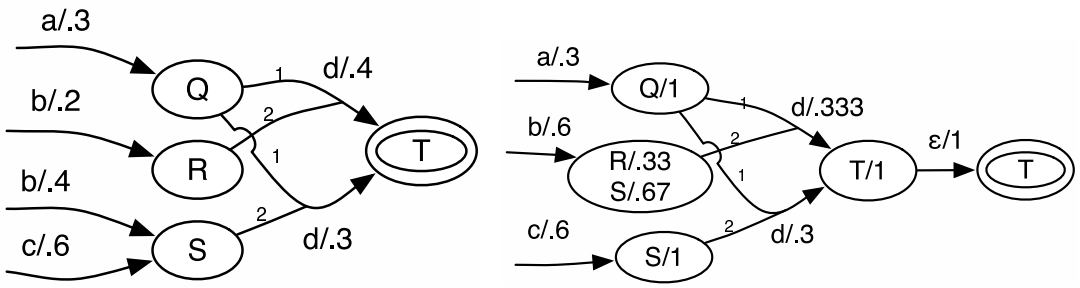
[1] A chain production-free wrtg in normal form is equivalent to a weighted tree automaton (wta). In this chapter we will primarily speak in terms of wrtgs but our visualizations will be of wtas, which provide some visual intuition. Additionally it should be noted that the original work this chapter is based on discussed wtas exclusively.

are, however, also capable of producing lists of runners-up, and such lists have many practical uses:

- The lists may be inspected to determine the quality of runners-up and motivate model changes.
- The lists may be re-ranked with extra knowledge sources that are difficult to apply during the main search.
- The lists may be used with annotation and a tuning process, such as in Collins and Roark [26], to iteratively alter feature weights and improve results.

Figure 3.2 shows the best 10 English translation parse trees obtained from a syntax-based translation system based on that of Galley et al. [47]. Notice that the same tree occurs multiple times in this list. This repetition is quite characteristic of the output of ranked lists. It occurs because many systems, such as the ones proposed by Bod [10], Galley et al. [47], and Langkilde and Knight [84] represent their result space in terms of weighted partial results of various sizes that may be assembled in multiple ways. There is in general more than one way to assemble the partial results to derive the same complete result. Thus, the k-best list of results is really a k-best list of *derivations*.

When list-based tasks, such as the ones mentioned above, take as input the top k results for some constant k, the effect of repetition on these tasks is deleterious. A list with many repetitions suffers from a lack of useful information, hampering diagnostics. Repeated results prevent alternatives that would be highly ranked in a secondary reranking system from even being considered. And a list of fewer unique trees than expected can cause overfitting when this list is used to tune. Furthermore, the actual weight of



(a) Nondeterministic wrtg (b) After determinization

Figure 3.1: Example of weighted determinization. We have represented a nondeterministic wrtg as a weighted tree automaton, then used the algorithm presented in this chapter to return a deterministic equivalent.

obtaining any particular tree is split among its repetitions, distorting the actual relative weights between trees. As a more concrete example, consider Figure 3.1a, which depicts a nondeterministic wrtg (as a bottom-up wta) over the probability semiring. This wrtg has three paths, one of which recognizes the tree $d(a, c)$ with weight .054, and two of which recognize the tree $d(a, b)$, once with weight .024 and once with weight .036. The highest weighted path in this wrtg does not recognize the highest weighted tree in the associated tree series. We would prefer a deterministic wrtg, such as in Figure 3.1b, which combines the two paths for $d(a, b)$ into one.

Mohri [99] encountered this problem in speech recognition, and presented a solution to the problem of repetition in k-best lists of strings that are derived from wfsas. That work described a way to use a powerset construction along with an innovative bookkeeping system to determinize a wfsa, resulting in a wfsa that preserves the language but provides a single, properly weighted derivation for each string in it. Put another way, if the input wfsa has the ability to generate the same string with different weights, the output wfsa generates that string with weight equal to the sum of all of the paths generating that string in the input wfsa. Mohri and Riley [104] combined this technique with a procedure for

```
34.73:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(caused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
•34.74:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
34.83:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(caused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
•34.83:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
34.84:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(caused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
34.85:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(caused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
•34.85:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
•34.85:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
34.87:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VB(arouse) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
•34.92:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
```

Figure 3.2: Ranked list of machine translation results with repeated trees. Scores shown are negative logs of calculated weights, thus a lower score indicates a higher weight. The bulleted sentences indicate identical trees.

efficiently obtaining k-best lists, yielding a list of string results with no repetition. Mohri's algorithm was shown to be correct if it terminates, and shown to terminate for acyclic wsas over the probability and tropical semirings [101] as well as for cyclic wsas over the tropical semiring that satisfy a structural property called the *twins property*.

In this chapter we extend that work to deal with wrtgs. We will present an algorithm for determinizing wrtgs that, like Mohri's algorithm, provides a correct result if it terminates and that terminates both for acyclic wrtgs over the probability and tropical semirings and for cyclic wrtgs satisfying an analogously defined twins property. We apply this algorithm to wrtgs representing vast forests of potential outputs generated by machine translation and parsing systems. We then use a variant of the k-best algorithm of Huang and Chiang [59] to obtain lists of trees from the forests, both before and after determinization, and demonstrate that applying the determinization algorithm improves our results.

3.2 Related work

Comon et al. [27] show the determinization of unweighted finite-state tree automata, and prove its correctness. Borchardt and Vogler [14] present determinization of wtas with a different method than the one we present here. Like our method, their method returns a correct result if it terminates. However, their method requires the implementor to specify an equivalence relation of trees; if such a relation has a finite number of equivalence classes, then their method terminates ([13], Cor. 5.1.8). We consider our method more practical than theirs because we require no such explicit specification, and because when their method is applied to an acyclic wta, the resulting wta has a size on the order of the number of derivations represented in the wta. Our method has the same liability in the worst case, but in practice rarely exhibits this behavior.

3.3 Practical determinization

We recall basic definitions from Chapter 2, particularly that of semirings (Section 2.1.2), tree series (Section 2.1.3), and wrtgs (Section 2.2). We introduce some more semiring notions (cf. [17], Section 2). Let $\mathbb{W}$ be a semiring, and let w_1 and w_2 be values in $\mathbb{W}$. We say $\mathbb{W}$ is *zero-divisor free* if $w_1 \cdot w_2 = 0$ implies that $w_1 = 0$ or $w_2 = 0$, $\mathbb{W}$ is *zero-sum free* if $w_1 + w_2 = 0$ implies that $w_1 = w_2 = 0$, and $\mathbb{W}$ is a *semifield* if it admits multiplicative inverses, i.e., for every $w \in \mathbb{W} \setminus \{0\}$ there is a uniquely determined $w^{-1} \in \mathbb{W}$ such that $w \cdot w^{-1} = 1$. Note that the probability semiring is a zero-divisor free and zero-sum free semifield. The tropical semiring is also a zero-divisor free and zero-sum free semifield,

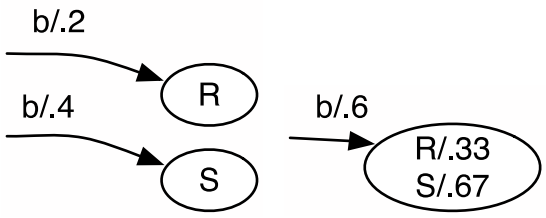
if we disallow the $*$ operator and consequently remove the value $-\infty$ from its carrier set. For the probability semiring, $w^{-1} = 1/w$ and for the tropical semiring, $w^{-1} = -w$.

The determinization algorithm is presented as Algorithm 15. It takes as input a chain production-free and normal form wrtg G_{in} over a zero-sum free and zero-divisor free semifield $\mathbb{W}$, and returns as output a deterministic wrtg G_{out} such that $L_{G_{\text{in}}} = L_{G_{\text{out}}}$. Like the algorithm of Mohri [99], this algorithm is correct if it terminates, and will terminate for acyclic wrtgs. It may terminate on some cyclic wrtgs, as described in the joint work portion of Section 3.4, but we do not otherwise consider these cases in this chapter.

Determinizing a wrtg can be thought of as a two-stage process. First, the structure of the wrtg must be determined such that a single derivation exists for each recognized input tree. This is achieved by a classic powerset construction, i.e., a nonterminal must be constructed in the output wrtg that represents all the possible reachable destination nonterminals given an input and a label. Note that the structure of Algorithm 15 is very similar to classical (unweighted) determinization, as presented in Algorithm 5.

In the second stage we deal with weights. For this we will use Mohri [99]'s concept of the *residual weight*.[2] We represent in the construction of nonterminals in the output wrtg not only a subset of nonterminals of the input wrtg, but also a value associated with each of these nonterminals, called the residual. Since we want the weight of the derivation of each tree in G_{out} to be equal to the sum of the weights of all derivations of that tree in G_{in}, we replace a set of productions in G_{in} that have the same right side with a single production in G_{out} bearing the label and the sum of the weights of the productions. The left nonterminal of the production in G_{out} represents the left nonterminals in each of the

[2]For ease of intuitive understanding we describe weights in this section over the probability semiring. However, any semiring matching the conditions specified in Algorithm 15 will suffice.



(a) Before determinization (b) After determinization

Figure 3.3: Portion of the example wrtg from Figure 3.1 before and after determinization. Weights of similar productions are summed and nonterminal residuals indicate the proportion of weight due to each original nonterminal.

combined productions and, for each nonterminal, the proportion of the weight from the relevant original production.

Figure 3.3 shows the determinization of a portion of the example wrtg. Note that the production leading to nonterminal R in the input wrtg contributes 0.2, which is $\frac{1}{3}$ of the weight on the output wrtg production. The production leading to nonterminal S in the input wrtg contributes the remaining $\frac{2}{3}$. This is reflected in the nonterminal construction in the output wrtg. The complete determinization of the example wrtg is shown in Figure 3.1b.

Let us consider in more detail how productions in G_{out}, the output wrtg, are formed. It is illustrative to first describe the construction of terminal productions, i.e., those with right side labels in $\Sigma^{(0)}$. This construction is done in lines 10–16. To be deterministic, there can only be one terminal production for each unique terminal symbol α. And as determined in line 11, the weight of this production is the sum of the weight of all α-labeled productions in G_{in}. It remains to determine the left nonterminal of this production. In the analogous construction for unweighted determinization this would simply be a nonterminal representing the union of all left nonterminals of the α-labeled productions in G_{in}, as can be seen in line 11 of Algorithm 5. For the weighted case we

need to add a residual weight; to do this we simply calculate the fraction of the total weight contributed by each production and assign that value to that production's left nonterminal, in line 12. The (possibly) newly constructed nonterminal is then added to the set of nonterminals to be used in forming nonterminal productions.

The formation of nonterminal productions follows the same principle as that of terminal productions, but a sequence of descendant (i.e., right side) nonterminals and the contribution of that sequence to the production weight and the left nonterminal must be considered. The formation in lines 21–29 is a joint generalization of the formation in lines 20–26 for rtgs and that of lines 10–12 in Figure 10 of [99] for wsas. A nonterminal symbol σ and a sequence of nonterminals $\vec{\rho}$ is chosen in lines 21–22. To be deterministic, there can only be one nonterminal production in G_{out} with right side $\sigma(\vec{\rho})$. To calculate the weight of this production we consider each production in G_{in} that "fits" the selected symbol and sequence, as indicated in the elements under the summation of line 23. A production fits if it has the correct symbol σ and if each of the nonterminals in its descendant sequence has a non-zero residual weight in the corresponding position of $\vec{\rho}$. As calculated to the right of the summation symbol in line 23, every production that fits contributes the product of its weight and the residuals associated with its descendant nonterminals to the new production's weight. The contribution of each production is summed together to form the total new production weight, in line 24. As in the unweighted case, the left side nonterminal of the new production is calculated by taking the union of the left side nonterminals of all fitting productions. The residual for each nonterminal in this union is calculated by summing the individual contributions of fitting productions with left sides

equal to that nonterminal.[3] That fraction of the total production weight is the residual for that nonterminal, as seen in line 26.

Finally, to ensure G_{out} has a single initial nonterminal, an unused symbol is added to the set of nonterminals in G_{out}.[4] A chain production is created linking the new initial nonterminal to each nonterminal that has a non-zero residual associated with the initial nonterminal of G_{in}. The weight of this production is equal to the residual associated with the original initial nonterminal.[5] This construction is done in lines 15–16 and 30–31.

3.4 Determinization using factorizations

This section presents a proof of correctness of Algorithm 15. It is joint work with Matthias Büchse and Heiko Vogler, first published in [17]. The results are primarily due to the other two authors, but our presentation of the material differs significantly, in order to match our formalisms. In this section we present the idea of factorizations for weighted automata, introduced by Kirsten and Mäurer [72], and extend it to wrtgs. We then show that the construction of Algorithm 15 is indeed done via factorization. We also present the *initial algebra semantics*, a method of calculating the tree series represented by a wrtg that is different from that in Section 2.2, but equivalent in conclusion. This semantics is crucial in demonstrating language equivalence. Next we prove that if determinization via factorization terminates, it generates a wrtg with the same language as the input wrtg, thus proving the construction of Algorithm 15 is correct. Finally we describe conditions

[3] Unlike in the terminal case, there can be more than one such production for a given symbol.
[4] In fact, it is the same initial nonterminal from G_{in}, but clearly that symbol has not been used thus far.
[5] This corresponds to the calculation of final weights in Algorithm 10 of [99].

Algorithm 15 WEIGHTED-DETERMINIZE

inputs
  wrtg $G_{\text{in}} = (N_{\text{in}}, \Sigma, P_{\text{in}}, n_0)$ over zero-sum free and zero-divisor free semifield $\mathbb{W}$ in normal form with no chain productions
outputs
  deterministic wrtg $G_{\text{out}} = (N_{\text{out}} \subseteq (\mathcal{P}(N \times \mathbb{W}) \cup n_0), \Sigma, P_{\text{out}}, n_0)$ over $\mathbb{W}$ in normal form such that $L_{G_{\text{in}}} = L_{G_{\text{out}}}$.
complexity
  $O(|\Sigma| k^{\tilde{z}|supp(L_{G_{\text{in}}})|})$, where k is the highest rank in Σ and $\tilde{z}$ is the size of the largest tree in $supp(L_{G_{\text{in}}})$, if $L_{G_{\text{in}}}$ is finite.

$P_{\text{out}} \leftarrow \emptyset$
$\Xi \leftarrow \emptyset$ {Seen nonterminals}
$\Psi \leftarrow \emptyset$ {New nonterminals}
for all $\alpha \in \Sigma^{(0)}$ **do**
  $w_{total} = \bigoplus_{n \xrightarrow{w} \alpha \in P_{\text{in}}} w$
  $\rho_{dst} \leftarrow \{(n, w \cdot w_{total}^{-1}) \mid n \xrightarrow{w} \alpha \in P_{\text{in}}\}$
  $\Psi \leftarrow \Psi \cup \{\rho_{dst}\}$
  $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{\rho_{dst} \xrightarrow{w_{total}} \alpha\}$
  if $(n_0, w) \in \rho_{dst}$ for some $w \in \mathbb{W}$ **then**
    $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{n_0 \xrightarrow{w} \rho_{dst}\}$
while $\Psi \neq \emptyset$ **do**
  $\rho_{new} \leftarrow$ any element of Ψ
  $\Xi \leftarrow \Xi \cup \{\rho_{new}\}$
  $\Psi \leftarrow \Psi \setminus \{\rho_{new}\}$
  for all $\sigma^{(k)} \in \Sigma \setminus \Sigma^{(0)}$ **do**
    for all $\vec{\rho} = \rho_1 \dots \rho_k \mid \rho_1 \dots \rho_k \in \Xi^k, \rho_i = \rho_{new}$ for some $1 \leq i \leq k$ **do**
      $\phi \leftarrow \{(n, \bigoplus_{n \xrightarrow{w} \sigma(n_1,\dots,n_k) \in P_{\text{in}}^{(n)}, (n_1,w_1) \in \rho_1, \dots, (n_k,w_k) \in \rho_k} w \cdot \prod_{i=1}^{k} w_i) \mid P_{\text{in}}^{(n)} \neq \emptyset\}$
      $w_{total} = \bigoplus_{(n,w) \in \phi} w$
      if $w_{total} \neq 0$ **then**
        $\rho_{dst} \leftarrow \{(n, w \cdot w_{total}^{-1}) \mid (n, w) \in \phi\}$
        if $\rho_{dst} \notin \Xi$ **then**
          $\Psi \leftarrow \Psi \cup \{\rho_{dst}\}$
        $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{\rho_{dst} \xrightarrow{w_{total}} \sigma(\vec{\rho})\}$
        if $(n_0, w) \in \rho_{dst}$ for some $w \in \mathbb{W}$ **then**
          $P_{\text{out}} \leftarrow P_{\text{out}} \cup \{n_0 \xrightarrow{w} \rho_{dst}\}$
$N_{\text{out}} \leftarrow \Xi \cup \{n_0\}$
return G_{out}

for which the algorithm terminates, and note that one of these applies to the experiments presented in Section 3.5.

3.4.1 Factorization

We begin with a definition of factorization, an abstract algebraic structure, and then describe how it relates to the determinization construction. We recall the typical notion of vectors, and for some vector s we write s_i to indicate the member of s associated with $i \in I$, where I is an index set for s. We write a finite vector s with members a, b, c as $\langle a, b, c \rangle$. When the indices are not clear (typically when $I \neq \mathbb{N}$) and are, e.g., i_1, i_2, i_3 we denote their association with values as $\langle i_1 = a, i_2 = b, i_3 = c \rangle$. We extend the definition of the semiring multiplication $\cdot$ such that $w \cdot s$, where w is a value of $\mathbb{W}$ and s is a vector of values of $\mathbb{W}$, is equal to the magnification of s by w. For example, if $s = \langle a, b, c \rangle$, then $w \cdot s = \langle w \cdot a, w \cdot b, w \cdot c \rangle$.

Definition 3.4.1 (cf. [17]) Let A be a nonempty finite set. A pair (f, g) is a *factorization (of dimension A over semiring $\mathbb{W}$)* if $f : \mathbb{W}^A \setminus \{0^A\} \to \mathbb{W}^A$, $g : \mathbb{W}^A \setminus \{0^A\} \to \mathbb{W}$, and $u = g(u) \cdot f(u)$ for every $u \in \mathbb{W}^A \setminus \{0^A\}$, where $\mathbb{W}^A$ denotes a vector of values of $\mathbb{W}$ indexed on A, and 0^A denotes the vector indexed on A where all values are 0. A factorization is *maximal* if for every $u \in \mathbb{W}^A$ and $w \in \mathbb{W}$, $w \cdot u \neq 0^A$ implies $f(u) = f(w \cdot u)$. There is a uniquely defined *trivial* factorization (f, g) where $f(u) = u$ and $g(u) = 1$ for every $u \in \mathbb{W}^A \setminus \{0^A\}$.

We now introduce a particular choice for (f, g). We will first show that this particular factorization is a maximal factorization, and then we will show that this factorization is the factorization used in Algorithm 15.

Lemma 3.4.2 (cf. [17], Lemma 4.2) Let A be a nonempty finite set, let $\mathbb{W}$ be a zero-sum free semifield, and define $f(u)$ and $g(u)$ for every $u \in \mathbb{W}^A \setminus \{0^A\}$, such that $g(u) = \bigoplus_{a \in A} u_a$ and $f(u) = g(u)^{-1} \cdot u$. (f, g) is a maximal factorization.

Proof We show that (f, g) is a factorization. Let $u \in \mathbb{W}^A \setminus \{0^A\}$. Since $\mathbb{W}$ is zero-sum free, $g(u) \neq 0$ and hence, $g(u) \cdot f(u) = g(u) \cdot g(u)^{-1} \cdot u = u$. We show that (f, g) is maximal. Let $w \in \mathbb{W}$ such that $w \cdot u \neq 0^A$. Moreover, let $a \in A$. Then

$$\begin{aligned}
[f(w \cdot u)]_a &= \left[g(w \cdot u)^{-1} \cdot w \cdot u\right]_a \\
&= \left(\bigoplus_{a' \in A} w \cdot u_{a'}\right)^{-1} \cdot w \cdot u_a \\
&= \left(w \cdot \bigoplus_{a' \in A} u_{a'}\right)^{-1} \cdot w \cdot u_a \\
&= \left(\bigoplus_{a' \in A} u_{a'}\right)^{-1} \cdot w^{-1} \cdot w \cdot u_a \\
&= g(u)^{-1} \cdot u_a \\
&= [f(u)]_a \ .
\end{aligned}$$

■

N_{out}, the nonterminal set of G_{out}, the result of Algorithm 15, consists of elements that are subsets of N_{in}, the original nonterminal set, paired with nonzero values from $\mathbb{W}$.[6] We can thus interpret some $n \in N_{\text{out}}$ as a member of $\mathbb{W}^{N_{\text{in}}}$, where nonterminals not appearing in the original formulation of n take on the value 0 in this vector formulation.

Observation 3.4.3 Consider the mapping ϕ constructed in line 23 of Algorithm 15. The weight of newly constructed productions in P_{out}, in line 29 is $g(\phi)$ and the left side

[6] It also consists of the special start nonterminal n_0, but we do not consider that item in the current discussion.

nonterminal ρ_{dst}, formed in line 26, is $f(\phi)$. Analogous constructions are done in lines 11 and 12 for terminal productions, though ϕ is not explicitly represented.

3.4.2 Initial algebra semantics

In Section 2.2 we presented a semantics for wrtgs based on derivations that allows us, given a wrtg $G = (N, \Sigma, P, n_0)$, to calculate the tree series L_G. According to the semantics presented in Section 2.2, the weight of a tree $t \in T_\Sigma$ is the sum of the weights of all derivations from n_0 to t using productions in P.

Here we present another semantics commonly used in the study of tree automata, called the *initial algebra semantics* [45]. This semantics calculates the weight of a tree recursively, as a function of the weights of its subtrees. In order to define this semantics for wrtgs without significant modification from the definition for wtas, we limit the applicability to normal form, mostly chain production-free wrtgs. The only chain productions allowed are those used to simulate the "final weights" used in typical definitions of wta (e.g., that of Fülöp and Vogler [45]). We consider them to be in a set P_{chain} distinct from P and to all have the form $n_0 \xrightarrow{w} n$, where $n \neq n_0$. Furthermore, if $P_{chain} \neq \emptyset$ then no $p \in P$ has n_0 as its left nonterminal. If $P_{chain} = \emptyset$, this is equivalent to a final weight of 1 for n_0 and 0 for all others. If $P_{chain} \neq \emptyset$, this is equivalent to a final weight of w for each n such that $n_0 \xrightarrow{w} n \in P_{chain}$ and 0 for all others.

We have previously considered the productions P of a wrtg as a set or pair (see Section 2.2). In the initial algebra semantics we consider P a family $(P_k \mid k \in \mathbb{N})$ of mappings, $P_k : \Sigma^{(k)} \to \mathbb{W}^{N^k \times N}$. Thus, if P contains two productions $n_1 \xrightarrow{w_1} \sigma(n_2, n_3, n_4)$ and $n_5 \xrightarrow{w_2} \sigma(n_2, n_3, n_4)$ (assuming $\sigma \in \Sigma^{(3)}$), we can write $P_3(\sigma)_{n_2 n_3 n_4, n_1} = w_1$ and

$P_3(\sigma)_{n_2n_3n_4} = \langle n_1 = w_1, n_2 = 0, n_3 = 0, n_4 = 0, n_5 = w_2\rangle$. Since the particular mapping is obvious given the state sequence subscript, we omit it, e.g., we write $P(\sigma)_{n_2n_3n_4,n_1}$ instead of $P_3(\sigma)_{n_2n_3n_4,n_1}$. We now denote the weight of deriving a tree t from nonterminal n using P [7] by $h_P(t)_n$. Then $h_P(t) \in \mathbb{W}^N$.

Let $G = (N, \Sigma, P \cup P_{chain}, n_0)$ be in normal form. Then μ_P is a family $(\mu_P(\sigma) \mid \sigma \in \Sigma)$ of mappings where for every $k \in \mathbb{N}$ and $\sigma \in \Sigma^{(k)}$ we have $\mu_P(\sigma) : \underbrace{\mathbb{W}^N \times \ldots \times \mathbb{W}^N}_{k} \to \mathbb{W}^N$ and for every $s_1, \ldots, s_k \in \mathbb{W}^N$

$$\mu_P(\sigma)(s_1, \ldots, s_k)_n = \bigoplus_{(n_1,\ldots,n_k)\in N^k} (s_1)_{n_1} \cdot \ldots \cdot (s_k)_{n_k} \cdot P_k(\sigma)_{n_1\ldots n_k,n} \ .$$

For a tree $t = \sigma(t_1, \ldots, t_k)$, $h_P(t) = \mu_P(\sigma)(h_P(t_1), \ldots, h_P(t_k))$. Additionally, if $P_{chain} \neq \emptyset$, $\mu_{P_{chain}}$ is a mapping $N \setminus \{n_0\} \to \mathbb{W}$. If $P_{chain} = \emptyset$, then $h_P(t)_{n_0}$ is the value of t in L_G. Otherwise, the value is $\bigoplus_{n\in N\setminus\{n_0\}} h_P(t)_n \cdot \mu_{P_{chain}}(n)$. We finally note that the semantics in Section 2.2, when applied to the class of wrtgs discussed here, is equivalent to the traditional *run semantics*, and that the run and initial algebra semantics are proven equivalent in Section 3.2 of Fülöp and Vogler [45].

Observation 3.4.4 Consider $\phi \in \mathbb{W}^{N_{\text{in}}}$ constructed in line 23 of Algorithm 15 for $\sigma \in \Sigma^{(k)}$ and $\vec{\rho} \in (\mathbb{W}^{N_{\text{in}}})^k$. Clearly, $\phi = \mu_P(\sigma)(\vec{\rho})$.

3.4.3 Correctness

Before we prove the correctness of Algorithm 15 we need to note a particular property of deterministic wrtgs. The following lemma shows that there is at most one nonterminal

[7] which is equivalent to the sum of all derivations from n to t

that can begin a chain production-free derivation of a tree in a deterministic normal-form wrtg.

Lemma 3.4.5 Let $G = (N, \Sigma, P \cup P_{chain}, n_0)$ be a deterministic normal-form wrtg over $\mathbb{W}$ and let $t \in T_\Sigma$. There is at most one $n \in N$ such that $n \Rightarrow^* t$ using P and at most one derivation d from n to t using P.

Proof We prove by induction on $size(t)$. Let $size(t) = 1$. Thus, t has the form α, where $\alpha \in \Sigma^{(0)}$. Assume there are $n, n' \in N$ such that $n \Rightarrow^* \alpha$ and $n' \Rightarrow^* \alpha$ using P. Since the size is 1, $n \Rightarrow^p \alpha$ and $n' \Rightarrow^{p'} \alpha$ for some $p, p' \in P$. By definition, $p = n \xrightarrow{w} \alpha$ and $p' = n' \xrightarrow{w'} \alpha$ for some nonzero $w, w' \in \mathbb{W}$. But then G is not deterministic, so our assumption must be false. Clearly, if there is one nonterminal n such that $n \Rightarrow^p \alpha$, then the single derivation d is (p).

Now assume the lemma is true for trees of size i or smaller. Let t be a tree of size $i+1$. Then, t has the form $\sigma(t_1, \ldots, t_k)$ where $\sigma \in \Sigma^{(k)}$ and $size(t_j) \leq i, 1 \leq j \leq k$. By the induction hypothesis either there exist unique $n_1, \ldots, n_k \in N$ such that $n_j \Rightarrow^* t_j$ using P for $1 \leq j \leq k$, or at least one such n_j does not exist. In the latter case, clearly there is no $n \in N$ such that $n \Rightarrow^* t$ using P. For the former case, this means that $\sigma(n_1, \ldots, n_k) \Rightarrow^* t$ using P. Let $d_1, \ldots, d_k$ be the single derivations of, respectively, $t_1, \ldots, t_k$. Then $(d_1 \ldots d_k)$ is clearly the single derivation from $\sigma(n_1, \ldots, n_k)$ to t, that notation implying a concatenation of the productions in the derivation of each subtree. Then, assume there are $n, n' \in N$ such that $n \Rightarrow^* t$ and $n' \Rightarrow^* t$ using P. Because G is in normal form and because of the uniqueness of $n_1, \ldots, n_k$, $n \Rightarrow^p \sigma(n_1, \ldots, n_k) \Rightarrow^* t$ and $n' \Rightarrow^{p'} \sigma(n_1, \ldots, n_k) \Rightarrow^* t$ for some $p, p' \in P$. By definition, $p = n \xrightarrow{w} \sigma(n_1, \ldots, n_k)$ and $p' = n' \xrightarrow{w'} \sigma(n_1, \ldots, n_k)$ for some nonzero $w, w' \in \mathbb{W}$.

But then G is not deterministic, so our assumption must be false. Then, if there is a single nonterminal n such that $n \Rightarrow^p \sigma(n_1, \ldots, n_k) \Rightarrow^* t$, then the single derivation d is $(pd_1 \ldots d_k)$.

■

We break the proof of correctness of Algorithm 15 into an initial lemma that does most of the work, and the theorem, which finishes the proof. These two components are very related to Theorem 5.3 of [17], though the structure is somewhat different.

Lemma 3.4.6 (cf. [17], Thm. 5.3) Let $G_{\text{in}} = (N_{\text{in}}, \Sigma, P_{\text{in}}, n_0)$ be the input to Algorithm 15, let the algorithm terminate on G, and let $G_{\text{out}} = (N_{\text{out}}, \Sigma, P_{\text{out}} \cup P_{\text{chain}}, n_0)$ be the output. For every $t \in T_\Sigma$ and $n \in N_{\text{in}}$, $h_{P_{\text{in}}}(t) = \bigoplus_{n' \in N_{\text{out}}} h_{P_{\text{out}}}(t)_{n'} \cdot n'$.

Proof We immediately note that by Lemma 3.4.5 we can rewrite the conclusions of this lemma as "$h_{P_{\text{in}}}(t) = h_{P_{\text{out}}}(t)_{n'} \cdot n'$ if there is some $n' \in N_{\text{out}}$ such that $h_{P_{\text{out}}}(t)_{n'} \neq 0$, or 0 otherwise."

We will prove the lemma by induction on the size of t. Let t be of size 1. Then t has the form α, where $\alpha \in \Sigma^{(0)}$. According to line 14 of Algorithm 15, if there is at least one production with right side α in P_{in}, there is a single production $p' = n' \xrightarrow{w_{total}} \alpha$ in P_{out}, where w_{total} is nonzero. If p' does not exist, then $h_{P_{\text{in}}}(t)_n = 0$ for any selection of n, so the statement is true. If p' does exist, then $h_{P_{\text{out}}}(t)_{n'} = w_{total}$. Now, for any $n \in N_{\text{in}}$, $h_{P_{\text{in}}}(t)_n = P_{\text{in}}(\alpha)_{\epsilon,n}$. If there is some $p \in P_{\text{in}}$ of the form $n \xrightarrow{w} \alpha$ this value is w, or else it is 0. As indicated on line 12, in the former case, the weight of n'_n is $w \cdot w^{-1}_{total'}$, and in the latter case it is 0. Thus, if $h_{P_{\text{in}}}(t)_n = w$, $h_{P_{\text{out}}}(t)_{n'} = w_{total} \cdot w \cdot w^{-1}_{total} = w$ and otherwise both sides are 0.

Now assume the lemma is true for trees of size i or smaller. Let t be a tree of size $i+1$. Then, t has the form $\sigma(t_1, \ldots, t_k)$ where $\sigma \in \Sigma^{(k)}$ and $size(t_j) \leq i$, $1 \leq j \leq k$. By Lemma 3.4.5 there are uniquely defined $n'_1, \ldots, n'_k \in N_{\text{out}}$ such that for $1 \leq i \leq k$, $h_{P_{\text{out}}}(t_i)_{n'_i} \neq 0$.

$$
\begin{aligned}
h_{P_{\text{in}}}(t) &= \mu_{P_{\text{in}}}(\sigma)(h_{P_{\text{in}}}(t_1), \ldots, h_{P_{\text{in}}}(t_k)) && \text{(definition of semantics)} \\
&= \mu_{P_{\text{in}}}(\sigma)(h_{P_{\text{out}}}(t_1)_{n'_1} \cdot n'_1, \ldots, h_{P_{\text{out}}}(t_k)_{n'_k} \cdot n'_k) && \text{(induction hypothesis)} \\
&= h_{P_{\text{out}}}(t_1)_{n'_1} \cdot \ldots \cdot h_{P_{\text{out}}}(t_k)_{n'_k} \cdot \mu_{P_{\text{in}}}(\sigma)(n'_1, \ldots, n'_k) && \text{(commutativity)}
\end{aligned}
$$

Either $h_{P_{\text{in}}}(t) = 0^{N_{\text{in}}}$ or it does not. In the former case, since $\prod_{j=1}^{k} h_{P_{\text{out}}}(t_j)_{n'_j}$ is not 0,[8] it must be that $\mu_{P_{\text{in}}}(\sigma)(n'_1, \ldots, n'_k) = 0$. If the sequence $n'_1, \ldots, n'_k$ is chosen as $\vec{\rho}$ on line 22 of the algorithm then the value for w_{total} set on line 24 is 0, and thus $P_{\text{out}}(\sigma)_{n'_1 \ldots n'_k} = 0^{N_{\text{out}}}$. Thus, $h_{P_{\text{out}}}(t) = 0^{N_{\text{out}}}$.

In the latter case, consider line 26, in which ρ_{dst} is calculated as $f(\mu_{P_{\text{in}}}(\sigma)(n'_1, \ldots, n'_k))$. Note further that on line 29, $g(\mu_{P_{\text{in}}}(\sigma)(n'_1, \ldots, n'_k))$ is chosen as the weight of the newly produced production, thus $P_{\text{out}}(\sigma)_{s', \rho_{dst}} = g(\mu_{P_{\text{in}}}(\sigma)(\vec{\rho}))$. We continue the derivation from above:

$$
\begin{aligned}
h_{P_{\text{in}}}(t) &= h_{P_{\text{out}}}(t_1)_{n'_1} \cdot \ldots \cdot h_{P_{\text{out}}}(t_k)_{n'_k} \cdot \mu_{P_{\text{in}}}(\sigma)(n'_1, \ldots, n'_k) && \text{(from above)} \\
&= h_{P_{\text{out}}}(t_1)_{n'_1} \cdot \ldots \cdot h_{P_{\text{out}}}(t_k)_{n'_k} \cdot g(\mu_{P_{\text{in}}}(\sigma)(n'_1, \ldots, n'_k)) \cdot \vec{\rho} && \text{(definition of f and g)} \\
&= h_{P_{\text{out}}}(t)_{\vec{\rho}} \cdot \vec{\rho} && \text{(line 29 and definition)}
\end{aligned}
$$

■

[8] No term in that product is 0 and $\mathbb{W}$ is zero divisor-free

We are finally ready to show language equivalence, which is now a simple matter. We note that $L_{G_{\text{in}}} = L_{G_{\text{out}}}$ iff for any $t \in T_\Sigma$, $h_{P_{\text{in}}}(t)_{n_0} = h_{P_{\text{out}} \cup P_{\text{chain}}}(t)_{n_0}$.

Theorem 3.4.7 Let $G_{\text{in}} = (N_{\text{in}}, \Sigma, P_{\text{in}}, n_0)$ be the input to Algorithm 15, let the algorithm terminate on G_{in}, and let $G_{\text{out}} = (N_{\text{out}}, \Sigma, P_{\text{out}} \cup P_{\text{chain}}, n_0)$ be the output. For every $t \in T_\Sigma$, $h_{P_{\text{in}}}(t)_{n_0} = h_{P_{\text{out}} \cup P_{\text{chain}}}(t)_{n_0}$.

Proof If there is no unique $n' \in N_{\text{out}}$ such that $h_{P_{\text{out}}}(t)_{n'}$ is nonzero then by Lemma 3.4.6 and by definition, $h_{P_{\text{in}}}(t)_{n_0} = h_{P_{\text{out}} \cup P_{\text{chain}}}(t)_{n_0} = 0$. If $P_{\text{chain}} = \emptyset$, then for all $n' \in N_{\text{out}} \setminus \{n_0\}$, $n'_{n_0} = 0$, since otherwise some chain production would be added in lines 16 or 31 of the algorithm. Then, again, by Lemma 3.4.6 and by definition, $h_{P_{\text{in}}}(t)_{n_0} = h_{P_{\text{out}} \cup P_{\text{chain}}}(t)_{n_0} = 0$. The remaining case assumes that there is a unique $n' \in N_{\text{out}}$ such that $h_{P_{\text{out}}}(t)_{n'}$ is nonzero and that P_{chain} is nonempty.

$$
\begin{aligned}
h_{P_{\text{in}}}(t)_{n_0} &= h_{P_{\text{out}}}(t)_{n'} \cdot n'_{n_0} && \text{(Lemma 3.4.6)} \\
&= h_{P_{\text{out}}}(t)_{n'} \cdot P_{\text{chain}}(n') \\
&= h_{P_{\text{out}} \cup P_{\text{chain}}}(t)_{n_0}
\end{aligned}
$$

■

3.4.4 Termination

We have shown, in Theorem 3.4.7, that Algorithm 15 is correct if it terminates. Since, even if it terminates, the algorithm's runtime is in the worst case exponential, to an engineer the conditions for termination are not as useful as the conditions for terminating with

a specified time. In fact, we have followed that tactic in our implementation of the algorithm by simply providing users with a maximum time to wait before considering determinization not possible for a given input; see Chapter 6 for details. However, it is theoretically interesting and can be generally useful to know certain conditions for termination. In this section we prove that Algorithm 15 terminates on acyclic normal form chain production-free wrtgs. This was also proven in [17] but we use a different approach. We briefly note that Algorithm 15 terminates when $\mathbb{W}$ has a finite carrier set, echoing a similar statement in [17]. And, we describe the main result of [17], sufficient conditions for determinization of cyclic wrtgs to terminate. The proof of this result is beyond the scope of this thesis, so we only outline the conditions and refer the reader to [17] for the details.

A wrtg $G = (N, \Sigma, P, n_0)$ is *cyclic* if, for some $n \in N$ and $t \in T_\Sigma(N)$ such that $n \in ydset(t)$, $n \Rightarrow^* t$. It should be clear that for an acyclic normal form chain production-free wrtg G, $L_G(t) = 0$ for all $t \in T_\Sigma$ such that $height(t) > |N|$. Since for any $k \in \mathbb{N}$, the set of trees in T_Σ of height k is finite, $supp(L_G)$ is finite.

Theorem 3.4.8 Algorithm 15 terminates on acyclic input.

Proof As can be seen from the structure of Algorithm 15, the only way it can fail to terminate is if the loop at line 17 is always satisfied, i.e., if the nonterminal set is not finite. As noted above, an acyclic wrtg implies finite support. Then, consider the means by which nonterminals of G_{out} are formed. At most $|\Sigma^{(0)}|$ are added at line 14. The only other addition of nonterminals comes at line 28. As noted in Observation 3.4.3, a nonterminal is formed as a function of the ϕ created in line 23. There is thus at most one nonterminal

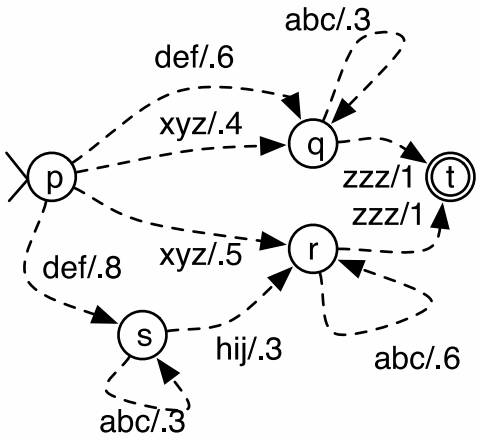


Figure 3.4: Sketch of a wsa that does not have the twins property. The dashed arcs are meant to signify a path between states, not necessarily a single arc. q and r are siblings because they can both be reached from p with a path reading "xyz", but are not twins, because the cycles from q and r reading "abc" have different weights. s and r are not siblings because they cannot be both reached from p with a path reading the same string. q and s are siblings because they can both be reached from p with a path reading "def" and they are twins because the cycle from both states reading "abc" has the same weight (and they share no other cycles reading the same string). Since q and r are siblings but not twins, this wsa does not have the twins property.

for every unique ϕ that can be formed in this algorithm. As noted in Observation 3.4.4, $\phi = \mu_P(\sigma)(\vec{\rho})$. If, for a given $\sigma \in \Sigma^{(k)}$, there is a finite choice of $\vec{\rho}$ that produces $\phi \neq 0^{N_{\text{out}}}$, then N_{out} is finite. By Lemma 3.4.5, for any tree $t \in T_\Sigma$ there is at most one $n' \in N_{\text{out}}$ such that $h_{P_{\text{out}}}(t) \neq 0^{N_{\text{out}}}$. Thus the size of N_{out} is at most the number of unique subtrees in $supp(L_G)$. If we let $\tilde{z}$ be the size of the largest tree in $supp(L_G)$, then there are at most $k^{\tilde{z}|supp(L_G)|}$ choices for $\vec{\rho}$ that produce $\phi \neq 0^{N_{\text{out}}}$. ∎

We note that if $\mathbb{W}$ has a finite carrier set, i.e., finitely many possible value, then clearly Algorithm 15 terminates, since $|N_{\text{out}}| = |\mathbb{W}||N_{\text{in}}| + 1$, where $|\mathbb{W}|$ is the cardinality of the carrier set.

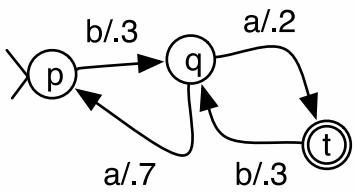


Figure 3.5: A wsa that is not cycle-unambiguous. The state q has two cycles reading the string "ab" with two different weights.

Finally, we present sufficient conditions for termination of cyclic wrtgs, the proof of which is the main result of [17]. As noted previously, the proof of these termination conditions is beyond the scope of this work.

A cyclic wrtg G over $\mathbb{W}$ is determinizable by Algorithm 15 if $\mathbb{W}$ is extremal and if G has the twins property ([17], Theorem 5.2). A semiring is *extremal* if for every $w, w' \in \mathbb{W}$, $w + w' \in \{w, w'\}$. For example, the tropical semiring is extremal.

To introduce the twins property, let us first consider the analogous concept for wsas. Two states in a wsa that can be reached from the start state reading string e are *siblings*, and two siblings q and q' are *twins* if for every string f such that there is a cycle at q and at q' reading f, the weights of these cycles are the same. A wsa has the *twins property* if all siblings in the wsa are twins. Figure 3.4 is a sketch of a wsa demonstrating sibling and twin states. It was shown by Mohri [99] that cyclic wsa are determinizable if they have the twins property. Furthermore, Allauzen and Mohri [2] showed that the twins test is decidable if a wsa is *cycle-unambiguous*, that is, if there is at most one cycle at a state reading the same string. Figure 3.5 sketches a wsa that is not cycle-unambiguous.

The concepts of twins and cycle-unambiguity are elevated to the tree case as follows: A wrtg $G = (N, \Sigma, P, n_0)$ has the twins property if, for every $n, n' \in N$, $t \in T_\Sigma$, and $u \in T_\Sigma(\{z\})$ where $z \notin (N \cup \Sigma)$ and $u(v) = z$ for exactly one $v \in pos(u)$, if $L_G(t)_n \neq 0$,

$L_G(t)_{n'} \neq 0$, $L_G(u[n]_v)_n \neq 0$, and $L_G(u[n']_v)_{n'} \neq 0$, then $L_G(u[n]_v)_n = L_G(u[n']_v)_{n'}$. The twins property for wrtgs is depicted in Figure 3.6. G is cycle-unambiguous if for any $n \in N$ and $u \in T_\Sigma(\{n\})$ where $u(v) = z$ for exactly one $v \in pos(u)$, there is at most one derivation from n to u. Büchse et al. [18] showed, in Theorem 5.17, that the twins property for cycle-unambiguous normal-form and chain-production free wrtgs over a commutative zero-sum-free and zero-divisor-free is decidable.

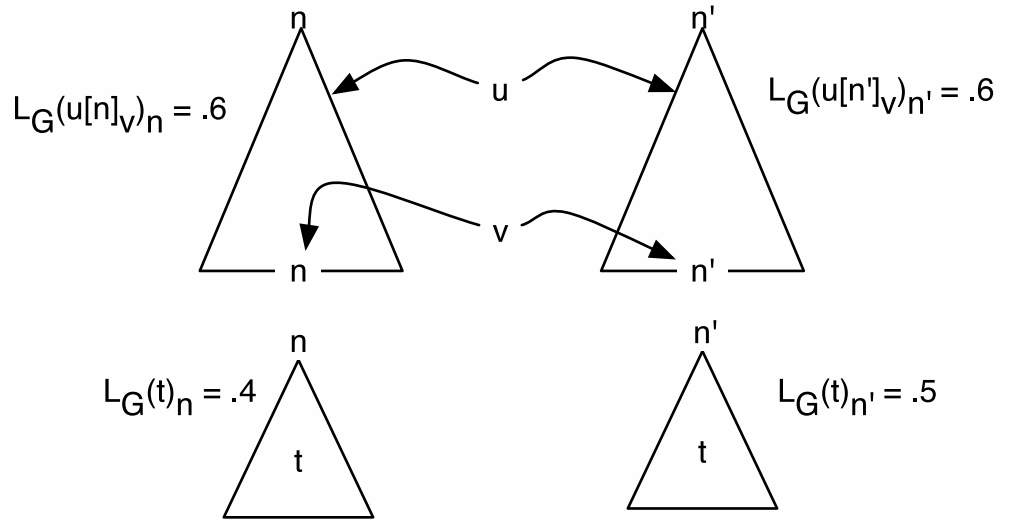


Figure 3.6: Demonstration of the twins test for wrtgs. If there are non-zero derivations of a tree t for nonterminals n and n′, and if the weight of the sum of derivations from n of u substituted at v with n is equal to the weight of the sum of derivations from n′ of u substituted at v with n′ for all u where these weights are nonzero for both cases, then n and n′ are twins.

3.5 Empirical studies

We now turn to some empirical studies. We examine the practical impact of the presented work by showing:

- That the multiple derivation problem is pervasive in practice and determinization is effective at removing duplicate trees.
- That duplication causes misleading weighting of individual trees and the summing achieved from weighted determinization corrects this error, leading to re-ordering of the k-best list.
- That weighted determinization positively affects end-to-end system performance.

We also compare our results to a commonly used technique for estimation of k-best lists, i.e., summing over the top $j > k$ derivations to get weight estimates of the top $m \leq j$ unique elements.

Method	BLEU
undeterminized	21.87
top-500 "crunching"	23.33
determinized	24.17

Table 3.1: BLEU results from string-to-tree machine translation of 116 short Chinese sentences with no language model. The use of best derivation (undeterminized), estimate of best tree (top-500), and true best tree (determinized) for selection of translation is shown.

3.5.1 Machine translation

We obtain packed-forest English outputs from 116 short Chinese sentences computed by a string-to-tree machine translation system based on that of Galley et al. [47]. The system is trained on all Chinese-English parallel data available from the Linguistic Data Consortium. The decoder for this system is a CKY algorithm that negotiates the space described by DeNeefe et al. [30]. No language model was used in this experiment.

The forests contain a median of 1.4×10^{12} English parse trees each. We remove cycles from each forest,[9] apply our determinization algorithm, and extract the *k*-best trees using a variant of the algorithm of Huang and Chiang [59]. The effects of weighted determinization on a *k*-best list are obvious to casual inspection. Figure 3.7 shows the improvement in quality of the top 10 trees from our example translation after the application of the determinization algorithm.

The improvement observed circumstantially holds up to quantitative analysis as well. The forests obtained by the determinized grammars have between 1.39% and 50% of the number of trees of their undeterminized counterparts. On average, the determinized forests contain 13.7% of the original number of trees. Since a determinized forest contains no repeated trees but contains exactly the same unique trees as its undeterminized counterpart, this indicates that an average of 86.3% of the trees in an undeterminized MT output forest are duplicates.

Weighted determinization also causes a surprisingly large amount of *k*-best reordering. In 77.6% of the translations, the tree regarded as "best" is different after determinization. This means that in a large majority of cases, the tree with the highest weight is not recognized as such in the undeterminized list because its weight is divided among its multiple derivations. Determinization allows these instances and their associated weights to combine and puts the highest weighted tree, not the highest weighted derivation, at the top of the list.

[9] As in Mohri [99], determinization may be applicable to some wrtgs that recognize infinite languages. In practice, cycles in forests of MT results are almost never desired, since these represent recursive insertion of words.

We can compare our method with the more commonly used methods of "crunching" j-best lists, where $j > k$. The duplicate sentences in the k trees are combined, hopefully resulting in at least k unique members with an estimation of the true tree weight for each unique tree. Our results indicate this is a rather crude estimation. When the top 500 derivations of the translations of our test corpus are summed, only 50.6% of them yield an estimated highest-weighted tree that is the same as the true highest-weighted tree.

As a measure of the effect weighted determinization and its consequential re-ordering has on an actual end-to-end evaluation, we obtain BLEU scores for our 1-best translations from determinization, and compare them with the 1-best translations from the undeterminized forest and the 1-best translations from the top-500 "crunching" method. The results are in Table 3.1. Note that in 26.7% of cases determinization did not terminate in a reasonable amount of time. For these sentences we used the best parse from top-500 estimation instead. It is not surprising that determinization may occasionally take a long time; even for a language of monadic trees (i.e., strings) the determinization algorithm is NP-complete, as implied by Casacuberta and de la Higuera [19] and, e.g., Dijkstra [32].

3.5.2 Data-Oriented Parsing

Determinization of wrtgs is also useful for parsing. Data-Oriented Parsing (DOP)'s methodology is to calculate weighted derivations, but as noted by Bod [11], it is the highest ranking parse, not derivation, that is desired. Since Sima'an [121] showed that finding the highest ranking parse is an NP-complete problem, it has been common to estimate the highest ranking parse by the previously described "crunching" method.

```
31.87:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
32.11:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(caused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
32.15:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VB(arouse) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
32.55:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VB(cause) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
32.60:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(attracted) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
33.16:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VB(provoke) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
33.27:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBG(causing) NP-C(NPB(DT(the) JJ(american)
NNS(protests)))) .(.))
33.29:  S(NP-C(NPB(DT(this) NN(case))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the) JJ(american)
NNS(protests))))) .(.))
33.31:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(aroused) NP-C(NPB(DT(the) NN(protest))
PP(IN(of) NP-C(NPB(DT(the) NNS(united_states)))))) .(.))
33.33:  S(NP-C(NPB(DT(this) NNS(cases))) VP(VBD(had) VP-C(VBN(incurred) NP-C(NPB(DT(the)
JJ(american) NNS(protests))))) .(.))
```

Figure 3.7: Ranked list of machine translation results with no repeated trees.

We create a DOP-like parsing model[10] by extracting and weighting a subset of subtrees from sections 2-21 of the Penn Treebank and use a DOP-style parser to generate packed forest representations of parses of the 2416 sentences of section 23. The forests contain a median of 2.5×10^{15} parse trees. We then remove cycles and apply weighted determinization to the forests. The number of trees in each determinized parse forest is reduced by a factor of between 2.1 and 1.7×10^{14}. On average, the number of trees is reduced by a factor of 900,000, demonstrating a much larger number of duplicate parses prior to determinization than in the machine translation experiment. The top-scoring parse after determinization is different from the top-scoring parse before determinization for 49.1% of the forests. When the determinization method is "approximated" by crunching the top-500 parses from the undeterminized list, only 55.9% of the top-scoring parses are the same. This indicates the crunching method is not a very good approximation of determinization. We use the standard F-measure combination of recall and precision to

[10]This parser acquires a small subset of subtrees, in contrast with DOP, and the beam search for this problem has not been optimized.

Method	Recall	Precision	F-measure
undeterminized	80.23	80.18	80.20
top-500 "crunching"	80.48	80.29	80.39
determinized	81.09	79.72	80.40

Table 3.2: Recall, precision, and F-measure results on DOP-style parsing of section 23 of the Penn Treebank. The use of best derivation (undeterminized), estimate of best tree (top-500), and true best tree (determinized) for selection of parse output is shown.

Experiment	Undeterminized	Determinized
machine translation	1.4×10^{12}	2.0×10^{11}
parsing	2.5×10^{15}	2.3×10^{10}

Table 3.3: Median trees per sentence forest in machine translation and parsing experiments before and after determinization is applied to the forests, removing duplicate trees.

score the top-scoring parse in each method against reference parses, and report results in Table 3.2. Note that in 16.9% of cases determinization did not terminate. For those sentences we used the best parse from top-500 estimation instead.

3.5.3 Conclusion

We presented a novel algorithm for practical determinization of wrtgs and, together with colleagues, proved that it is correct and that it terminates on acyclic wrtgs. We have shown that weighted determinization is useful for recovering k-best unique trees from a weighted forest. As summarized in Table 3.3, the number of repeated trees prior to determinization was typically very large, and thus determinization is critical to recovering true tree weight. We have improved evaluation scores by incorporating the presented algorithm into our MT work and we believe that other NLP researchers working with trees can similarly benefit from this algorithm. An implementation of this algorithm is available in the Tiburon tree automata toolkit (See Chapter 6).

Chapter 4

Inference through Cascades

In this chapter we present algorithms for forward and backward application of weighted tree-to-tree and tree-to-string transducers to wrtgs, trees, and strings. We discuss application of cascades of transducers and methods of efficient inference that perform an integrated search through a cascade. We also present algorithms for constructing derivation wrtgs for EM training of cascades of tree transducers, based on the application algorithms.

Although application is a well-established concept for both string and tree transducers, we do not believe explicit algorithms for conducting application have been previously presented, even for unweighted tree transducers. Current theoretical work is being done on application of weighted tree-to-tree transducers [43], but again, we are the first to describe these explicit algorithms. Our algorithm for backward application of tree-to-string transducers, while making use of classic parsing algorithms, is novel. So, too, is our extension of application to xwtt and our use of application to construct derivation wrtgs of cascades.

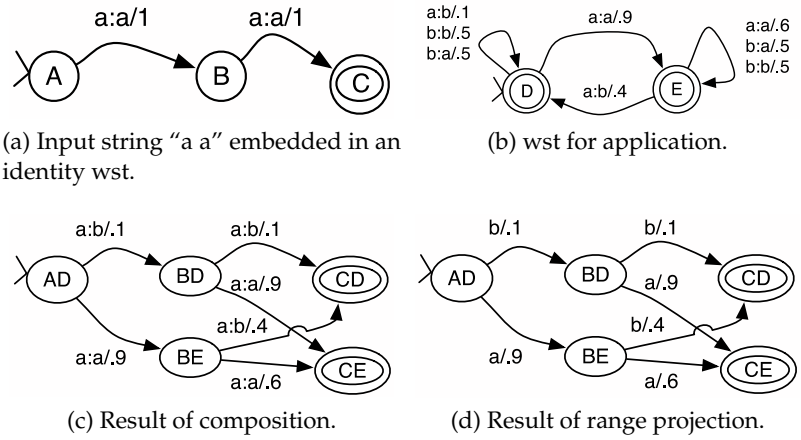
4.1 Motivation

We are interested in the result of transformation of some input by a transducer, called *application*. We'd like to do some inference on this result, such as determining the k-best transformations of the input or knowing all the paths that transform the input into some given output. When we are given the input and want to know how the transducer transforms it, we call this *forward application*. When we are given the output and want to know the possible inputs that cause the transducer to produce this output, we call this *backward application*. In this chapter we discuss forward and backward application algorithms.

We also consider a generalization of this problem. We want to divide up our problems into manageable chunks, each represented by a transducer. It is easier for designers to write several small transducers where each performs a simple transformation, rather than a single complicated transducer. We'd like to know, then, the result of transformation of input by a *cascade* of transducers, one operating after the other. As we will see, there are various ways of approaching this problem. We will consider *offline composition*, *bucket brigade application*, and *on-the-fly application*.

4.2 String case: application via composition

Before we discuss application of tree transducers, it is helpful to consider the solutions already known for application of string transducers. Imagine we have a string and a wst that can transform that string in a possibly infinite number of ways, each with some weight. We'd like to know the k highest-weighted of these outputs. If we could represent



(a) Input string "a a" embedded in an identity wst.

(b) wst for application.

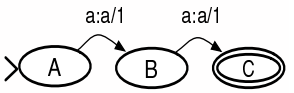
(c) Result of composition.

(d) Result of range projection.

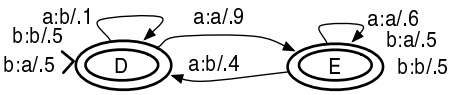
Figure 4.1: Application of a wst to a string.

these outputs as a wsa this would be an easy task, as there are well-known algorithms to efficiently find the k-best paths in a wsa [104]. Fortunately, we know it is in fact possible to find this wsa—wsts *preserve recognizability*, meaning given some weighted regular language[1] and some wst there exists some weighted regular language representing all the outputs. This language can be represented as a wsa, which we call the *application wsa*. The properties of wsts are such that, given several already-known algorithms, we can build application wsas without defining a new algorithm. Specifically, we will achieve application through a series of embedding, composition, and projection operations. As an aid in explanation, we provide a running example in Figure 4.1. Embedding is a trivial operation on strings and wsas: a string is embedded in a wsa by creating a single state and outgoing edge per symbol in the string. In turn, a wsa is embedded in an *identity wst* by, for every edge in a wsa with label a, forming an edge in the embedded wst with the same incoming state, outgoing state, and weight, but with both input and output labels a. In Figure 4.1a, the string "a a" has been embedded in a wst. Composition of wst is well

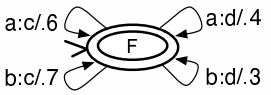
[1]The set of regular languages includes those languages containing a single string with weight 1.



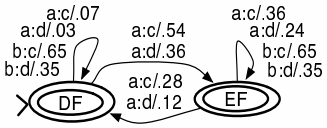
(a) Input string "a a" embedded in an identity wst.



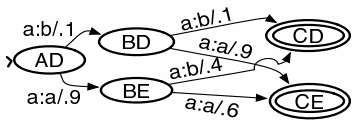
(b) First wst in cascade.



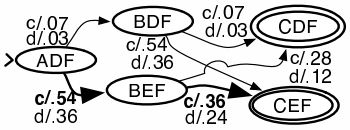
(c) Second wst in cascade.



(d) Offline composition approach: Compose the transducers.

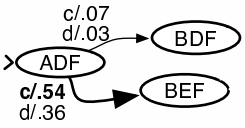


(e) Bucket brigade approach: Apply wst (b) to wst (a).

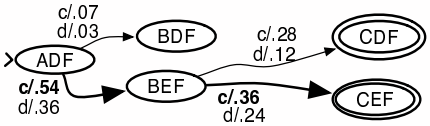


(f) Result of offline or bucket application after projection.

(g) Initial on-the-fly stand-in for (f).



(h) On-the-fly stand-in after exploring outgoing edges of state ADF.



(i) On-the-fly stand-in after best path has been found.

Figure 4.2: Three different approaches to application through cascades of wsts.

covered by, e.g., Mohri [101]. The result of the composition of the wsts of Figures 4.1a and 4.1b is shown in Figure 4.1c. Projecting a wst to a wsa is also trivial: to obtain a range projection, we ignore all the input labels in our wst, and to obtain a domain projection, we ignore all the output labels. The range projection of the transducer of Figure 4.1c is shown in Figure 4.1d. Figure 4.1d, then, depicts the result of the application of the transducer of Figure 4.1b to the string "a a". Note that we can also use this method to solve the reverse problem, where we are given an output string (or set of outputs) and we want the k-best of the set of inputs. To do so, we follow an analogous procedure, embedding the given output string or wsa in an identity wst, composing, and this time projecting the domain, and running k-best.

4.3 Extension to cascade of wsts

Now consider the case of an input wsa and a sequence of transducers. Our running example in Figure 4.2 again uses the string "a a" as input and a cascade comprising the transducer from Figure 4.1 (reproduced in Figure 4.2b) and a second transducer, depicted in Figure 4.2c. There are at least three ways to perform application through this cascade. Firstly, we can compose the sequence of transducers before even considering the given wsa, using the aforementioned composition algorithm. We will call this approach *offline composition*. The result of this composition is depicted in Figure 4.2d. The problem, then, is reduced to application through this single-transducer case. We compose the embedded transducer of Figure 4.2a with the result from Figure 4.2d and project, forming the wsa in Figure 4.2f.

Secondly, we can begin by only initially considering the first transducer in the chain, and compose the embedded wsa with it, as if we were doing a single-transducer application, obtaining the result in Figure 4.2e. If there are "dead" states and edges in this transducer, i.e., those that cannot be in any path, they may be removed at this time. Then, instead of projecting, we continue by composing that transducer with the next transducer in the chain, i.e., that of Figure 4.2c, and then finally take the range projection, again obtaining the wsa in Figure 4.2f. This approach is called the *bucket brigade*.

A third approach builds the application wsa incrementally, as dictated by some algorithm that requests information about it. Such an approach, which we call *on-the-fly*, was described in, e.g., [109, 101, 103]. These works generally described the effect of this approach rather than concrete algorithms, and for the moment we will do likewise, and

presume we can efficiently calculate the outgoing edges of a state of some wsa or wst on demand, without calculating all edges in the entire machine. The initial representation of the desired application wsa is depicted in Figure 4.2g and consists of only the start state. Now, consider Algorithm 16, an instantiation of Dijkstra's algorithm that seeks to find the cost of the highest-cost path in a wsa.[2] Notice that this algorithm only ever needs to know the identity of the start state, whether a state is the final state, and for a given state, the set of outgoing edges. We can use the wsa formed by offline composition or bucket brigade as input, but we can also use our initial wsa of Figure 4.2g as input. At line 15, the algorithm needs to know the outgoing edges from state ADF, the only state it knows about. Our on-the-fly algorithms allow us to discover such edges, and the application wsa is now that depicted in Figure 4.2h. We are able to determine the cost of a path to a final state once our application wsa looks like Figure 4.2i, at which point we can return the value ".1944". No other edges need be added to the application wsa. Notice an advantage over the other two methods is that not all the possible edges of the application wsa are built, and thus on-the-fly application can be faster than traditional methods. However, one disadvantage of the on-the-fly method is that work may be done to build states and edges that are in *no* valid path, as the incrementally-built wsa cannot be trimmed while the bucket brigade-built wsa can have dead states trimmed after each composition.

[2] Algorithm 16 requires a unique final state. We can convert any wsa into a wsa of this form by adding transitions with ϵ labels and weights of 1 from any final states to a new unique final state.

Algorithm 16 DIJKSTRA

inputs
  wsa $A = (Q, \Sigma, E, q_0, \{q_f\})$ over $\mathbb{W}$
outputs
  weight $w \in \mathbb{W}$, the weight of the highest-cost path from q_0 to q_f in A
complexity
  $O(|E| + |Q| \log |Q|)$, if a Fibonacci heap is used for queuing.

$\texttt{cost}(q_0) \leftarrow 1$
$Q \leftarrow q_0$
$\Xi \leftarrow \{q_0\}$ {seen nonterminals}
while Q is not empty **do**
  $q \leftarrow$ state from Q with highest cost
  Remove q from Q
  if $q = q_f$ **then**
    return $\texttt{cost}(q)$
  for all e of the form $q \xrightarrow{\sigma/w} p$ in E **do**
    if $p \notin \Xi$ **then**
      $\texttt{cost}(p) \leftarrow \texttt{cost}(q) \cdot w$
      $Q \leftarrow Q \cup \{p\}$
      $\Xi \leftarrow \Xi \cup \{p\}$
    else
      $\texttt{cost}(p) \leftarrow \max(\texttt{cost}(p), \texttt{cost}(q) \cdot w)$
return 0

Type	Pres. recog?	Source
wxT	No	See NT
xT	No	See NT
wT	No	See NT
T	No	See NT
wxLT	OQ	[91]
xLT	Yes	[90], Thm. 4, cl. 2
wLT	OQ	[91]
LT	Yes	[48] Cor. IV.6.6
wxNT	No	See NT
xNT	No	See NT
wNT	No	See NT
NT	No	[48] Thm. 6.11
wxLNT	Yes	[43]
xLNT	Yes	[90], Thm. 4, cl. 1
wLNT	Yes	[82], Cor. 14
LNT	Yes	see wLNT

(a) Preservation of forward recognizability

Type	Pres. recog?	Source
wxT	No	See wNT
xT	Yes	Thm. 4.4.5
wT	No	See wNT
T	Yes	[48] Cor. IV.3.17
wxLT	Yes	[43]
xLT	Yes	See wxLT
wLT	Yes	See wxLT
LT	Yes	See wxLT
wxNT	No	See wNT
xNT	Yes	See xT
wNT	No	[91]
NT	Yes	See xT
wxLNT	Yes	See wxLT
xLNT	Yes	See wxLT
wLNT	Yes	See wxLT
LNT	Yes	See wxLT

(b) Preservation of backward recognizability

Table 4.1: Preservation of forward and backward recognizability for various classes of top-down tree transducers. Here and elsewhere, the following abbreviations apply: w = weighted, x = extended left side, L = linear, N = nondeleting, OQ = open question.

4.4 Application of tree transducers

Now let us revisit these stories with trees and tree transducers. Imagine we have a tree and a wtt that can transform that tree with some weight. We'd like to know the k-best trees the wtt can produce as output, along with their weights. We already know of at least one method for acquiring k-best trees from a wrtg [59], so we then must ask if, analogously to the string case, wtts preserve recognizability and we can form an application wrtg.

We consider preservation of recognizability first. Known results for top-down tree transducer classes considered in this thesis are shown in Table 4.1. Unlike the string case, preservation of recognizability is not universal or symmetric. If a transducer preserves *forward recognizability*, then a regularly limited domain implies a regular range, and if

it preserves *backward recognizability*, then a regularly limited range implies a regular domain.[3] Succinctly put, wxLNT and its subclasses preserve forward recognizability, as does xLT and its subclass LT. The two cases marked as open questions and the other classes, which are superclasses of NT, do not or are presumed not to. All classes except wNT and its subclasses preserve backward recognizability. We do not consider cases where recognizability is not preserved in the remainder of this chapter. If a transducer M of a class that preserves forward recognizability is applied to a wrtg G, we can call the forward application wrtg $M(G)^{\triangleright}$ and if M preserves backward recognizability, we can call the backward application wrtg $M(G)^{\triangleleft}$.

Now that we have defined the application problem and determined the classes for which application is possible, let us consider how to build forward and backward application wrtgs. An initial approach is to mimic the results found for wsts, by using an embed-compose-project strategy. However, we must first consider whether, as in string world, there is a connection between recognizability and composition. We introduce a theorem to present that connection and outline the cases where this strategy is valid.

We recall the various definitions from Section 2.1. To them we add the following: The *application* of a weighted tree transformation $\tau : T_\Sigma \times T_\Delta$ to a tree series $L : T_\Sigma \to \mathbb{W}$ is a tree series $\tau\langle L\rangle : T_\Delta \to \mathbb{W}$ where for every $s \in T_\Delta$, $\tau\langle L\rangle(s) = \bigoplus_{t \in T_\Sigma} L(t) \cdot \tau(t, s)$.

Next, we need three lemmas. The first two demonstrate the semantic equivalence between application and composition.

[3] Formally speaking, it is not the transducers that preserve recognizability but their *transformations* and one must speak of a transformation preserving recognizability (analogous to a transducer preserving forward recognizability) or its inverse doing so (analogous to preserving backward recognizability).

Lemma 4.4.1 (forward application composition connection) Let L be a tree series $T_\Sigma \to \mathbb{W}$. Let τ be a weighted tree transformation $T_\Sigma \times T_\Delta \to \mathbb{W}$. Then $\tau\langle L\rangle = \text{range}(\iota_L ; \tau)$.

Proof For any $t \in T_\Delta$:

$$\begin{aligned}
\text{range}(\iota_L ; \tau)(t) &= \bigoplus_{s \in T_\Sigma} \iota_L ; \tau(s,t) && \text{Definition of range} \\
&= \bigoplus_{s \in T_\Sigma} \bigoplus_{u \in T_\Sigma} \iota_L(s,u) \cdot \tau(u,t) && \text{Definition of composition} \\
&= \bigoplus_{s \in T_\Sigma} \iota_L(s,s) \cdot \tau(s,t) && \text{Remove elements equal to } 0 \\
&= \bigoplus_{s \in T_\Sigma} L(s) \cdot \tau(s,t) && \text{Definition of identity} \\
&= \tau\langle L\rangle(t) && \text{Definition of application}
\end{aligned}$$

■

Lemma 4.4.2 (backward application composition connection) Let L be a tree series $T_\Delta \to \mathbb{W}$. Let τ be a weighted tree transformation $T_\Sigma \times T_\Delta \to \mathbb{W}$. Then $\tau^{-1}\langle L\rangle = \text{dom}(\tau ; \iota_L)$.

Proof For any $t \in T_\Sigma$:

$$\begin{aligned}
\text{dom}(\tau ; \iota_L)(t) &= \bigoplus_{s \in T_\Delta} \tau ; \iota_L(t,s) && \text{Definition of domain} \\
&= \bigoplus_{s \in T_\Delta} \bigoplus_{u \in T_\Delta} \tau(t,u) \cdot \iota_L(u,s) && \text{Definition of composition} \\
&= \bigoplus_{s \in T_\Delta} \tau(t,s) \cdot \iota_L(s,s) && \text{Remove elements equal to } 0 \\
&= \bigoplus_{s \in T_\Delta} \tau(t,s) \cdot L(s) && \text{Definition of identity} \\
&= \tau^{-1}\langle L\rangle(t) && \text{Definition of application}
\end{aligned}$$

■

For the third lemma we need to introduce the *universal* tree series, which recognizes all possible trees in a language with weight 1. We will then show that for any transformation,

application of the universal tree series is equivalent to the range of the transformation, and for the transformation's inverse, application is equivalent to the domain.

Definition 4.4.3 The universal tree series of T_Σ over semiring $\mathbb{W}$ is a tree series $U_{T_\Sigma} : T_\Sigma \to \mathbb{W}$ where, for every $t \in T_\Sigma$, $U_{T_\Sigma}(t) = 1$. Note that U_{T_Σ} is recognizable. This can be easily verified by defining the wrtg $G = (\{n\}, \Sigma, P, n)$ where, for every $\sigma \in \Sigma^{(k)}$, $n \xrightarrow{1} \sigma(n, ..., n) \in P$. Clearly, $L_G = U_{T_\Sigma}$.

Lemma 4.4.4 (application of universal equal to range) Let τ be a weighted tree transformation $T_\Sigma \times T_\Delta \to \mathbb{W}$. Then $\tau\langle U_{T_\Sigma}\rangle = \text{range}(\tau)$ and $\tau^{-1}\langle U_{T_\Delta}\rangle = \text{dom}(\tau)$.

Proof For any $t \in T_\Delta$:

$$\tau\langle U_{T_\Sigma}\rangle = \bigoplus_{s \in T_\Sigma} U_{T_\Sigma}(s) \cdot \tau(s,t) \qquad \text{Definition of application}$$

$$= \bigoplus_{s \in T_\Sigma} \tau(s,t) \qquad \text{Definition of universal tree series}$$

$$= \text{range}(\tau)(t) \qquad \text{Definition of range}$$

The second statement is easily obtained by substituting τ^{-1} and U_{T_Δ} for τ and U_{T_Σ}, respectively. ■.

We may now prove the theorem.

Theorem 4.4.5 (due to Maletti [91]) For a weighted top-down tree transducer class A over semiring $\mathbb{W}$, if class A is closed under left-composition[4] (respectively, right-composition) with LNT($\mathbb{W}$) then "A has recognizable range (respectively, domain) over semiring $\mathbb{W}$" $\Leftrightarrow$ "A (respectively, A^{-1}) preserves recognizability".

[4]Tree transducer class A is said to be closed under left- (respectively, right-) composition with class B if $B \circ A \subseteq A$ (respectively, $A \circ B \subseteq A$).

Proof We begin with the forward case. Let A be a class of wtt such that A is closed under left-composition with LNT. Let M be a wtt of class A over semiring $\mathbb{W}$ and let L be a recognizable tree series over $\mathbb{W}$. First we assume that M has recognizable range. Due to the properties of class A, $\imath_L \circ M$ is in class A. Then, from Lemma 4.4.1, $\tau_M\langle L\rangle = \text{range}(\tau_{\imath_L \circ M})$. Based on our assumption, the latter has recognizable range, thus the former does too, and thus τ_M preserves recognizability. Next we assume that τ_M preserves recognizability. Since U_{T_Σ} is recognizable, $\tau_M\langle U_{T_\Sigma}\rangle$ is recognizable. From Lemma 4.4.4, $\tau_M\langle U_{T_\Sigma}\rangle = \text{range}(\tau)$, thus $\text{range}(\tau)$ is recognizable.

Now for the backward case, let A be a class of wtt such that A is closed under right-composition with LNT. Again, let M be a wtt of class A over semiring $\mathbb{W}$ and let L be a recognizable tree series over $\mathbb{W}$. First, we assume that M has recognizable domain. Then, $M \circ \imath_L$ has recognizable domain and from Lemma 4.4.2, $\tau_M^{-1}\langle L\rangle = \text{dom}(\tau_{M \circ \imath_L})$ does too. Thus τ_M^{-1} preserves recognizability. Now assume τ_M^{-1} preserves recognizability. Since U_{T_Δ} is recognizable, $\tau_M^{-1}\langle U_{T_\Delta}\rangle$ is recognizable. From Lemma 4.4.4, $\tau_M^{-1}\langle U_{T_\Delta}\rangle = \text{dom}(\tau)$, thus $\text{dom}(\tau)$ is recognizable. ∎

This theorem implies that, for those classes that preserve recognizability and are closed under the appropriate composition,[5] a composition-based approach to application can be taken. Table 4.2 shows relevant composition results for potentially viable classes.

As can be seen from Table 4.2, the embed-compose-project approach will work for backward application cases but will not work for most forward application cases. The reader should note, however, that the composition results in Table 4.2 are for composition

[5] and have the appropriate recognizable projection, but this is always true by definition of classes preserving recognizability; see the proof.

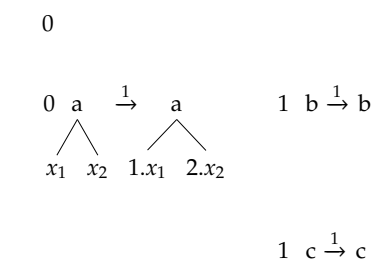
Type	Left-closed with (w)LNT?	Source
xLT	No	[92], Lem. 4.3
LT	No	[39], p. 207
wxLNT	No	See xLNT
xLNT	No	[92], Thm. 5.2
wLNT	Yes	[89], Thm. 26
LNT	Yes	see wLNT

(a) For forward application

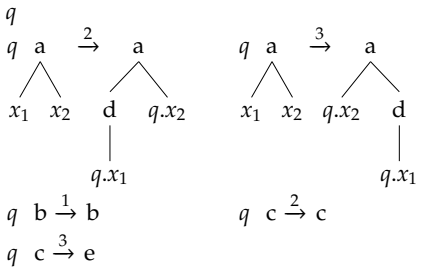
Type	Right-closed with (w)LNT?	Source
xT	Yes	See T
T	Yes	[6], Cor. 2
wxLT	Yes	[89], Thm. 26
xLT	Yes	See xT
wLT	Yes	See wxLT
LT	Yes	See T
xNT	Yes	See xT
NT	Yes	See T
wxLNT	Yes	See wxLT
xLNT	Yes	See xT
wLNT	Yes	See wxLT
LNT	Yes	See T

(b) For backward application

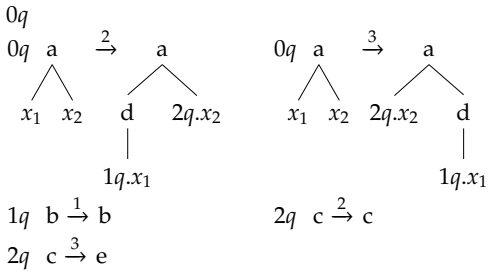
Table 4.2: For those classes that preserve recognizability in the forward and backward directions, are they appropriately closed under composition with (w)LNT? If the answer is "yes", then an embedding, composition, projection strategy can be used to do application.



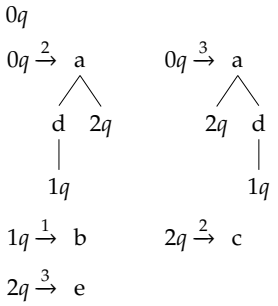
(a) Input tree a(b c) embedded in an identity wLNT.



(b) wLNT for forward application.

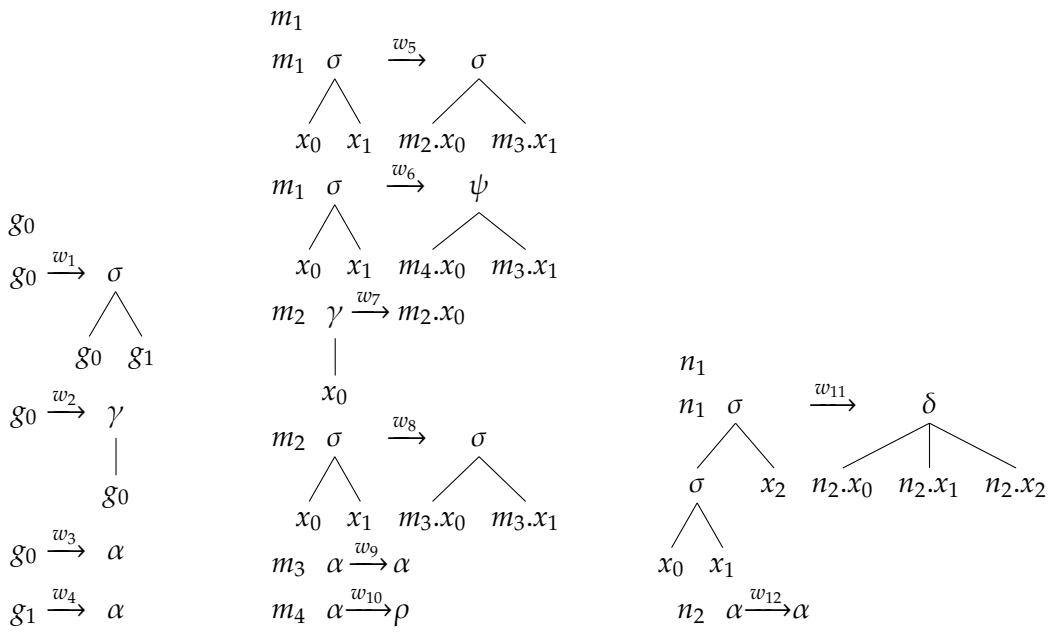


(c) Result of composition.



(d) Result of projection.

Figure 4.3: Composition-based approach to application of a wLNT to a tree.



(a) Input wrtg G.

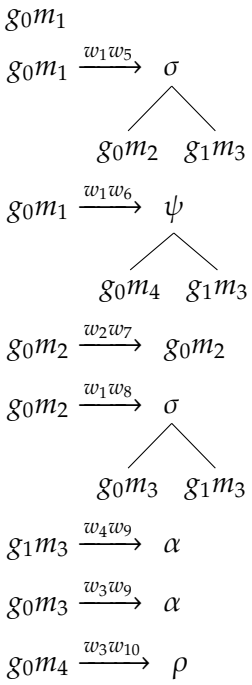
(b) First transducer (wLNT) M_1 in the cascade.

(c) Second transducer (wxLNT) M_2 in the cascade.

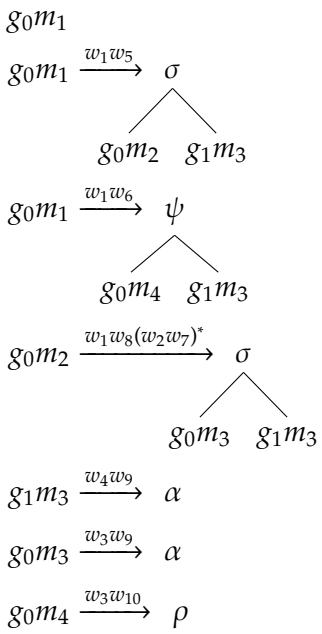
Figure 4.4: Inputs for forward application through a cascade of tree transducers.

with transducers of the class (w)LNT, while an embedded RTG is in a narrower class than this—it is a relabeling, deterministic (w)LNT. But then, in order to accomplish application via an embed-compose-project approach for the classes not appropriately closed with (w)LNT but closed with this narrow class, we must have a composition algorithm for this very specific class of transducers. Instead, it seems better to focus our energies on designing application algorithms.

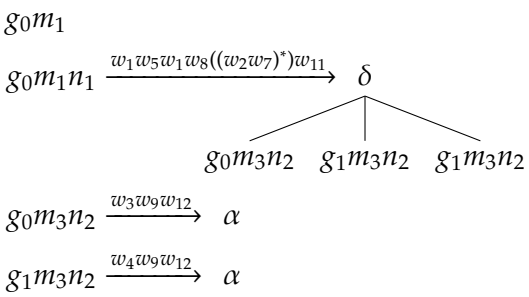
Before we discuss application algorithms, it is important to note the re-emergence of COVER algorithms in this chapter, similar to Algorithm 14 from Chapter 2. It should not be much of a surprise that the same kind of algorithm comes up in a discussion of application, which is very similar in principle to composition. We will in fact describe two separate COVER variants here, Algorithms 18 and 21. All three algorithms match



(a) Forward application of M_1 to G: G_1.



(b) After chain production removal of G_1.



(c) Final result through the cascade.

Figure 4.5: Results of forward application through a cascade of tree transducers.

trees to patterns of rules in a top-down manner, as described in Section 2.3.3. However, Algorithm 14 is concerned with joining together the right hand side of a rule from one tree transducer with a pattern of rules from a second, to form rules in a composition transducer. Algorithms 18 and 21, on the other hand, match a tree to a wrtg, and are concerned with the weight of matching derivations and the states reached in the wrtg, but do not need to construct new trees. These algorithms also differ from Algorithm 14 in that they invoke recursive calls to discover available productions in the input wrtgs, as will be discussed in more detail in Section 4.5. Both of the COVER algorithms in this chapter return mappings from tree positions to nonterminals; this allows the calling algorithms to convert trees that are parts of transducer rules into trees that form the right sides of new wrtg productions. Algorithm 18 primarily differs from Algorithm 21 in that the former matches wxtt rule left sides to trees and the latter matches right sides.

Algorithm 17 FORWARD-APPLICATION

inputs
  wrtg $G = (N, \Sigma, P, n_0)$ over $\mathbb{W}$ in normal form with no chain productions
  linear wxtt $M = (Q, \Sigma, \Delta, R, q_0)$ over $\mathbb{W}$
outputs
  wrtg $G' = (N', \Delta, P', n'_0)$ such that if M is nondeleting or $\mathbb{W}$ is Boolean, $L_{G'} = \tau_M(L_G)$
complexity
  $O(|R||P|^{\tilde{l}})$, where $\tilde{l}$ is the size of the largest left side tree in any rule in R

$N' \leftarrow (N \times Q)$
$n'_0 \leftarrow (n_0, q_0)$
$P' \leftarrow \emptyset$
for all $(n, q) \in N'$ **do**
  for all r of the form $q.t \xrightarrow{w_1} s$ in R **do**
    for all $(\phi, w_2) \in$ FORWARD-COVER(t, G, n) **do**
      Form substitution mapping $\varphi : Q \times X \to T_\Delta(N \times Q)$ such that $\varphi(q', x) = (n', q')$ if $\phi(v) = n'$ and $t(v) = x$ for all $q' \in Q$, $n' \in N$, $x \in X$, and $v \in pos(t)$.
      $P' \leftarrow P' \cup \{(n, q) \xrightarrow{w_1 \cdot w_2} \varphi(s)\}$
return G'

Algorithm 18 FORWARD-COVER

inputs
  $t \in T_\Sigma(A)$, where $A \cap \Sigma = \emptyset$
  wrtg $G = (N, \Sigma, P, n_0)$ in normal form, with no chain productions
  $n \in N$
outputs
  set Π of pairs $\{(\phi, w) : \phi$ a mapping $pos(t) \to N$ and $w \in \mathbb{W}\}$, each pair indicating a successful run on t by productions in G, starting from n, and the weight of the run.
complexity
  $O(|P|^{size(t)})$

1: $\Pi_{last} \leftarrow \{((\varepsilon, n), 1)\}$
2: **for all** $v \in pos(t)$ such that $t(v) \notin A$ in pre-order **do**
3: $\Pi_v \leftarrow \emptyset$
4: **for all** $(\phi, w) \in \Pi_{last}$ **do**
5: **if** $G \preceq \overline{M}(\overline{G})^\triangleright$ for some wrtg $\overline{G}$ and w(x)tt $\overline{M}$ **then**
6: $G \leftarrow$ FORWARD-PRODUCE$(\overline{G}, \overline{M}, G, \phi(v))$
7: **for all** $\phi(v) \xrightarrow{w'} t(v)(n_1, ..., n_k) \in P$ **do**
8: $\Pi_v \leftarrow \Pi_v \cup \{(\phi \cup \{(vi, n_i), 1 \le i \le k\}, w \cdot w')\}$
9: $\Pi_{last} \leftarrow \Pi_v$
10: **return** Π_{last}

Algorithm 17 produces the application wrtg for those classes of tree transducer preserving recognizability listed in Table 4.1a. The implementation optimization previously described in Chapter 2 for Algorithms 6, 7, 8, and 13 applies to this algorithm as well. Note that it covers several cases not allowed by the embed-compose-project strategy, specifically wxLNT. It does require that the input wrtg be in normal form with no chain productions; algorithms for ensuring these properties are described in previous sections. The algorithm pairs nonterminals n from a wrtg $G = (N, \Sigma, P, n_0)$ with states q from a transducer $M = (Q, \Sigma, \Delta, R, q_0)$, and finds a derivation starting with n that matches the left hand side of a rule starting with q; this is done by a call to FORWARD-COVER at line 13. The substitution mapping that converts the right hand side of the rule from a

Type	Closed?	Source
wxT	No	See T
xT	No	See T
wT	No	See T
T	No	[48] Cor. IV.3.14
wxLT	No	See LT
xLT	No	See LT
wLT	No	See LT
LT	No	[48] Thm. IV.3.6
wxNT	No	See NT
xNT	No	See NT
wNT	No	See NT
NT	No	[6] Thm. 1
wxLNT	No	See xLNT
xLNT	No	[92] Thm. 5.4
wLNT	Yes	[45] Lem. 5.11
LNT	Yes	[48] Thm. IV.3.6

Table 4.3: Closure under composition for various classes of top-down tree transducer.

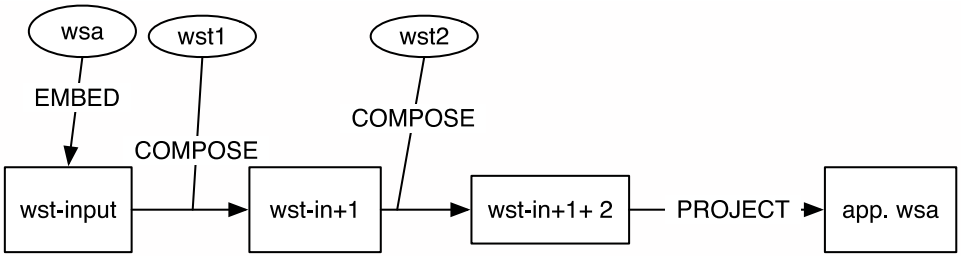
tree in $T_\Delta(Q \times X)$ to a tree in $T_\Delta(N \times Q)$ appropriate for the output wrtg, is formed at line 14, enabling the new wrtg production to be built at line 15.

An example that uses this algorithm is provided below, in the discussion of application through cascades; the example is relevant to the single transducer case as well.

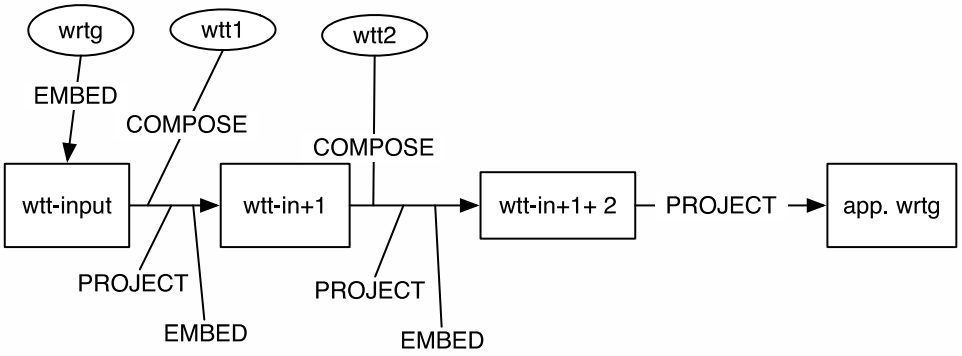
4.5 Application of tree transducer cascades

What about the case of an input wrtg and a sequence of tree transducers? We will revisit the three ways of accomplishing application discussed above for the string case.

In order for offline composition to be a viable strategy, the transducers in the cascade must be closed under composition. As Table 4.3 shows, in general a cascade of tree transducers of a given class cannot be composed offline to form a single transducer; the lone exceptions being for a cascade of wLNT and LNT transducers.



(a) Schematic of application on a wst cascade. The given wsa (or string) is embedded in an identity wst, then composed with the first transducer in the cascade. In turn, each transducer in the cascade is composed into the result. Then, a projection is taken to obtain the application wsa.



(b) Schematic of application on a wtt cascade, illustrating the additional complexity compared with the string case. The given wrtg (or tree) is embedded in an identity wtt, then composed with the first transducer in the cascade. Then a projection is taken to form a wrtg, and this wrtg must immediately be embedded in an identity wtt to ensure composability with the next transducer.

Figure 4.6: Schematics of application, illustrating the extra work needed to use embed-compose-project in wtt application vs. wst application.

When considering the bucket brigade, we have two options for specific methods; either the embed-compose-project approach or the custom algorithm approach can be used. The embed-compose-project process is somewhat more burdensome than in the string case. Recall that, for strings, application is obtained by an embedding, a series of compositions, and a projection (see Figure 4.6a). As discussed above, in general the "series of compositions" is impossible for trees. However, one *can* obtain application by a series of embed-compose-project operations, as depicted in Figure 4.6b.

The custom algorithm case, which applies in instances the embed-compose-project does not cover (see Table 4.2), is easily usable in a bucket brigade scenario. One must, however, ensure the output of the algorithm is in normal form (using Algorithm 1) and chain production-free (using Algorithm 3).

Let us now work through an example (depicted in Figures 4.4 and 4.5) of bucket brigade through a cascade using custom algorithms in order to better understand the mechanism of these algorithms. In particular we will use a cascade similar to one described in [92] that demonstrated lack of closure under composition for certain classes. As this example makes clear, even though the transducers in the cascade are not closed under composition, the property of preservation of recognizability is sufficient for application. In this example the weights are kept as variables so that the semiring operations being performed remain obvious.

The input wrtg, which we will call G, is in Figure 4.4a and the transducers M_1 and M_2 are in Figures 4.4b and 4.4c, respectively. In the example that follows we will first form $M_1\langle G\rangle$ and then form $M_2\langle M_1\langle G\rangle\rangle$.

Note that G is in normal form and is chain-production free, and M_1 is linear and nondeleting, so the inputs to Algorithm 17 are valid. As indicated on line 8, the nonterminals of the result will be pairs of (nonterminal, state) from the inputs and the initial nonterminal is $g_0 m_1$, corresponding to g_0 from G and m_1 from M_1. In the main while loop we consider productions from a particular new nonterminal, and we begin with the new initial nonterminal. At line 12 we choose a rule from M_1 beginning with m_1, namely $m_1.\sigma(x_0, x_1) \xrightarrow{w_5} \sigma(m_2.x_0, m_3.x_1)$. We then must invoke Algorithm 18, FORWARD-COVER, to form a covering of the left side of this rule. At line 15 of FORWARD-COVER, the production $g_0 \xrightarrow{w_1} \sigma(g_0, g_1)$ is chosen to cover $\sigma(x_0, x_1)$, and at line 16 g_0 is mapped to x_0 and g_1 to x_1 and the weight of this mapping is set to w_1, the weight of the production used.[6] Back in the main algorithm, this mapping is used to form the substitution mapping[7] φ that ultimately converts the right side of the transducer rule, $\sigma(m_2.x_0, m_3.x_1)$, into $\sigma(g_0 m_2, g_1 m_3)$. The weight of the rule is multiplied by the weight of the mapping, and the new production $g_0 m_1 \xrightarrow{w_1 w_5} \sigma(g_0 m_2, g_1 m_3)$ is formed. In a similar manner, the entire application wrtg depicted in Figure 4.5a is formed.

The next task in the cascade is to apply M_2 to the just-formed application wrtg. However, the wrtg we just formed has chain productions in it. Thus, before continuing a chain production removal algorithm (described elsewhere) is used to convert the application wrtg to that depicted in Figure 4.5b. We then continue with application of M_2 to the wrtg of Figure 4.5b. The application process is the same as that just described. Note, however, that the computation in FORWARD-COVER is somewhat more

[6] The meaning behind lines 13 and 14 of FORWARD-COVER will be shortly explained, but since the test does not apply they can be safely skipped for now.

[7] Recall this definition from Section 2.1.4.

complicated for $n_1.\sigma(\sigma(x_0, x_1), x_2) \xrightarrow{w_{11}} \delta(n_2.x_0, n_2.x_1, n_2.x_2)$ due to its extended left side. In this case, the tree $\sigma(\sigma(x_0, x_1), x_2)$ is covered by $g_0 m_1 \xrightarrow{w_1 w_5} \sigma(g_0 m_2, g_1 m_3)$ followed by $g_0 m_2 \xrightarrow{w_1 w_8 (w_2 w_7)^*} \sigma(g_0 m_3, g_1 m_3)$. If we had more transducers in the cascade, we would continue applying them to the result of the previous application step, but in our example we are done after two applications. The final result is in Figure 4.5c.

Algorithm 19 FORWARD-PRODUCE

inputs
  wrtg $G = (N, \Sigma, P, n_0)$ over $\mathbb{W}$ in normal form with no chain productions
  linear wxtt $M = (Q, \Sigma, \Delta, R, q_0)$ over $\mathbb{W}$
  wrtg $G'_{in} = (N'_{in}, \Delta, P'_{in}, n'_0)$ over $\mathbb{W}$
  $n_{in} \in N'_{in}$
outputs
  wrtg $G'_{out} = (N'_{out}, \Delta, P'_{out}, n'_0)$ over $\mathbb{W}$ where, if M is nondeleting or $\mathbb{W}$ is Boolean, $G'_{in} \preceq_{M(G)^\triangleright} G'_{out}$, and for all $w \in \mathbb{W}$, $t \in T_\Delta(N')$, $n_{in} \xrightarrow{w} t \in P'_{out} \Leftrightarrow n_{in} \xrightarrow{w} t \in M(G)^\triangleright$
complexity
  $O(|R||P|^{\tilde{l}})$, where $\tilde{l}$ is the size of the largest left side tree in any rule in R

if P'_{in} contains productions of the form $n_{in} \xrightarrow{w} u$ **then**
  return G'_{in}
$N'_{out} \leftarrow N'_{in}$
$P'_{out} \leftarrow P'_{in}$
Let n_{in} be of the form (n, q), where $n \in N$ and $q \in Q$.
for all r of the form $q.t \xrightarrow{w_1} s$ in R **do**
  for all $(\phi, w_2) \in$ FORWARD-COVER(t, G, n) **do**
    Form substitution mapping $\varphi : Q \times X \to T_\Delta(N \times Q)$ such that, for all $v \in ydset(t)$ and $q' \in Q$, if there exist $n' \in n$ and $x \in X$ such that $\phi(v) = n'$ and $t(v) = x$, $\varphi(q', x) = (n', q')$.
    for all $p'' \in$ NORMALIZE$((n, q) \xrightarrow{w_1 \cdot w_2} \varphi(s), N'_{out})$ **do**
      Let p'' be of the form $n'' \xrightarrow{w''} \delta(n''_1, \ldots, n''_k)$ for $\delta \in \Delta^{(k)}$.
      $N_{out} \leftarrow N_{out} \cup \{n'', n''_1, ..., n''_k\}$
      $P'_{out} \leftarrow P'_{out} \cup \{p''\}$
return G'_{out}

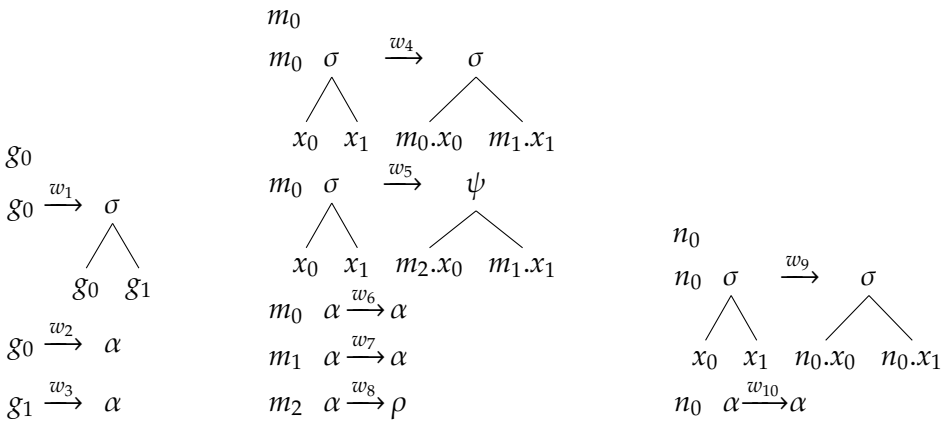
We next consider on-the-fly algorithms for application. As in the string case, an on-the-fly approach is driven by a calling algorithm that periodically needs to know

the productions in a wrtg with a common left side nonterminal. Both the embed-compose-project approach and the previously defined custom algorithms produce entire application wrtgs. In order to admit an on-the-fly approach we describe algorithms that only generate those productions in a wrtg that have a given left nonterminal.

The following set of algorithms have a common flavor in that they take as input a wrtg and a desired nonterminal and return another wrtg, different from the input wrtg in that it has more productions, specifically those beginning with that specified nonterminal. The wrtgs provided as input to and returned as output from these *produce* algorithms can be thought of as *stand-ins* for some wrtg built in a non-on-the-fly manner. Algorithms using these stand-ins should call an appropriate produce algorithm to ensure the stand-in they are using has the productions beginning with the desired nonterminal.

Algorithm 19, FORWARD-PRODUCE, obtains the effect of forward application in an on-the-fly manner. It takes as input a wrtg and appropriate transducer, as well as a stand-in and desired nonterminal. As an example, consider the invocation FORWARD-PRODUCE(G_1, M_1, G_{init}, g_0m_0), where G_1 is in Figure 4.7a, M_1 is in 4.7b, and G_{init} has an empty production set and a nonterminal set consisting of only the start nonterminal, g_0m_0. The stand-in wrtg that is output contains three productions: $g_0m_0 \xrightarrow{w_1w_4} \sigma(g_0m_0, g_1m_1)$, $g_0m_0 \xrightarrow{w_1w_5} \psi(g_0m_2, g_1m_1)$, and $g_0m_0 \xrightarrow{w_2w_6} \alpha$.

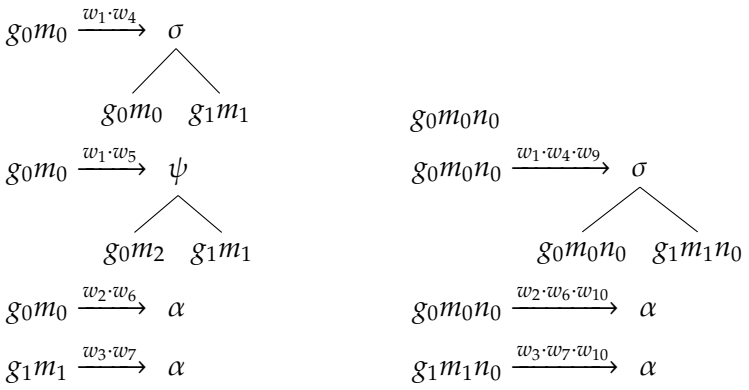
To demonstrate the use of on-the-fly application in a cascade, we next show the effect of FORWARD-PRODUCE when used with the cascade of G_1 M_1, and M_2, where M_2 is in Figure 4.7c. Our driving algorithm in this case is Algorithm 20, MAKE-EXPLICIT, which simply generates the full application wrtg using calls to FORWARD-PRODUCE. The input to MAKE-EXPLICIT is the first stand-in for $M_2(M_1(G_1)^{\triangleright})^{\triangleright}$, the result of forward



(a) Input wrtg G_1.

(b) First transducer M_1 in the cascade.

(c) Second transducer M_2 in the cascade.



(d) Productions of $M_1(G_1)^\triangleright$ built as a consequence of building the complete $M_2(M_1(G_1)^\triangleright)^\triangleright$.

(e) Complete $M_2(M_1(G_1)^\triangleright)^\triangleright$.

Figure 4.7: Forward application through a cascade of tree transducers using an on-the-fly method.

Algorithm 20 MAKE-EXPLICIT

inputs
  wrtg $G = (N, \Sigma, P, n_0)$ in normal form
outputs
  wrtg $G' = (N', \Sigma, P', n_0)$, in normal form, such that $L_G = L_{G'}$ and if $G \preceq \overline{M}(\overline{G})^{\triangleright}$ for some wrtg $\overline{G}$ and w(x)tt $\overline{M}$, $G' = \overline{M}(\overline{G})^{\triangleright}$.
complexity
  $O(|P'|)$

$G' \leftarrow G$
$\Xi \leftarrow \{n_0\}$ {seen nonterminals}
$\Psi \leftarrow \{n_0\}$ {pending nonterminals}
while $\Psi \neq \emptyset$ **do**
  $n \leftarrow$any element of Ψ
  $\Psi \leftarrow \Psi \setminus \{n\}$
  if $G' \preceq \overline{M}(\overline{G})^{\triangleright}$ for some wrtg $\overline{G}$ and w(x)tt $\overline{M}$ **then**
    $G' \leftarrow$ FORWARD-PRODUCE$(\overline{G}, \overline{M}, G', n)$
  for all $n \xrightarrow{w} \sigma(n_1, ..., n_k) \in P'$ **do**
    for $i = 1$ to k **do**
      if $n_i \notin \Xi$ **then**
        $\Xi \leftarrow \Xi \cup \{n_i\}$
        $\Psi \leftarrow \Psi \cup \{n_i\}$
return G'

application of M_2 to $M_1(G_1)^\triangleright$, which is itself the result of forward application of M_1 to G_1. This initial input is a wrtg with an empty production set and a single nonterminal, $g_0m_0n_0$, obtained by combining n_0, the initial state from M_2, with g_0m_0, the initial nonterminal from the first stand-in for $M_1(G_1)^\triangleright$. MAKE-EXPLICIT calls FORWARD-PRODUCE($M_1(G_1)^\triangleright$, M_2, $M_2(M_1(G_1)^\triangleright)^\triangleright$, $g_0m_0n_0$). FORWARD-PRODUCE then seeks to cover $n_0.\sigma(x_0, x_1) \xrightarrow{w_9} \sigma(n_0.x_0, n_0.x_1)$ with productions from $M_1(G_1)^\triangleright$, thus it needs to improve the stand-in for this wrtg. In FORWARD-COVER, there is a call to FORWARD-PRODUCE that accomplishes this. The productions of $M_1(G_1)^\triangleright$ that must be built to form the complete $M_2(M_1(G_1)^\triangleright)^\triangleright$ are shown in Figure 4.7d. The complete $M_2(M_1(G_1)^\triangleright)^\triangleright$ is shown in Figure 4.7e. Note that because we used this on-the-fly approach, we were able to avoid building all the productions in $M_1(G_1)^\triangleright$; in particular we did not build $g_0m_2 \xrightarrow{w_2w_8} \rho$, while a bucket brigade approach would have built this production.

Algorithm 22 is an analogous on-the-fly PRODUCE algorithm for backward application. It is only appropriate for application on linear transducers. We do not provide PRODUCE for non-linear tree transducers because weighted non-linear tree transducers do not preserve backward recognizability, and on-the-fly methods are not terribly useful for obtaining (unweighted) application rtgs. Since there is no early stopping condition the entire application may as well be carried out.

We have now defined several on-the-fly and bucket brigade algorithms, and also discussed the possibility of embed-compose-project and offline composition strategies to application of cascades of tree transducers. Tables 4.4 and 4.5 summarize the available methods of forward and backward application of cascades for recognizability-preserving tree transducer classes.

Algorithm 21 BACKWARD-COVER

inputs
  $t \in T_\Sigma(A)$, where $A \cap \Sigma = \emptyset$
  wrtg $G = (N, \Sigma, P, n_0)$ in normal form, with no chain productions
  $n \in N$
outputs
  set Π of pairs $\{(\phi, w) : \phi$ a mapping $pos(t) \to N$ and $w \in \mathbb{W}\}$, each pair indicating a successful run on t by productions in G, starting from n, and the weight of the run.
complexity
  $O(|P|^{size(t)})$

$\Pi_{last} \leftarrow \{((\varepsilon, n), 1)\}$
for all $v \in pos(t)$ such that $t(v) \notin A$ in pre-order **do**
  $\Pi_v \leftarrow \emptyset$
  for all $(\phi, w) \in \Pi_{last}$ **do**
    if $G \preceq \overline{M}(\overline{G})^{\triangleleft}$ for some wtt $\overline{M}$ and wrtg $\overline{G}$ **then**
      $G \leftarrow$ BACKWARD-PRODUCE$(\overline{M}, \overline{G}, G, \phi(v))$
    for all $\phi(v) \xrightarrow{w'} t(v)(n_1, ..., n_k) \in P$ **do**
      $\Pi_v \leftarrow \Pi_v \cup \{(\phi \cup \{(vi, n_i), 1 \le i \le k\}, w \cdot w')\}$
  $\Pi_{last} \leftarrow \Pi_v$
return Π_{last}

4.6 Decoding experiments

The main purpose of this chapter has been to present novel algorithms for performing application. However, it is beneficial to demonstrate these algorithms on realistic data. We thus demonstrate bucket brigade and on-the-fly backward application on a typical NLP task cast as a cascade of wLNT. We adapted the Japanese-to-English translation model of Yamada and Knight [136] by transforming it from an English tree-to-Japanese string model to an English tree-to-Japanese *tree* model. The Japanese trees are unlabeled, meaning they have syntactic structure but no node labels. We then cast this modified model as a cascade of LNT tree transducers. We now describe the individual transducers in more detail.

Algorithm 22 BACKWARD-PRODUCE

inputs
  linear wtt $M = (Q, \Sigma, \Delta, R, q_0)$,
  wrtg $G = (N, \Sigma, P, n_0)$ in normal form with no chain productions
  wrtg $G'_{in} = (N'_{in}, \Delta, P'_{in}, n'_0)$
  $n' \in (Q \times N)$
outputs
  wrtg $G'_{out} = (N'_{out}, \Delta, P'_{out}, n'_0)$ where $G'_{in} \preceq_{M(G)^\triangleleft} G'_{out}$, and $n' \xrightarrow{w} t \in P'_{out} \Leftrightarrow n' \xrightarrow{w} t \in M(G)^\triangleleft$
complexity
  $O(|R||P|^{\bar{r}})$

if $n' \in N'_{in}$ **then**
  return G'_{in}
$N'_{out} \leftarrow N'_{in} \cup \{n'\}$
$P'_{out} \leftarrow P'_{in}$
if $n' = \perp$ **then**
  for all $\sigma \in \Sigma$ with rank k **do**
    $P'_{out} \leftarrow P'_{out} \cup \{\perp \xrightarrow{1} \sigma(\perp, ..., \perp)\}$
  return G'_{out}
Let n' be of the form (q, n), where $q \in Q$ and $n \in N$.
for all r of the form $q.\sigma \xrightarrow{w_1} t$ in R where $\sigma \in \Sigma^{(k)}$ **do**
  for all $(\phi, w_2) \in$ BACKWARD-COVER(t, G, n) **do**
    $d_1 \leftarrow d_2 \leftarrow ... \leftarrow d_k \leftarrow \perp$
    for all $v \in leaves(t)$ such that $t(v)$ is of the form $(q', x) \in Q \times X_k$ **do**
      if $\phi(v) \neq \emptyset$ **then**
        $d_i \leftarrow (q', \phi(v))$
    $P'_{out} \leftarrow P'_{out} \cup \{(q, n) \xrightarrow{w_1 \cdot w_2} \sigma(d_1, ..., d_k)\}$
return G'_{out}

Rotation: The rotation transducer captures the reordering of subtrees such that the leaves of the tree are transformed from English to Japanese word order. Individual rules denote the likelihood of a particular sequence of sibling subtrees reordering in a particular way. The structure of the English trees ensures that the maximum number of siblings is four. Preterminals and English words are transformed as an identity, with no extra weight incurred. There are 6,453 rules in the rotation transducer in our experimental model. Some example rules are in Figure 4.8a.

Method	wfst	[x]LT	[w]xLNT	[w]LNT
oc	√	×	×	√
ecp	√	×	×	√
bb	√	√	√	√
otf	√	√	√	√

Table 4.4: Transducer types and available methods of forward application of a cascade. oc = offline composition, ecp = embed-compose-project, bb = custom bucket brigade algorithm, otf = on the fly.

Method	wfst	[x]T	[w][x]LT	[x]NT	[w]xLNT	[w]LNT
oc	√	×	×	×	×	√
ecp	√	√	√	√	√	√
otf	√	×	√	×	√	√

Table 4.5: Transducer types and available methods of backward application of a cascade. oc = offline composition, ecp = embed-compose-project, otf = on the fly.

Insertion: The insertion transducer captures the likelihood of inserting Japanese function words into the reordered English sentence. Individual rules denote the likelihood of inserting a function word to the left or right of a tree's immediate subtrees but do not specify what that word is (this is left for the translation transducer). Preterminals and English words are transformed as an identity with no extra weight incurred, with some English words receiving an annotation indicating they are not to be translated into a null symbol in Japanese, based on structural constraints of the model. There are 8,122 rules in the insertion transducer in our experimental model. Some example rules are in Figure 4.8b.

Translation: The translation transducer relabels all internal nodes of an English tree with the symbol "X" and captures the likelihood of translating each English and inserted word into a Japanese word or a null symbol, indicating there is no direct translation of the English word.[8] Individual rules that relabel with the symbol "X" have no extra

[8]Inserted words and specially annotated English words cannot translate into the null symbol.

Queue	Given	Actions
—	—	$O(n_o, 1, 1)$
$O(n, c)$	$out[n] = 0$	$out[n] \leftarrow c$ $G \leftarrow$ BACKWARD-PRODUCE$(\overline{G}, \overline{M}, G, n)$ $\forall p : n \xrightarrow{w} \sigma^{(k)}(n_1, ..., n_k) \in G, k > 0$ $\forall i \in 1 \ldots k$ $O(n_i, w \cdot c)$ $\forall p : n \xrightarrow{w} \alpha^{(0)} \in G$ $I(n, p, w, w \cdot c)$
$I(n, p, w, c)$	$in[n] = 0, deriv[n] = \emptyset$	$in[n] \leftarrow w, deriv[n] \leftarrow p$ return $deriv$ if $n = n_0$
—	$in[n_1] = w_1, \ldots, in[n_k] = w_k$ $out[n] = w_0$ $p : n \xrightarrow{w} \sigma^{(k)}(n_1, ..., n_k) \in G$	$I(n_0, p, w \cdot \prod_{i=1}^{k} w_i, w \cdot \prod_{i=0}^{k} w_i)$

Table 4.6: Deduction schema for the one-best algorithm of Pauls and Klein [108], generalized for a normal-form wrtg, and with on-the-fly discovery of productions. We are presumed to have a wrtg $G = (N, \Sigma, P, n_0)$ that is a stand-in for some $\overline{M}(\overline{G})^{\blacktriangleleft}$, a priority queue that can hold items of type I and O, prioritized by their cost, c, two N-indexed tables of (initially 0-valued) weights, *in* and *out*, and one N-indexed table of (initially null-valued) productions, *deriv*. In each row of this schema, the specified actions are taken (inserting items into the queue, inserting values into the tables, discovering new productions, or returning *deriv*) if the specified item is at the head of the queue and the specified conditions of *in*, *out*, and *deriv* exist. The one-best hyperpath in G can be found when *deriv* is returned by joining together productions in the obvious way, beginning with $deriv[n_0]$.

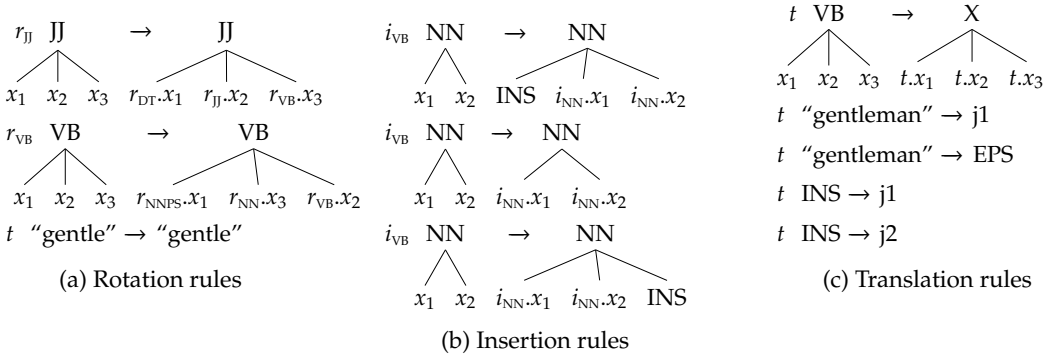


Figure 4.8: Example rules from transducers used in decoding experiment. j1 and j2 are Japanese words.

weight incurred. There are 37,311 rules in the translation transducer in our experimental model. Some example rules are in Figure 4.8c.

We added an English syntax language model to the cascade of transducers just described to better simulate an actual machine translation decoding task. The language model was cast as an identity wtt and thus fit naturally into the experimental framework. In our experiments we tried several different language models to demonstrate varying performance of the application algorithms. The most realistic language model was built from a PCFG, where each rule captured the probability of a particular sequence of child labels given a parent label. This model had 7,765 rules.

To demonstrate more extreme cases of the usefulness of the on-the-fly approach, we built a language model that recognized exactly the 2,087 trees in the training corpus, each with equal weight. It had 39,455 rules. Finally, to be ultra-specific, we included a form of the "specific" language model just described, but only allowed the English counterpart of the particular Japanese sentence being decoded in the language.

The goal in our experiments is to apply a single tree backward through the cascade and find the 1-best path in the application wrtg. We evaluate based on the speed of each approach: bucket brigade and on-the-fly. The algorithm we use to obtain this 1-best path is a modification of the k-best algorithm of Pauls and Klein [108]. Our algorithm finds the 1-best path in a wrtg and admits an on-the-fly approach. We present a schema for this algorithm, analogous to the schemata shown in Pauls and Klein [108], in Table 4.6. [9]

The results of the experiments are shown in Table 4.7. As can be seen, on-the-fly application was generally faster than the bucket brigade, about double the speed per

[9] We only show the 1-best variant here but a k-best variant is easily obtained, in the manner shown by Pauls and Klein [108].

LM type	Method	Time/Sentence
pcfg	bucket	28s
pcfg	otf	17s
exact	bucket	>1m
exact	otf	24s
1-sent	bucket	2.5s
1-sent	otf	.06s

Table 4.7: Timing results to obtain 1-best from application through a weighted tree transducer cascade, using on-the-fly vs. bucket brigade backward application techniques. pcfg = model recognizes any tree licensed by a pcfg built from observed data, exact = model recognizes each of 2,000+ trees with equal weight, 1-sent = model recognizes exactly one tree.

sentence in the traditional experiment that used an English PCFG language model. The results for the other two language models demonstrate more keenly the potential advantage that can be had using an on-the-fly approach—the simultaneous incorporation of information from all models allows application to be done more effectively than if each information source is considered in sequence. In the "exact" case, where a very large language model that simply recognizes each of the 2,087 trees in the training corpus is used, the final application is so large that it overwhelms the resources of a 4gb MacBook Pro. In this case, the on-the-fly approach is necessary to avoid running out of memory. The "1-sent" case is presented to demonstrate the ripple effect caused by using on-the fly. In the other two cases, a very large language model generally overwhelms the timing statistics, regardless of the method being used. But a language model that represents exactly one sentence is very small, and thus the effects of simultaneous inference are readily apparent—the time to retrieve the 1-best sentence is reduced by two orders of magnitude in this experiment.

4.7 Backward application of wxLNTs to strings

Tree-to-string transducers are quite important in modern syntax NLP systems. We frequently are given string data and want to transform it into a forest of trees by means of some grammar or transducer. This is generally known as *parsing*, but from our perspective, it is the inverse image of a tree-to-string transducer applied to a string. Despite the change in nomenclature, though, we can take advantage of the rich parsing literature in defining an algorithm for this problem. A good choice for a parsing strategy is Earley's algorithm [36], and we will look to Stolcke's extension to the weighted case [124] for guidance, though since we are required to build an entire parse chart, and ultimately preserve weights from our input transducer, strictly speaking weights are not needed in the application algorithm.

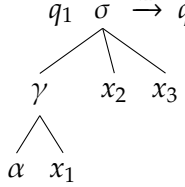
Stolcke's presentation [124] builds a parse forest from a wcfg.[10] Given a wcfg and a string of k words, a hypergraph, commonly called a *chart* is built. The states in the chart are represented as (p, v, i, j) tuples, where p is a wcfg production, v is a position, i.e., an index into the right side of p, and i, j is a pair of integers, $0 \leq i \leq j \leq k$, signifying a range of the input string. The hyperedges of the chart are unary or binary and are unlabeled. For a unary edge, if the destination state of an edge is of the form (p, v, i, j) where $v > 0$, then its source state is of the form $(p, v-1, i, j-1)$. For a binary edge, the right source state will be of the form (p', v', h, j), where p' has as its left nonterminal the nonterminal at the vth position of the right side of p, v' is equal to the length of the right side of p', and $i \leq h \leq j$. The left source state will be of the form $(p, v-1, i, h)$. A state of the form $(p, 0, i, i)$ has no incoming hyperedges.

[10] We alter the presentation of indices somewhat from Stolcke's approach but still use it for guidance.

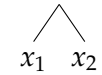
We can build a chart with a xLNTs and string in the same way as we build one for a wcfg, treating a rule of the form $q.y \xrightarrow{w} g$ as if it were a production of the form $q \xrightarrow{w} g'$, where g' is modified from g by replacing all (q, x) items with q. The left sides and variables are ignored, however, not omitted. After the chart is formed, it is traversed top down, and at each state q' of the form (r, v, i, j) where r is of the form $q.y \xrightarrow{w} g$ and v is equal to the length of g, a set of state sequences is formed, by appending, for each binary edge arriving at q', the right source state with each of the sequences formed from the left source state. Terminal states form no sequences, and unary edges form the sequences in their source state. Each of these sequences is assigned to the variable attached to the original g, and this is used, with y, to form a production. Additionally, to reduce the state space, all states formed from rules with the same left side, with positions at the right extreme, and with the same covering span are merged. In this way the domain projection is formed. Note that the result of this operation can then be the input to the backward application of a wxLT, and by this we may accomplish bucket brigade application of a cascade of wxLT, followed by a wxLNTs and a string.

Example 4.7.1 Consider the wxLNTs M_1 from Example 2.4.3, whose rules are reproduced in Figure 4.9a. To apply this transducer to the output string $\lambda\ \lambda\ \lambda\ \lambda$, first Earley's algorithm is used to form a parse chart, a portion of which is shown in Figure 4.10. Then, the chart is explored from the top down and descendant state sequences are gathered at each state that represents a fully-covered rule (such states are highlighted in Figure 4.10). States representing rules with common left sides that cover the same span are merged and the wrtg projected is depicted in Figure 4.9b.

1. $q_1\ \sigma \xrightarrow{.1} q_2.x_1\ q_2.x_2\ q_2.x_3$



2. $q_2\ \gamma \xrightarrow{.2} q_2.x_1\ q_2.x_2$

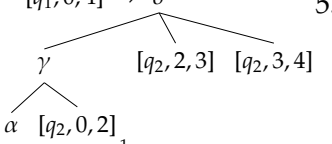


3. $q_2\ \beta \xrightarrow{.3} \lambda$

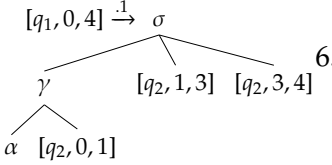
4. $q_2\ \alpha \xrightarrow{.4} \lambda$

(a) Rules from example xLNTs used for parsing, and originally seen in Figure 2.17.

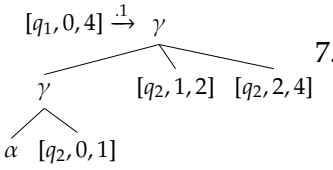
1. $[q_1, 0, 4] \xrightarrow{.1} \sigma$



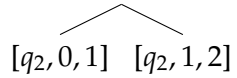
2. $[q_1, 0, 4] \xrightarrow{.1} \sigma$



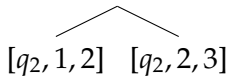
3. $[q_1, 0, 4] \xrightarrow{.1} \gamma$



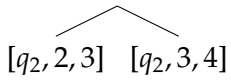
4. $[q_2, 0, 2] \xrightarrow{.2} \gamma$



5. $[q_2, 1, 3] \xrightarrow{.2} \gamma$



6. $[q_2, 2, 4] \xrightarrow{.2} \gamma$



7. $[q_2, 0, 1] \xrightarrow{.3} \beta$

8. $[q_2, 1, 2] \xrightarrow{.3} \beta$

9. $[q_2, 2, 3] \xrightarrow{.3} \beta$

10. $[q_2, 3, 4] \xrightarrow{.3} \beta$

11. $[q_2, 0, 1] \xrightarrow{.4} \alpha$

12. $[q_2, 1, 2] \xrightarrow{.4} \alpha$

13. $[q_2, 2, 3] \xrightarrow{.4} \alpha$

14. $[q_2, 3, 4] \xrightarrow{.4} \alpha$

(b) Backward application rules extracted from the chart in Figure 4.10 after state merging.

Figure 4.9: Input wxLNTs and final backward application wrtg formed from parsing $\lambda\ \lambda\ \lambda\ \lambda$, as described in Example 4.7.1.

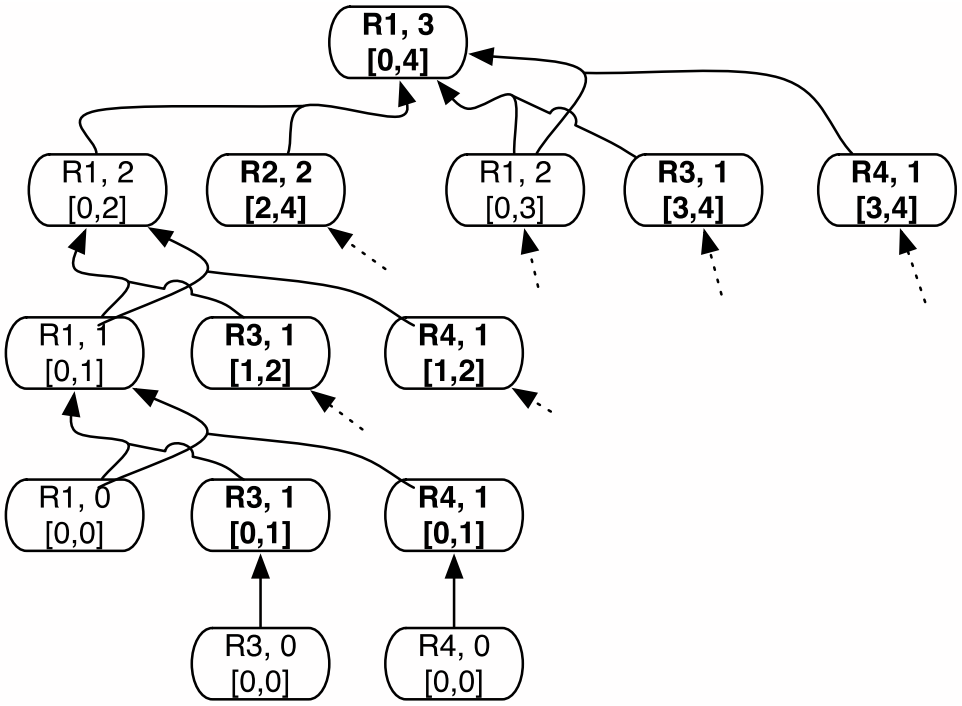
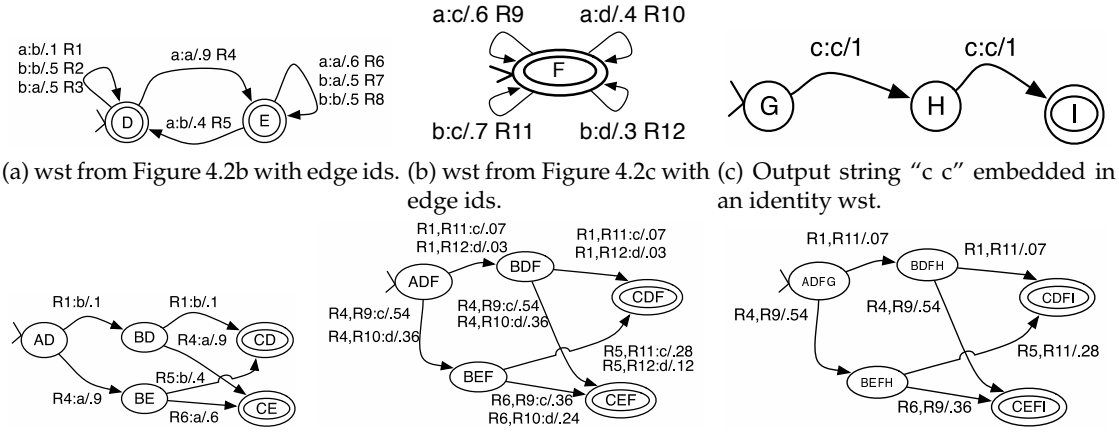


Figure 4.10: Partial parse chart formed by Earley's algorithm applied to the rules in Figure 4.9a, as described in Example 4.7.1. A state is labeled by its rule id, covered position of the rule right side, and covered span. Bold face states have their right sides fully covered, and are thus the states from which application wrtg productions are ultimately extracted. Dashed edges indicate hyperedges leading to sections of the chart that are not shown.



(a) wst from Figure 4.2b with edge ids.

(b) wst from Figure 4.2c with edge ids.

(c) Output string "c c" embedded in an identity wst.

(d) Derivation wsa construction after input embed (Figure 4.2a) is composed with first wst.

(e) Derivation wsa construction after composition with second wst.

(f) Final derivation wsa construction after composition with output embed and projection.

Figure 4.11: Construction of a derivation wsa.

4.8 Building a derivation wsa

As motivated in Section 4.1, another useful structure we may want to infer is a *derivation automaton (or grammar)*. Let's return to the string world for a moment. Given a string pair (i, o), a wst M and unique identifiers for each of the edges in M, a derivation wsa is a wsa representing the (possibly infinite) sequences of edges in M that derive o from i.[11] It can be used by forward-backward training algorithms to learn weights on a wst to maximize the likelihood of a string corpus [38]. Forming a derivation wsa is in fact similar to forming an application wsa, with an additional relabeling to keep track of edge ids. First, we embed i in an identity wst, I. We then compose I with M, but replace the input label of every edge formed via the use of some wst edge with its id instead. Then we compose this result with the embedding of o in an identity output wst, O. The domain projection of the result is a wsa representing the sequences of wst ids needed to derive o from i using the wst.

One may also want to form a derivation wsa for a cascade of wsts. In this case, the edges of the derivation wsa will contain ids from each of the transducers in the cascade. Again, the procedure is analogous to that used for forming an application wsa from a cascade, and the three strategies (offline composition, bucket brigade, and on-the-fly) each apply.[12] Figure 4.11 shows the construction of a derivation wsa from the wst cascade of Figure 4.2, now augmented with edge ids for ease of understanding.

[11] Derivation wsas are typically formed from unweighted wsts, but the generalization holds.
[12] Offline composition requires a bit of fancy bookkeeping to maintain edge ids. See Eisner [38].

4.9 Building a derivation wrtg

Turning to the tree case, the reader may be pleasantly surprised to learn that the restrictions for application wrtg construction do not apply for basic derivation wrtg construction—a derivation wrtg can be formed from a tree pair (i, o) and a wtt M of the most general class allowed—wxT. An algorithm to do this was first proposed by Graehl and Knight [55]. The theoretical justification for this is as follows: As in the string case, if we can compose I, M, and O, where I and O are embeddings of I and O, respectively, and perform the same kind of relabeling done for strings, we can form the derivation wrtg.

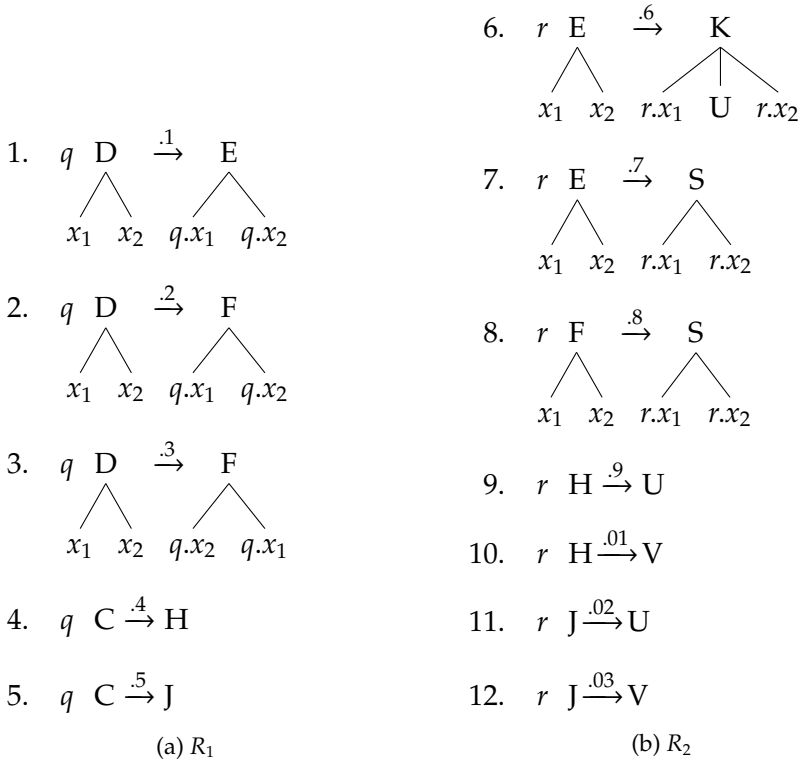
First let us consider $I = (Q, \Sigma, \Sigma, R, q_0)$ and $M = (Q', \Sigma, R', q'_0)$. The standard embedding of i forms a deterministic relabeling wtt. We can form $I' = (Q, \Sigma, \Sigma \cup \Upsilon, R \cup R_\Upsilon, q_0)$ where $\Upsilon \cap \Sigma = \emptyset$ and $\Upsilon = \{v_i\}, v_i \in \Upsilon^{(i)}$ for $0 \leq i \leq \max(rk(\sigma) | \sigma \in \Sigma)$. The rules of R_Υ are defined as follows: For each $\sigma \in \Sigma^{(k)}$ and $q \in Q$, if there is no rule of the form $q.\sigma \xrightarrow{w} t$ in R, add $q.\sigma \xrightarrow{1} v_k(q.x_0, ..., q.x_k)$ to R_Υ. We then take the input alphabet of M to be $\Sigma \cup \Upsilon$. It is clear that I' is a deterministic and *total* relabeling wtt and that $\tau_{I'}; \tau_M = \tau_I; \tau_M$ since the only effect of augmenting I to I' is to produce extra outputs that are not accepted by M. Following the principles of Baker [6], Theorem 1, $I' \circ M \subseteq wxT$, as I' is total and deterministic. The composition can be modified to use rule ids on the input labels without any problems, since the nature of the composition ensures that exactly one rule from M is used to form a rule in $I' \circ M$. Augmenting the composition with this replacement results in a transducer of type wT, since an extended left side is replaced with a single symbol,

denoting a rule id of M, with appropriate rank. Since O is in wLNT we can immediately produce $I' \circ M \circ O$, and by taking the domain projection we have a derivation wrtg.

A sufficient condition for forming a derivation wrtg for a cascade of wtts and a training pair is that the members of the cascade preserve recognizability. Then we can use a modification of the embed-compose-project approach to build the derivation wrtg.

Essentially, we modify the composition algorithm, Algorithm 13 from Section 2.3.3, such that the traditional creation of a composition rule, at line 13, is altered by creating a new symbol to replace y that contains the set of rules used to construct z (which may be inferred from θ, the mapping of tree positions to states) and has an appropriate rank. This transducer will be deterministic and, with the addition of sufficient rules that match no subsequent transducer, total, so we are ensured, by the conditions set forth by Baker [6], that this wtt can be composed with the next wtt in the cascade. Note that no projection or embedding is needed in this case. After the process has been repeated (unioning the obtained rule sequences with the contents of the special symbols formed previously), a projection may be taken, forming the derivation wrtg. The following example uses the approach just described, though modifications of the on-the-fly algorithms in this chapter can also be used for this construction, by simply adjoining the created rules with placeholders for rule sequence information.

Example 4.9.1 Consider the wtts $M_1 = (\{q\}, \Sigma, \Delta, R_1, q)$ and $M_2 = (\{r\}, \Delta, \Gamma, R_2, r)$ where Σ = {C, D}, Δ = {H, J, E, F}, Γ = {U, V, S, K}, and R_1 and R_2 are presented in Figures 4.12a and 4.12b, respectively. We would like to build a derivation wrtg for the pair (D(C, C), S(U, V)). Figure 4.13 shows productions in the intermediate application wrtgs using



(a) R_1 (b) R_2

Figure 4.12: Input transducers for cascade training.

- n_0 $\{\}^2$ $\xrightarrow{1}$ D
  x_1 x_2 $n_1.x_1$ $n_2.x_2$
- n_1 $\{\}^0 \xrightarrow{1}$ C
- n_2 $\{\}^0 \xrightarrow{1}$ C

(a) Converted and embedded wrtg of input tree.

- n_0q $\{1\}$ $\xrightarrow{.1}$ E
  x_1 x_2 $n_1q.x_1$ $n_2q.x_2$
- n_0q $\{2\}$ $\xrightarrow{.2}$ F
  x_1 x_2 $n_1q.x_1$ $n_2q.x_2$
- n_0q $\{3\}$ $\xrightarrow{.3}$ F
  x_1 x_2 $n_2q.x_2$ $n_1q.x_1$
- n_1q $\{4\} \xrightarrow{.4}$ H
- n_1q $\{5\} \xrightarrow{.5}$ J
- n_2q $\{4\} \xrightarrow{.4}$ H
- n_2q $\{5\} \xrightarrow{.5}$ J

(b) Application onto M_1.

- n_0qr $\{1, 6\}$ $\xrightarrow{.06}$ K
  x_1 x_2 $n_1qr.x_1$ U $n_2qr.x_2$
- n_0qr $\{2, 8, 1, 7\}$ $\xrightarrow{.23}$ S
  x_1 x_2 $n_1qr.x_1$ $n_2qr.x_2$
- n_0qr $\{3, 8\}$ $\xrightarrow{.24}$ S
  x_1 x_2 $n_2qr.x_2$ $n_1qr.x_1$
- n_1qr $\{4, 5, 9, 11\} \xrightarrow{.37}$ U
- n_1qr $\{4, 5, 10, 12\} \xrightarrow{.019}$ V
- n_2qr $\{4, 5, 9, 11\} \xrightarrow{.37}$ U
- n_2qr $\{4, 5, 10, 12\} \xrightarrow{.019}$ V

(c) Application onto M_2.

Figure 4.13: Progress of building derivation wrtg.

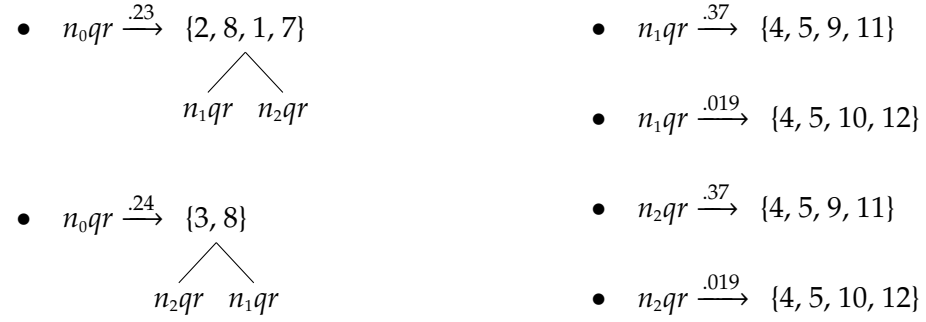


Figure 4.14: Derivation wrtg after final combination and conversion.

embed-compose-project, but as discussed above, the derivation wrtg, which is depicted in Figure 4.14, can be built using any valid application method.

4.10 Summary

We have presented, for the first time, algorithms for backward and forward application of cascades of weighted extended top-down tree-to-tree and tree-to-string transducers to tree grammar and string input. We have presented novel on-the-fly algorithms for application of tree transducer cascades and we have demonstrated the performance of these algorithms. We have also described how to use these algorithms to construct a derivation grammar for training a cascade of tree transducers that uses the application algorithms.

Chapter 5

Syntactic Re-Alignment Models for Machine Translation

In this chapter we use a wxtst framework and the tree transducer training algorithm described in [55, 56] for significant improvements in state-of-the-art syntax-based machine translation. Specifically, we present a method for improving word alignment that employs a syntactically informed alignment model closer to the translation model than commonly-used word alignment models. This leads to extraction of more useful linguistic patterns and improved BLEU scores on translation experiments in Chinese and Arabic. This work was first presented in [97] and was presented again as one component in a presentation of EM-based data-manipulation techniques to improve syntax MT in [132].

5.1 Methods of statistical MT

Roughly speaking, there are two paths commonly taken in statistical machine translation (Figure 5.1). The idealistic path uses an unsupervised learning algorithm such as EM [28] to learn parameters for some proposed translation model from a bitext training corpus, and then directly translates using the weighted model. Some examples of the

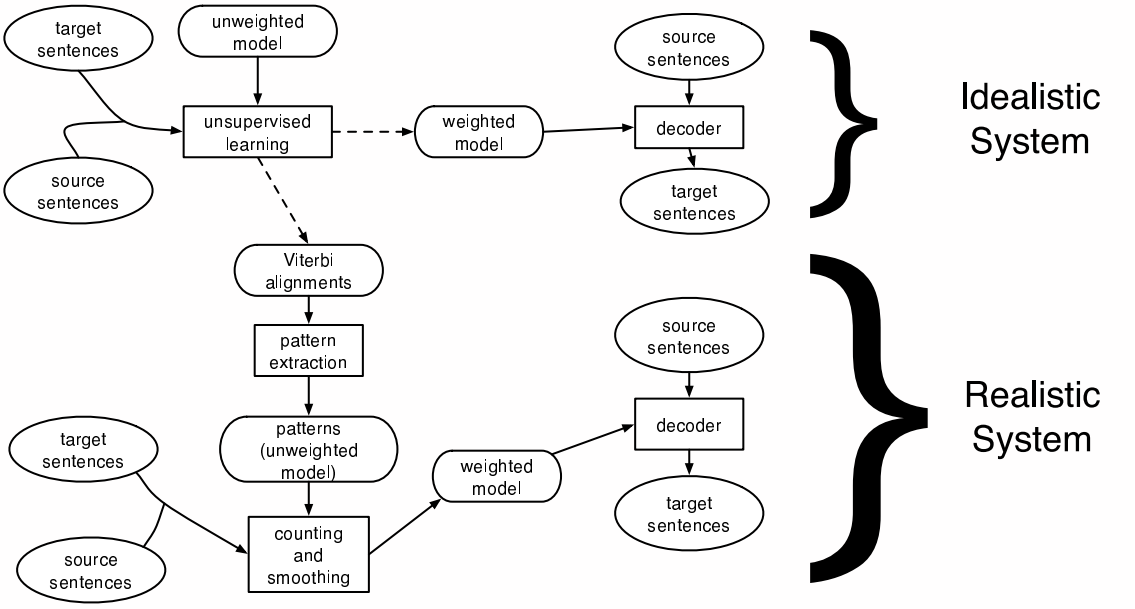


Figure 5.1: General approach to idealistic and realistic statistical MT systems.

idealistic approach are the direct IBM word model [7, 51], the phrase-based approach of Marcu and Wong [93], and the syntax approaches of Wu [134] and Yamada and Knight [136]. Idealistic approaches are conceptually simple and thus easy to relate to observed phenomena. However, as more parameters are added to the model the idealistic approach has not scaled well, for it is increasingly difficult to incorporate large amounts of training data efficiently over an increasingly large search space. Additionally, the EM procedure has a tendency to overfit its training data when the input units have varying explanatory powers, such as variable-size phrases or variable-height trees.

The realistic path also learns a model of translation, but uses that model only to obtain Viterbi word-for-word alignments for the training corpus. The bitext and corresponding alignments are then used as input to a pattern extraction algorithm, which yields a set of

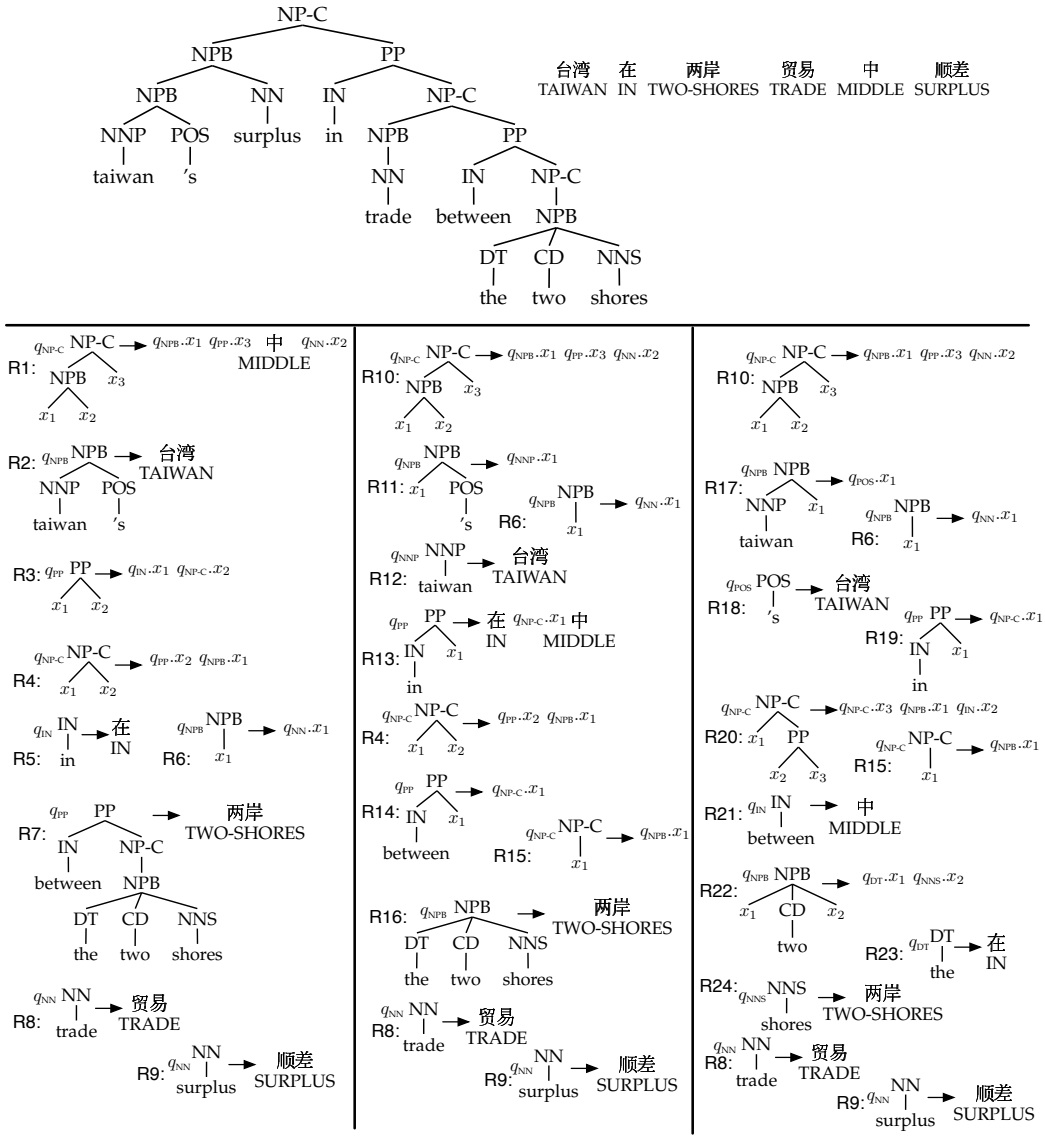


Figure 5.2: A (English tree, Chinese string) pair and three different sets of multilevel tree-to-string rules that can explain it; the first set is obtained from bootstrap alignments, the second from this paper's re-alignment procedure, and the third is a viable, if poor quality, alternative that is not learned.

patterns or rules for a second translation model (which often has a wider parameter space than that used to obtain the word-for-word alignments). Weights for the second model are then set, typically by counting and smoothing, and this weighted model is used for translation. Realistic approaches scale to large data sets and have yielded better BLEU performance than their idealistic counterparts, but there is a disconnect between the first model (hereafter, the *alignment model*) and the second (the *translation model*). Examples of realistic systems are the phrase-based ATS system of Och and Ney [107], the phrasal-syntax hybrid system Hiero [21], and the GHKM syntax system [47, 46]. For an alignment model, most of these use the Aachen HMM approach [131], the implementation of IBM Model 4 in GIZA++ [106] or, more recently, the semi-supervised EMD algorithm [42].

The two-model approach of the realistic path has undeniable empirical advantages and scales to large data sets, but new research tends to focus on development of higher order translation models that are informed only by low-order alignments. We would like to add the analytic power gained from modern translation models to the underlying alignment model without sacrificing the efficiency and empirical gains of the two-model approach. By adding the syntactic information used in the translation model to our alignment model we may improve alignment quality such that rule quality and, in turn, system quality are improved. In the remainder of this work we show how a touch of idealism can improve an existing realistic syntax-based translation system.

5.2 Multi-level syntactic rules for syntax MT

Galley et al. [47] and Galley et al. [46] describe a syntactic translation model that relates English trees to foreign strings via a tree-to-string transducer of class wxLNTs. The model describes joint production of a (tree, string) pair via a non-deterministic selection of weighted rules. Each rule has an English tree fragment with variables and a corresponding foreign string fragment with the same variables. A series of rules forms an explanation (or *derivation*) of the complete pair.

As an example, consider the parsed English and corresponding Chinese at the top of Figure 5.2. The three columns underneath the example are different rule sequences that can explain this pair; there are many other possibilities. Note how rules specify rotation (e.g., R10, R4), direct translation (R12, R8), insertion and deletion (R11, R1), and tree traversal (R6, R15). Note too that the rules explain variable-size fragments (e.g., R7 vs. R14) and thus the possible *derivation trees* of rules that explain a sentence pair have varying sizes. The smallest such derivation tree has a single large rule (which does not appear in Figure 5.2; we leave the description of such a rule as an exercise for the reader). A string-to-tree decoder constructs a *derivation forest* of derivation trees where the right sides of the rules in a tree, taken together, explain a candidate source sentence. It then outputs the English tree corresponding to the highest-scoring derivation in the forest.

5.3 Introducing syntax into the alignment model

We now lay the groundwork for a syntactically motivated alignment model. We begin by reviewing an alignment model commonly seen in realistic MT systems and compare it to a syntactically-aware alignment model.

5.3.1 The traditional IBM alignment model

IBM Model 4 [16] learns a set of 4 probability tables to compute $p(f|e)$ given a foreign sentence f and its target translation e via the following (greatly simplified) generative story:

1. A fertility y for each word e_i in e is chosen with probability $p_{fert}(y|e_i)$.

2. A null word is inserted next to each fertility-expanded word with probability p_{null}.

3. Each token e_i in the fertility-expanded word and null string is translated into some foreign word f_i in f with probability $p_{trans}(f_i|e_i)$.

4. The position of each foreign word f_i that was translated from e_i is changed by Δ (which may be positive, negative, or zero) with probability $p_{distortion}(\Delta|\mathcal{A}(e_i), \mathcal{B}(f_i))$, where $\mathcal{A}$ and $\mathcal{B}$ are functions over the source and target vocabularies, respectively.

Brown et al. [16] describe an EM algorithm for estimating values for the four tables in the generative story. However, searching the space of all possible alignments is intractable for EM, so in practice the procedure is bootstrapped by models with narrower search space such as IBM Model 1 [16] or Aachen HMM [131].

5.3.2 A syntax re-alignment model

Now let us contrast this commonly used model for obtaining alignments with a syntactically motivated alternative. We recall the rules described in Section 5.2. Our model learns a single probability table to compute $p(etree, f)$ given a foreign sentence f and a parsed target translation *etree*. In the following generative story we assume a starting variable with syntactic type v.

1. Choose a rule r to replace v, with probability $p_{rule}(r|v)$.

2. For each variable with syntactic type v_i in the partially completed (tree, string) pair, continue to choose rules r_i with probability $p_{rule}(r_i|v_i)$ to replace these variables until there are no variables remaining.

In Section 5.5.1 we discuss an EM learning procedure for estimating these rule probabilities.

As in the IBM approach, we must mitigate intractability by limiting the parameter space searched, which is potentially much wider than in the word-to-word case. We would like to supply to EM all possible rules that explain the training data, but this implies a rule relating each possible tree fragment to each possible string fragment, which is infeasible. We follow the approach of bootstrapping from a model with a narrower parameter space as is done by, e.g., Och and Ney [106] and Fraser and Marcu [42].

To reduce the model space we employ the rule acquisition technique of Galley et al. [47], which obtains rules given a (tree, string) pair as well as an initial alignment between them. We are agnostic about the source of this bootstrap alignment and in Section 5.5

present results based on several different bootstrap alignment qualities. We require an initial set of alignments, which we obtain from a word-for-word alignment procedure such as GIZA++ or EMD. Thus, we are not aligning input data, but rather *re-aligning* it with a syntax model.

5.4 The appeal of a syntax alignment model

Consider the example of Figure 5.2 again. The leftmost derivation is obtained from the bootstrap alignment set. This derivation is reasonable but there are some poorly motivated rules, from a linguistic standpoint. The third word in the Chinese sentence roughly means "the two shores" in this context, but the rule R7 learned from the alignment incorrectly includes "between". However, other sentences in the training corpus have the correct alignment, which yields rule R16. Meanwhile, rules R13 and R14, learned from yet other sentences in the training corpus, handle the second and fifth Chinese words (which, as a unit, translates to "in between"), thus allowing the middle derivation.

EM distributes rule probabilities in such a way as to maximize the probability of the training corpus. It thus prefers to use one rule many times instead of several different rules for the same situation over several sentences, if possible. R7 is a possible rule in 46 of the 329,031 sentence pairs in the training corpus, while R16 is a possible rule in 100 sentence pairs. Well-formed rules are more usable than ill-formed rules and the partial alignments behind these rules, generally also well-formed, become favored as well. The top row of Figure 5.3 contains an example of an alignment learned by the bootstrap alignment model that includes an incorrect link. Rule R24, which is extracted from this

	description	sentence pairs	
		Chinese	Arabic
Tune	NIST 2002 short	925	696
Test	NIST 2003	919	663

Table 5.1: Tuning and testing data sets for the MT system described in Section 5.5.2.

bootstrap GIZA corpus		re-alignment experiment			
English words	Chinese words	type	rules	tune	test
9,864,294	7,520,779	baseline	19,138,252	39.08	37.77
		initial	18,698,549	39.49	38.39
		adjusted	26,053,341	**39.76**	**38.69**

Table 5.2: A comparison of Chinese BLEU performance between the GIZA baseline (no re-alignment), re-alignment as proposed in Section 5.3.2, and re-alignment as modified in Section 5.5.4.

alignment, is a poor rule. A set of commonly seen rules learned from other training sentences provide a more likely explanation of the data, and the consequent alignment omits the spurious link.

5.5 Experiments

In this section, we describe the implementation of our semi-idealistic model and our means of evaluating the resulting re-alignments in an MT task.

5.5.1 The re-alignment setup

We begin with a training corpus of Chinese-English and Arabic-English bitexts, the English side parsed by a reimplementation of the standard Collins model [9]. In order to acquire a syntactic rule set, we also need a bootstrap alignment of each training sentence. We use an implementation of the GHKM algorithm [47] to obtain a rule set for each bootstrap alignment.

Now we need an EM algorithm for learning the parameters of the rule set that maximize $\prod_{corpus} p(tree, string)$. Such an algorithm is presented by Graehl et al. [56]. The algorithm consists of two components: DERIV, which is a procedure for constructing a packed forest of derivation trees of rules that explain a (tree, string) bitext corpus given that corpus and a rule set, and TRAIN, which is an iterative parameter-setting procedure.

We initially attempted to use the top-down DERIV algorithm of Graehl et al. [56], but as the constraints of the derivation forests are largely lexical, too much time was spent on exploring dead-ends. Instead we build derivation forests using the following sequence of operations:

1. Binarize rules using the synchronous binarization algorithm for tree-to-string transducers described by Zhang et al. [138].

2. Construct a parse chart with a CKY parser simultaneously constrained on the foreign string and English tree, similar to the bilingual parsing of Wu [135] [1].

3. Recover all reachable edges by traversing the chart, starting from the topmost entry.

Since the chart is constructed bottom-up, leaf lexical constraints are encountered immediately, resulting in a narrower search space and faster running time than the top-down DERIV algorithm for this application. Derivation forest construction takes around 400 hours of cumulative machine time (4-processor machines) for Chinese. The actual running of EM iterations (which directly implements the TRAIN algorithm of Graehl et al. [56]) takes about 10 minutes, after which the Viterbi derivation trees are directly

[1] In the cases where a rule is not synchronous-binarizable standard left-right binarization is performed and proper permutation of the disjoint English tree spans must be verified when building the part of the chart that uses this rule.

bootstrap GIZA corpus		re-alignment experiment			
English words	Chinese words	type	rules	tune	test
9,864,294	7,520,779	baseline	19,138,252	39.08	37.77
		re-alignment	26,053,341	**39.76**	**38.69**
221,835,870	203,181,379	baseline	23,386,535	39.51	38.93
		re-alignment	33,374,646	**40.17**	**39.96**

(a) Chinese re-alignment corpus has 9,864,294 English and 7,520,779 Chinese words.

bootstrap GIZA corpus		re-alignment experiment			
English words	Arabic words	type	rules	tune	test
4,067,454	3,147,420	baseline	2,333,839	**47.92**	47.33
		re-alignment	2,474,737	47.87	**47.89**
168,255,347	147,165,003	baseline	3,245,499	49.72	49.60
		re-alignment	3,600,915	49.73	**49.99**

(b) Arabic re-alignment corpus has 4,067,454 English and 3,147,420 Arabic words.

Table 5.3: Machine Translation experimental results evaluated with case-insensitive BLEU4.

recoverable. The Viterbi derivation tree tells us which English words produce which Chinese words, so we can extract a word-to-word alignment from it. We summarize the approach described in this paper as:

1. Obtain bootstrap alignments for a training corpus using GIZA++.

2. Extract rules from the corpus and alignments using GHKM, noting the partial alignment that is used to extract each rule.

3. Construct derivation forests for each (tree, string) pair, ignoring the alignments, and run EM to obtain Viterbi derivation trees, then use the annotated partial alignments to obtain Viterbi alignments.

4. Use the new alignments as input to the MT system described below.

5.5.2 The MT system setup

A truly idealistic MT system would directly apply the rule weight parameters learned via EM to a machine translation task. As mentioned in Section 5.1, we maintain the two-model, or realistic approach. Below we briefly describe the translation model, focusing on comparison with the previously described alignment model. Galley et al. [46] provide a more complete description of the translation model and DeNeefe et al. [31] provide a more complete description of the end-to-end translation pipeline.

Although in principle the re-alignment model and translation model learn parameter weights over the same rule space, in practice we limit the rules used for re-alignment to the set of smallest rules that explain the training corpus and are consistent with the bootstrap alignments. This is a compromise made to reduce the search space for EM. The translation model learns multiple derivations of rules consistent with the re-alignments for each sentence, and learns weights for these by counting and smoothing. A dozen other features are also added to the rules. We obtain weights for the combinations of the features by performing minimum error rate training [105] on held-out data. We then use a CKY decoder to translate unseen test data using the rules and tuned weights. Table 5.1 summarizes the data used in tuning and testing.

5.5.3 Initial results

An initial re-alignment experiment shows a reasonable rise in BLEU scores from the baseline (Table 5.2), but closer inspection of the rules favored by EM implies we can do even better. EM has a tendency to favor few large rules over many small rules, even when

LANGUAGE PAIR	TYPE	RULES	TUNE	TEST
CHINESE-ENGLISH	baseline	55,781,061	**41.51**	40.55
	EMD re-align	69,318,930	41.23	40.55
ARABIC-ENGLISH	baseline	8,487,656	**51.90**	51.69
	EMD re-align	11,498,150	51.88	**52.11**

Table 5.4: Re-alignment performance with semi-supervised EMD bootstrap alignments.

the small rules are more useful. Referring to the rules in Figure 5.2, note that possible derivations for translating between "taiwan 's" and the first word in the Chinese sentence are R2, R11-R12, and R17-R18. Clearly the third derivation is not desirable, and we do not discuss it further. Between the first two derivations, R11-R12 is preferred over R2, as the conditioning for possessive insertion is not related to the specific Chinese word being inserted. Of the 1,902 sentences in the training corpus where this pair is seen, the bootstrap alignments yield the R2 derivation 1,649 times and the R11-R12 derivation 0 times. Re-alignment does not change the result much; the new alignments yield the R2 derivation 1,613 times and again never choose R11-R12. The rules in the second derivation themselves are not rarely seen—R11 is in 13,311 forests *other* than those where R2 is seen, and R12 is in 2,500 additional forests. EM gives R11 a probability of $e^{-7.72}$—better than 98.7% of rules, and R12 a probability of $e^{-2.96}$. But R2 receives a probability of $e^{-6.32}$ and is preferred over the R11-R12 derivation, which has a combined probability of $e^{-10.68}$.

5.5.4 Making EM fair

The preference for shorter derivations containing large rules over longer derivations containing small rules is due to a general tendency for EM to prefer derivations with few atoms. Marcu and Wong [93] note this preference but consider the phenomenon a

feature, rather than a bug. Zollmann and Sima'an [139] combat the overfitting aspect for parsing by using a held-out corpus and a straight maximum likelihood estimate, rather than EM. We take a modeling approach to the phenomenon.

As the probability of a derivation is determined by the product of its atom probabilities, longer derivations with more probabilities to multiply have an inherent disadvantage against shorter derivations, all else being equal. EM is an iterative procedure and thus such a bias can lead the procedure to converge with artificially raised probabilities for short derivations and the large rules that comprise them. The relatively rare applicability of large rules (and thus lower observed partial counts) does not overcome the inherent advantage of large coverage. To combat this, we introduce size terms into our generative story, ensuring that all competing derivations for the same sentence contain the same number of atoms:

1. Choose a rule size s with cost $c_{size}(s)^{s-1}$.
2. Choose a rule r (of size s) to replace the start symbol with probability $p_{rule}(r|s, v)$.
3. For each variable in the partially completed (tree, string) pair, continue to choose sizes followed by rules, recursively to replace these variables until there are no variables remaining.

This generative story changes the derivation comparison from R2 vs R11-R12 to S2-R2 vs R11-R12, where S2 is the atom that represents the choice of size 2 (the size of a rule in this context is the number of non-leaf and non-root nodes in its tree fragment). Note that the variable number of inclusions implied by the exponent in the generative story above ensures that all derivations have the same size. For example, a derivation with

one size-3 rule, a derivation with one size-2 and one size-1 rule, and a derivation with three size-1 rules would each have three atoms. With this revised model that allows for fair comparison of derivations, the R11-R12 derivation is chosen 1636 times, and S2-R2 is not chosen. R2 does, however, appear in the translation model, as the expanded rule extraction described in Section 5.5.2 creates R2 by joining R11 and R12.

The probability of size atoms, like that of rule atoms, is decided by EM. The revised generative story tends to encourage smaller sizes by virtue of the exponent. This does not, however, simply ensure the largest number of rules per derivation is used in all cases. Ill-fitting and poorly-motivated rules such as R22, R23, and R24 in Figure 5.2 are not preferred over R16, even though they are smaller. However, R14 and R16 are preferred over R7, as the former are useful rules. Although the modified model does not sum to 1, it leads to an improvement in BLEU score, as can be seen in the last row of Table 5.2.

5.5.5 Results

We performed primary experiments on two different bootstrap setups in two languages: the initial experiment uses the same data set for the GIZA++ initial alignment as is used in the re-alignment, while an experiment on better quality bootstrap alignments uses a much larger data set. For each bootstrapping in each language we compared the baseline of using these alignments directly in an MT system with the experiment of using the alignments obtained from the re-alignment procedure described in Section 5.5.4. For each experiment we report: the number of rules extracted by the expanded GHKM algorithm of Galley et al. [46] for the translation model, converged BLEU scores on the

tuning set, and finally BLEU performance on the held-out test set. Data set specifics for the GIZA++ bootstrapping and BLEU results are summarized in Table 5.3.

5.5.6 Discussion

The results presented demonstrate we are able to improve on unsupervised GIZA++ alignments by about 1 BLEU point for Chinese and around 0.4 BLEU point for Arabic using an additional unsupervised algorithm that requires no human aligned data. If human-aligned data is available, the EMD algorithm provides higher baseline alignments than GIZA++ that have led to better MT performance [42]. As a further experiment we repeated the experimental conditions from Table 5.3, this time bootstrapped with the semi-supervised EMD method, which uses the larger bootstrap GIZA corpora described in Table 5.3 and an additional 64,469/48,650 words of hand-aligned English-Chinese and 43,782/31,457 words of hand-aligned English-Arabic. The results of this advanced experiment are in Table 5.4. We show a 0.42 gain in BLEU for Arabic, but no movement for Chinese. We believe increasing the size of the re-alignment corpora will increase BLEU gains in this experimental condition, but leave those results for future work.

We can see from the results presented that the impact of the syntax-aware re-alignment procedure of Section 5.3.2, coupled with the addition of size parameters to the generative story from Section 5.5.4 serves to remove links from the bootstrap alignments that cause less useful rules to be extracted, and thus increase the overall quality of the rules, and hence the system performance. We thus see the benefit to including syntax in an alignment model, bringing the two models of the realistic machine translation path somewhat closer together.

5.6 Conclusion

We have described a method for improving state-of-the-art syntax machine translation performance by casting a complicated mt system as a wxLNTs and employing transducer training algorithms to improve word alignment. This chapter demonstrates the real, practical gains suggested in Chapter 1.

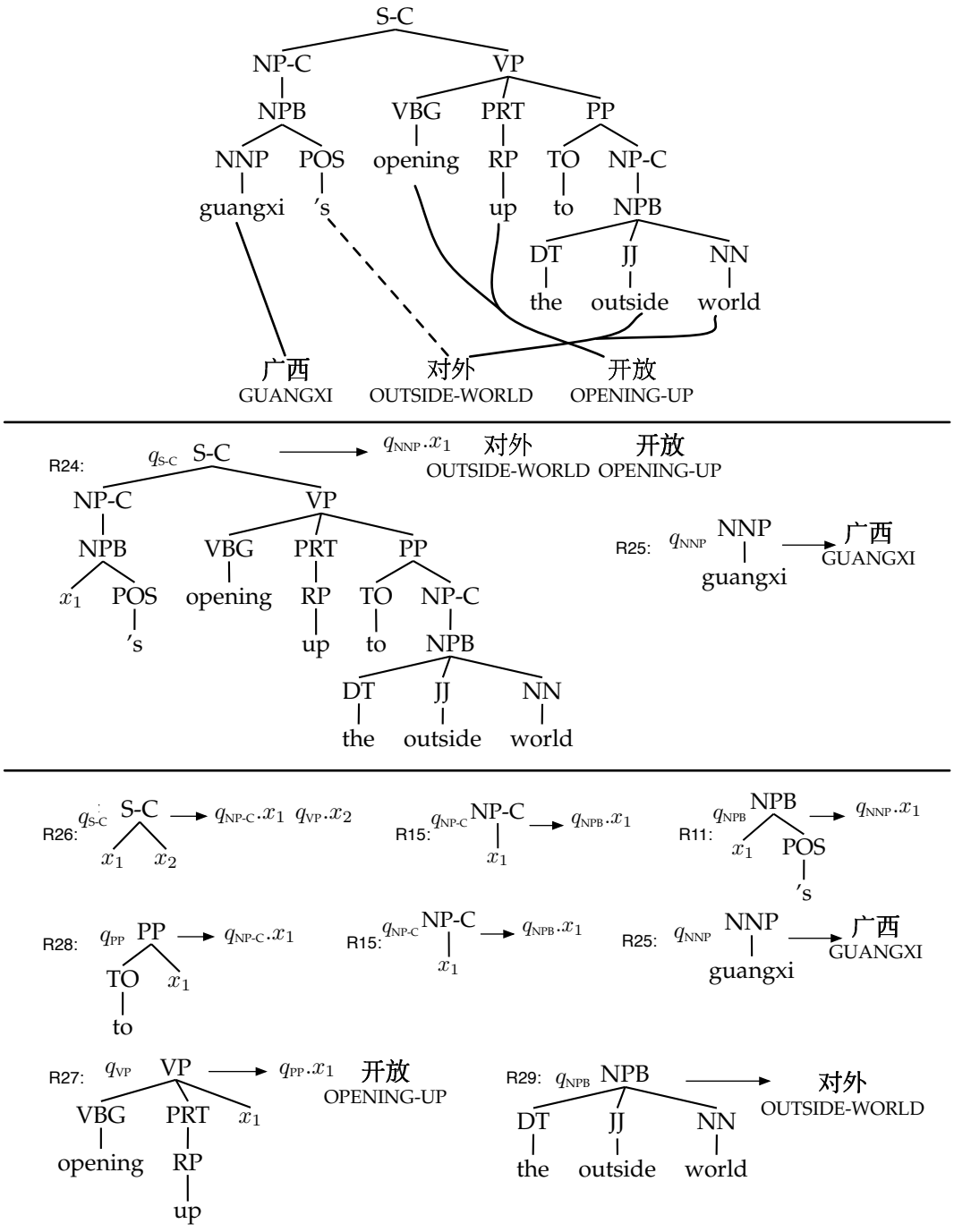


Figure 5.3: The impact of a bad alignment on rule extraction. Including the alignment link indicated by the dotted line in the example leads to the rule set in the second row. The re-alignment procedure described in Section 5.3.2 learns to prefer the rule set at bottom, which omits the bad link.

Chapter 6

Tiburon: A Tree Transducer Toolkit

In this chapter we describe Tiburon, a toolkit for manipulating weighted tree transducers and grammars. Tiburon contains implementations of many of the algorithms presented in previous chapters and is designed to be fairly intuitive and easy to use. We also place Tiburon in the context of other transducer and automata toolkits and software.

6.1 Introduction

The development of well-founded models of natural language processing applications has been greatly accelerated by the availability of toolkits for finite-state automata. The influential observation of Kaplan & Kay, that cascades of phonological rewrite rules could be expressed as regular relations (equivalent to finite-state transducers) [64], was exploited by Koskenniemi in his development of the two-level morphology and accompanying system for its representation [80]. This system, which was a general program for analysis and generation of languages, pioneered the field of finite-state toolkits [68].

Successive versions of the two-level compiler, such as that written by Karttunen and others at Xerox [67], were used for large-scale analysis applications in many languages

[68]. Continued advances, such as work by Karttunen in intersecting composition [71] and replacement [65, 66], eventually led to the development of the Xerox finite-state toolkit, which superseded the functionality and use of the two-level tools [68].

Meanwhile, interest in adding uncertainty to finite-state models grew alongside increased availability of large datasets and increased computational power. Ad-hoc methods and individual implementations were developed for integrating uncertainty into finite-state representations [115, 86], but the need for a general-purpose weighted finite-state toolkit was clear [103]. Researchers at AT&T led the way with their FSM Library [102] which represented weighted finite-state automata by incorporating the theory of semirings over rational power series cleanly into the existing automata theory. Other toolkits, such as van Noord's FSA utilities [130], the RWTH toolkit [62], and the USC/ISI Carmel toolkit [53], provided additional interfaces and utilities for working with weighted finite-state automata. As in the unweighted case, the availability of this software led to many research projects that took advantage of pre-existing implementations [61, 129, 78] and the development of the software led to the invention of new algorithms and theory [109, 99].

As has been described in Chapter 1, however, none of these toolkits are extendable to recognize syntactic structures. GRM, an extension of the AT&T toolkit that uses approximation theory to represent higher-complexity structure such as context-free grammars in the weighted finite-state string automata framework, was useful for handling certain representations [3], but a tree automata framework is required to truly capture tree models. Additionally, the incorporation of *weights* is crucial for modern natural language processing needs. Thus, contributions such as Timbuk [50], a toolkit for unweighted

finite state tree automata that has been used for cryptographic analysis, and MONA [58], an unweighted tree automata tool aimed at the logic community, were insufficient for our needs.

Knight and Graehl [76] put forward the case for the top-down tree automata theory of Rounds [116] and Thatcher [127] as a logical sequel to weighted string automata for NLP. Additionally, as Knight and Graehl mention [76], most of the desired general operations in a general weighted finite-state toolkit are applicable to top-down tree automata. Probabilistic tree automata were first proposed by Magidor and Moran [87]. Weighted tree transducers were first described by Fülöp and Vogler [44] as an operational representation of tree series transducers, first introduced by Kuich [82].

We present Tiburon, a toolkit designed in the spirit of its predecessors but with the tree, not the string, as its basic data structure and weights inherent in its operation. Tiburon is designed to be easy to construct automata and work with them—after reading this chapter a linguist with no computer science background or a computer scientist with only the vaguest notions of tree automata should be able to write basic acceptors and transducers. To achieve these goals we have maintained simplicity in data format design, such that acceptors and transducers are very close to the way they appear in tree automata literature. We also provide a small set of generic but powerful operations that allow robust manipulation of data structures with simple commands. In subsequent sections we present an introduction to the formats and operations in the Tiburon toolkit and demonstrate the powerful applications that can be easily built. Tiburon was first introduced in [96].

6.2 Getting started

Tiburon is written in Java and is distributed as a Java archive file (jar) with a simple bash wrapping script. After downloading the software, the command

```
% ./tiburon
```

produces output that looks like

```
This is Tiburon, version 1.0
Error: Parameter 'infiles' is required.
Usage: tiburon
             [-h|--help] (-e|--encoding) <encoding>
     (-m|--semiring) <srtype> [--leftapply]
     [--rightapply] [-b|--batch] [(-a|--align) <align>]
     [-l|--left] [-r|--right] [-n|--normalizeweight]
     [--no-normalize] [--removeloops] [--normform]
     [(-p|--prune) <prune>] [(-d|--determinize) <determ>]
     [(-t|--train) <train>] [(-x|--xform) <xform>]
     [--training-deriv-location <trainderivloc>]
     [--conditional] [--no-deriv] [--randomize]
     [--timedebug <time>] [-y|--print-yields]
     [(-k|--kbest) <kbest>] [(-g|--generate) <krandom>]
     [-c|--check] [(-o|--outputfile) <outfile>] infiles1
     infiles2 ... infilesN
```

The salient features of this output are that Tiburon was invoked, the program expected some input files, and the usage statement was displayed. Since input files are crucial to Tiburon's operation, we discuss them next. There are several file types Tiburon uses: *wrtg*, *wcfg*, *wxtt*, *wxtst*, and *batch*. With the exception of the batch file, each of these files corresponds to a formal structure described in Chapter 2. We describe these files next.

6.3 Grammars

In this section we describe file formats and operations on grammars, the wrtgs that recognize tree languages and the wcfgs that recognize context-free string languages.

6.3.1 File formats

Both kinds of grammar discussed in this thesis have a similar formal structure: (N, Σ, P, n_0)[1], though Σ and P have somewhat different meanings. Consequently, the grammar files, wrtg and wcfg, both have a similar overall format:[2]

<n0>

<prd>+

where <n0> is the start nonterminal and each <prd> is a member of P. A nonterminal can be most any alphanumeric sequence. To be safe, it should not contain the following reserved characters:[3]

. ( ) # @ % >

[1] We previously used Δ for the terminal alphabet of a wcfg, but this is an issue of terminology only

[2] We use a BNF-like syntax, but hope the reader will rely on the examples that follow for greater clarity.

[3] The file parser for Tiburon is brittle and if you try to break it, you will probably succeed.

The format of a production, <prd> is:

```
<nterm> "->" <rhs> ["#" <wgt>] ["@" <tie>]
```

where <nterm> is a nonterminal, <wgt> is a real number, and <tie> is an integer. Ties are rarely used and will be discussed later. If the file is a wrtg, then <rhs> has the following format:

```
<nterm> | <sym> | <sym>"("<rhs>+")"
```

where <sym> is a member of Σ and is subject to the same definitional constraints as the nonterminals. If the file is a cfg, then <rhs> has the following format:

```
"*e*" | (<nterm>|<sym>)+
```

Note that the nonterminal and terminal sets are thus defined *implicitly*; all nonterminals must appear as left sides of productions at least once, and the nonterminal and terminal alphabets must not coincide.

The symbol % denotes a comment, and all characters after this symbol to the end of the line are ignored, with one exception. Since a file can be ambiguously a wcfg or a wrtg, if the initial line of the file is one of:

```
% TYPE CFG
% TYPE RTG
```

then that type is presumed. It is best to further explain these formats with some examples, which are in Figure 6.1.

We will later discuss training, so we now describe batch files appropriate for training wrtgs and wcfgs. The format for batch files is

```
q3
q3 -> A(q1 q1) # .25
q3 -> A(q3 q2) # .25
q3 -> A(q2 q3) # .25
q3 -> B(q2)    # .25
q2 -> A(q2 q2) # .25
q2 -> A(q1 q3) # .25
q2 -> A(q3 q1) # .25
q2 -> B(q1)    # .25
q1 -> A(q3 q3) # .025
q1 -> A(q1 q2) # .025
q1 -> A(q2 q1) # .025
q1 -> B(q3)    # .025
q1 -> C        # .9
```

(a) `three.rtg` is a wrtg that recognizes trees with size divisible by three.

```
q
q -> np vp # 1
pp -> prep np # 1
vp -> vb do # 1
nn -> boy # .4
nn -> monkey # .1
nn -> clown # .5
np -> dt nn # .5
np -> dt nn pp # .5
dt -> the # 1
vb -> ran # 1
do -> *e* # .9
do -> home # .1
prep -> with # .6
prep -> by # .4
```

(b) `ran.cfg` is a wcfg that recognizes an infinite language.

```
qe
qe -> A(qe qo) # .1
qe -> A(qo qe) # .8
qe -> B(qo) # .1
qo -> A(qo qo) # .6
qo -> A(qe qe) # .2
qo -> B(qe) # .1
qo -> C # .1
```

(c) `even.rtg` is a wrtg that recognizes trees with size divisible by two.

```
q
q -> S(subj vb obj)
q -> S(subj likes obj)
obj -> candy
vb -> likes
vb -> hates
subj -> John
subj -> Stacy
```

(d) `candy.rtg` is a rtg that recognizes a finite language. It is not in normal form or deterministic.

```
TOP
S -> S-C , NP-C VP
S -> NP-C VP
S -> ADVP NP-C VP
S -> NP-C ADVP VP
...
NP-C -> NPB
NP-C -> NPB , VP
NP-C -> NPB NP
...
```

(e) Portion of `train.cfg`, a cfg used for training.

```
% TYPE RTG
qS_TOP+
qS_TOP+ -> qS_TOP # .2
qS_TOP+ -> qS # 0.8
qS -> S(qNP_S+ : qVP_S+ EOL) # 0.0009
qS -> S(qNP_S+ qVP_S+ : EOL) # 0.0010
qS -> S(qNP_S+ DT qVP_S+ EOL) # 0.0005
...
qS_TOP -> S(qNP_S+ , qVP_S+ EOL) # 0.0258
...
```

(f) Portion of `train.rtg`, a wrtg used for training.

Figure 6.1: Example wrtg and wcfg files used to demonstrate Tiburon's capabilities.

```
(<count>
<item>)+
|
(<item>)+
```

A batch file for training wrtgs is a file of trees. The format for <item> in this case is:

```
<sym> | <sym>"("<item>+")"
```

A batch file for training wcfgs is a file of strings. The format for <item> in this case is:

```
<sym>+
```

Here is a portion of a batch file of strings:

```
DT VBN NNS RB MD VB NNS TO VB NNS IN NNS RBR CC RBR RB EOL
IN PRP$ NNS TO VB RP IN DT NNS , DT NN RB MD VB DT NN EOL
NNS WP VBP TO VB DT JJ NN MD VB PRP$ NNS IN NNP , PRP VBD EOL
```

Here is a portion of a batch file of trees:

```
TOP(S(NP-C(NPB(NNP NNP)) VP(VBZ NP-C(NPB(NNP))) EOL))
TOP(S(NP-C(NPB(DT JJ NN)) VP(MD VP-C(VB ADJP(JJ NP(NPB(NNP CD))))) EOL))
TOP(S(NP-C(NPB(DT NN NN)) VP(VBZ RB VP-C(VBN VP-C(VBN))) EOL))
```

6.3.2 Commands using grammar files

If you write any of the example grammars in Figure 6.1 as a plain text file and then call Tiburon with that file as an argument, the contents of that file, minus any comments, and

with weights of 1 in place if no weight is specified, will be returned. If there is a syntax error, this will be reported instead. Here is an example of Tiburon reading `candy.rtg`, from Figure 6.1d:

```
tiburon candy.rtg
This is Tiburon, version 1.0
q
q -> S(subj vb obj) # 1.000000
q -> S(subj likes obj) # 1.000000
obj -> candy # 1.000000
vb -> likes # 1.000000
vb -> hates # 1.000000
subj -> John # 1.000000
subj -> Stacy # 1.000000
```

Tiburon automatically places the weight 1 on unweighted productions. The `--randomize` makes Tiburon randomly choose some weights, which can be useful in debugging scenarios:

```
tiburon --randomize candy.rtg
This is Tiburon, version 1.0
q
q -> S(subj vb obj) # 0.357130
q -> S(subj likes obj) # 0.362367
obj -> candy # 0.291670
```

```
vb -> likes # 0.280298

vb -> hates # 0.686882

subj -> John # 0.671051

subj -> Stacy # 0.377440
```

Since we often want to work with probabilistic grammars, where the sum of weights of productions with a common left nonterminal is 1, we could have also enforced the weights to be normalized by adding the -n flag:

```
tiburon --randomize -n candy.rtg

This is Tiburon, version 1.0

q

q -> S(subj vb obj) # 0.496361

q -> S(subj likes obj) # 0.503639

obj -> candy # 1.000000

vb -> likes # 0.289810

vb -> hates # 0.710190

subj -> John # 0.640016

subj -> Stacy # 0.359984
```

The -k <kbest> option, where <kbest> is an integer, returns the trees recognized by the <kbest> highest weighted paths, or all of the paths with placeholder lines if there are not sufficient paths. This command, too, can be strung together with other options. Note in the following example that a new random weighting is chosen before the trees are generated, and that there are only six paths in the grammar.

```
tiburon --randomize -n -k 10 candy.rtg
This is Tiburon, version 1.0
Warning: returning fewer trees than requested
S(Stacy likes candy) # 0.235885
S(Stacy likes candy) # 0.228522
S(Stacy hates candy) # 0.151262
S(John likes candy) # 0.147251
S(John likes candy) # 0.142655
S(John hates candy) # 0.094425
0
0
0
0
```

The -g <grand> option, where <grand> is an integer, randomly follows <grand> paths and returns the strings or trees recognized by them. We show an example of this on the wcfg ran.cfg.

```
./tiburon -g 5 ran.cfg
This is Tiburon, version 1.0
the clown with the boy ran home # 0.001435
the boy by the monkey ran home # 0.000191
the clown by the clown ran home # 0.002500
the clown ran home # 0.011957
```

```
the monkey with the boy by the clown with the clown ran  # 0.000043
```

The -y flag prints the yield of trees as a string and is combined with -k or -g options. It only has any effect with wrtgs.

```
tiburon -yk 5 three.rtg
This is Tiburon, version 1.0
C C # 0.202500
C # 0.056250
C C C # 0.011391
C C C # 0.011391
C C C # 0.011391
```

We can convert wrtgs to wcfgs by simply replacing production right side trees with yield strings. We can also convert wcfgs to wrtgs (provided they do not have ϵ productions) by introducing symbols, typically repurposing nonterminal names for that purpose. The -x flag provides this functionality—in the first example that follows the ϵ production from ran.cfg was replaced with the production do -> away # .9 to demonstrate the feature, forming the file ran.noeps.cfg:

```
tiburon -x RTG ran.noeps.cfg
This is Tiburon, version 1.0
q_q
q_q -> q(q_np q_vp) # 1.000000
q_dt -> dt(the) # 1.000000
```

```
q_pp -> pp(q_prep q_np) # 1.000000

q_vb -> vb(ran) # 1.000000

q_vp -> vp(q_vb q_do) # 1.000000

q_np -> np(q_dt q_nn) # 0.500000

q_np -> np(q_dt q_nn q_pp) # 0.500000

q_nn -> nn(boy) # 0.400000

q_nn -> nn(monkey) # 0.100000

q_nn -> nn(clown) # 0.500000

q_prep -> prep(with) # 0.600000

q_prep -> prep(by) # 0.400000

q_do -> do(away) # 0.900000

q_do -> do(home) # 0.100000
```

```
tiburon -x CFG even.rtg

This is Tiburon, version 1.0

qe

qe -> qe qo # 0.100000

qe -> qo qe # 0.800000

qe -> qo # 0.100000

qo -> qo qo # 0.600000

qo -> qe qe # 0.200000

qo -> qe # 0.100000
```

```
qo -> C # 0.100000
```

We can also convert grammars to identity transducers, but will cover that in the next section.

If a sequence of wrtgs is input as arguments to Tiburon, they will be intersected, and any other flags will be invoked on the intersected grammar[4]. For example, we can combine the intersection of `three.rtg` and `even.rtg` from Figures 6.1a and 6.1c, respectively, with the `-c` flag, which displays information about the input:

```
tiburon -c three.rtg even.rtg
This is Tiburon, version 1.0
RTG info for input rtg three.rtg:
3 states
13 rules
1 unique terminal symbols
infinite derivations
RTG info for input rtg even.rtg:
2 states
7 rules
1 unique terminal symbols
infinite derivations
RTG info for intersected RTG:
6 states
```

[4] As it is undecidable whether the intersection of two context-free languages is empty, we do not attempt to intersect wcfgs.

```
43 rules
1 unique terminal symbols
infinite derivations
```

As described in Chapter 3, grammars produced by automated systems such as those used to perform machine translation [47] or parsing [10] frequently contain multiple derivations for the same item with different weight. This is due to the systems' representation of their result space in terms of weighted partial results of various sizes that may be assembled in multiple ways. This property is undesirable if we wish to know the total probability of a particular item in a language. It is also frequently undesirable to have repeated results in a *k*-best list. The `-d` operation invokes May and Knight's weighted determinization algorithm for wrtgs [95], which is applicable to wcfgs too. Note that `candy.rtg` is nondeterministic; this can be seen, since it has multiple paths that recognize the same tree. Let's generate some weights for this rtg again, and normalize them, but save off the file to use again, using the standard Unix `tee` command:

```
tiburon --randomize -n  candy.rtg | tee candy.wrtg
This is Tiburon, version 1.0
q
q -> S(subj vb obj) # 0.447814
q -> S(subj likes obj) # 0.552186
obj -> candy # 1.000000
vb -> likes # 0.779565
vb -> hates # 0.220435
```

```
subj -> John # 0.663922

subj -> Stacy # 0.336078
```

If we enumerate the paths, we will see the same tree is recognized more than once:

```
tiburon -k 6  candy.wrtg

This is Tiburon, version 1.0

S(John likes candy) # 0.366608

S(John likes candy) # 0.231775

S(Stacy likes candy) # 0.185578

S(Stacy likes candy) # 0.117325

S(John hates candy) # 0.065538

S(Stacy hates candy) # 0.033176
```

The -d operation[5] has the effect of combining duplicate paths:

```
tiburon -d 1 -k 6  candy.wrtg

This is Tiburon, version 1.0

Warning: returning fewer trees than requested

S(John likes candy) # 0.598384

S(Stacy likes candy) # 0.302902

S(John hates candy) # 0.065538

S(Stacy hates candy) # 0.033176

0

0
```

[5] which is invoked with a timeout flag, since determinization can potentially be an exponential algorithm, even on wrtgs with finite language

As a side note, in order to obtain the correct result, Tiburon may have to produce a wrtg that is not probabilistic! That is in fact what happens here:

```
tiburon -d 1  candy.wrtg
This is Tiburon, version 1.0
q5
q5 -> S(q2 q1 q4) # 0.506464
q5 -> S(q2 q3 q4) # 0.447814
q3 -> hates # 0.220435
q1 -> likes # 1.779565
q4 -> candy # 1.000000
q2 -> Stacy # 0.336078
q2 -> John # 0.663922
```

In real systems using large grammars to represent complex tree languages, memory and cpu time are very real issues. Even as computers increase in power, the added complexity of tree automata forces practitioners to combat computationally intensive processes. One way of avoiding long running times is to prune weighted automata before operating on them. One technique for pruning finite-state (string) automata is to use the forward-backward algorithm to calculate the highest-scoring path each arc in the automaton is involved in, and then prune the arcs that are only in relatively low-scoring paths [122].

We apply this technique for tree automata by using an adaptation [54] of the inside-outside algorithm [85]. The `-p` option with argument x removes productions from a

tree grammar that are involved in paths x times or more worse than the best path. The `-c` option provides an overview of a grammar, and we can use this to demonstrate the effects of pruning. The file `c1s4.determ.rtg` (not shown here) represents a language of possible translations of a particular Chinese sentence. We inspect the grammar as follows:

```
./tiburon -m tropical -c c1s4.determ.rtg
Check info:
        113 states
        168 rules
        28 unique terminal symbols
        2340 derivations
```

Note that the `-m tropical` flag is used because this grammar is weighted in the tropical semiring. We prune the grammar and then inspect it as follows:

```
java -jar tiburon.jar -m tropical -p 8 -c c1s4.determ.rtg
Check info:
        111 states
        158 rules
        28 unique terminal symbols
        780 derivations
```

Since we are in the tropical semiring, this command means "Prune all productions that are involved in derivations scoring worse than the best derivation plus 8". This roughly corresponds to derivations with probability 2980 times worse than the best derivation.

Note that the pruned grammar has fewer than half the derivations of the unpruned grammar. A quick check of the top derivations after the pruning (using `-k`) shows that the pruned and unpruned grammars do not differ in their sorted derivation lists until the 455th-highest derivation.

Tiburon contains an implementation of EM training as described in [56] that is applicable for training wrtgs and wcfgs. In Figure 6.1f a portion of `train.rtg` was shown, but the entire wrtg is quite large:

```
tiburon -c weighted_trainable.rtg
This is Tiburon, version 1.0
File is large (>10,000 rules) so time to read in will be reported below
Read 10000 rules: 884 ms
Done reading large file
RTG info for input rtg weighted_trainable.rtg:
671 states
12136 rules
45 unique terminal symbols
infinite derivations
```

We invoke training using the `-t` flag and the number of desired iterations. Note that the "ties" described above could be used here to ensure that two productions in a wrtg with the same tie id are treated as the same parameter, for purposes of counting, and thus their weights are always the same[6]. This particular case does not, however, have

[6]In the M-step, the weight of a single parameter with multiple rules is the sum of each rule's count divided by the sum of each rule's normalization group count. To ensure a probabilistic grammar, this

ties. We provide a corpus of 100 trees, `corp.100`, a portion of which was shown above, and train for 10 iterations:

```
tiburon -t 10 corp.100 train.rtg > train.t10.rtg
This is Tiburon, version 1.0
File is large (>10,000 rules) so time to read in will be reported below
Read 10000 rules: 775 ms
Done reading large file
Cross entropy with normalized initial weights is 1.254; corpus prob
   is e^-3672.832
Cross entropy after 1 iterations is 0.979; corpus prob is e^-2868.953
Cross entropy after 2 iterations is 0.951; corpus prob is e^-2786.804
Cross entropy after 3 iterations is 0.939; corpus prob is e^-2750.741
Cross entropy after 4 iterations is 0.933; corpus prob is e^-2732.815
Cross entropy after 5 iterations is 0.929; corpus prob is e^-2723.522
Cross entropy after 6 iterations is 0.928; corpus prob is e^-2718.316
Cross entropy after 7 iterations is 0.927; corpus prob is e^-2715.071
Cross entropy after 8 iterations is 0.926; corpus prob is e^-2712.862
Cross entropy after 9 iterations is 0.925; corpus prob is e^-2711.239
```

Since the input rtg has weights attached, these are used as initial parameter values. This particular wrtg generates a node in a tree based on context of either the parent node

weight is then "removed" from the available weight for the remaining members of each normalization group.

or the parent and grandparent nodes. Chain productions are used to choose between the amount of context desired. Two such productions from `train.rtg` are:

```
qSBAR_NP+ -> qSBAR_NP # .2
qSBAR_NP+ -> qSBAR # .8
```

So initially there is bias toward forgetting grandparent information when the context is (SBAR, NP). The result of training from `train.t10.rtg` is:

```
qSBAR_NP+ -> qSBAR_NP # 0.936235
qSBAR_NP+ -> qSBAR # 0.063765
```

The data causes EM to reverse this initial bias.

6.4 Transducers

In this section we describe file formats and operations on transducers, the wxtts that transform trees to trees, and the wxtsts that transform trees to strings.

6.4.1 File formats

Just as for the grammar case, both kinds of transducer have a similar formal structure: $(Q, \Sigma, \Delta, R, n_0)$, though Δ is ranked for wxtts and a simple terminal alphabet for wxtsts. Consequently, the grammar files, wxtt and wxtst, both have a similar overall format, which is remarkably similar to the grammar format:

```
<q0>
<rle>+
```

where <q0> is the start state and each <rle> is a member of R. Nonterminals can in general look like states, with the exception that : should not be used in their names. The format of a rule, <rle> is:

```
<state>"."<lhs> "->" <rhs> ["#" <wgt>] ["@" <tie>]
```

where <state> is a state, and <wgt> and <tie> are as for grammars. <lhs> has the following format:

```
<vbl>":" | <sig-sym> | <sig-sym>"("<lhs>+")"
```

where <sig-sym> is a member of Σ and <vbl> is a variable. Variables have the same form as alphabet symbols, nonterminals, and states, though by convention we generally name them like x4. A <lhs> is invalid if the same <vbl> appears more than once.

If the file is a wxtt, then <rhs> has the following format:

```
<state>"."<vbl> | <del-sym> | <del-sym>"("<rhs>+")"
```

If the file is a wxtst, then <rhs> has the following format:

```
"*e*" | (<state>"."<vbl>|<del-sym>)+
```

A <rhs> is invalid if it contains a variable not present in <lhs>. As for grammars, the various alphabets are defined implicitly. Aside from the regular use of comments, type can be enforced as follows:

```
% TYPE XR
% TYPE XRS
```

```
q
q.A(Z(x0:) x1:) -> B(C(q.x0 r.x0) q.x1) # 0.3
q.E(x0:) -> F # 0.5
r.E -> G # 0.7
```

(a) `wxtt.trans` is a wxT transducer.

```
s
s.B(x0: x1:) -> D(t.x1 s.x0)
s.C(x0: x1:) -> H(s.x0 s.x1) # 0.6
s.C(x0: x1:) -> H(s.x1 s.x0) # 0.4
t.B(x0: x1:) -> D(s.x0 s.x1)
s.F -> L
t.F -> I
s.G -> J # 0.7
s.G -> K # 0.3
```

(b) `wlnt.trans` is a wLNT transducer.

```
rJJ
rJJ.JJ(x0: x1:) -> JJ(rJJ.x0 rTO.x1) # 0.250000
rJJ.JJ(x0: x1:) -> JJ(rTO.x1 rJJ.x0) # 0.750000
rJJ.JJ(x0:) -> JJ(t.x0) # 1.000000
t."abhorrent" -> "abhorrent" # 1.000000
rTO.TO(x0: x1:) -> TO(rPRP.x1 rTO.x0) # 0.333333
rTO.TO(x0: x1:) -> TO(rTO.x0 rPRP.x1) # 0.666667
rTO.TO(x0:) -> TO(t.x0) # 1.000000
rPRP.PRP(x0:) -> PRP(t.x0) # 1.000000
t."them" -> "them" # 1.000000
t."to" -> "to" # 1.000000
```

(c) Rotation transducer fragment.

```
iJJ
iJJ.JJ(x0: x1:) -> JJ(iJJ.x0 iJJ.x1) # 0.928571 @ 108
iJJ.JJ(x0: x1:) -> JJ(iJJ.x0 iJJ.x1 INS) # 0.071429 @ 107
iJJ.JJ(x0:) -> JJ(t.x0) # 0.928571 @ 108
iJJ.TO(x0: x1:) -> TO(iTO.x0 iTO.x1) # 1.000000 @ 159
iTO.TO(x0:) -> TO(t.x0) # 0.800000 @ 165
iTO.TO(x0:) -> TO(s.x0 INS) # 0.111842 @ 164
iTO.TO(x0:) -> TO(INS s.x0) # 0.088158 @ 163
iTO.PRP(x0:) -> PRP(t.x0) # 1.000000 @ 150
t."abhorrent" -> "abhorrent" # 1.000000
t."to" -> "to" # 1.000000
t."them" -> "them" # 1.000000
s."to" -> nn-to # 1.000000
```

(d) Insertion transducer fragment.

Figure 6.2: Example wxtt and wxtst files used to demonstrate Tiburon's capabilities.

We now present several wxtt and wxtst examples.

As with grammars, Tiburon tries to automatically detect the type of file input, but the following lines, included as the first line of the transducer file, remove ambiguity:

```
% TYPE XR
% TYPE XRS
```

The format for batch files for training transducers is

```
(<count>
<in-item>
<out-item>)+
|
(<in-item>
<out-item>)+
```

The format for <in-item> is:

```
<sig-sym> | <sig-sym>"("<in-item>+")"
```

A batch file for training wxtts is a file of tree-tree pairs. The format for <out-item> in this case is:

```
<del-sym> | <del-sym>"("<out-item>+")"
```

A batch file for training wxtsts is a file of tree-string pairs. The format for <out-item> in this case is:

```
<del-sym>+
```

Here is a sample corpus of English tree-Japanese string pairs:

```
VB(NN("hypocrisy") VB("is") JJ(JJ("abhorrent") TO(TO("to") PRP("them"))))
"彼ら" "は" "偽善" "が" "大嫌い" "だ"
VB(PRP("he") VB("has") NN(JJ("unusual") NN("ability")) IN(IN("in") NN("english")))
"彼" "は" "英語" "に" "ずばぬけ" "た" "才能" "を" "持っ" "て" "いる"
```

6.4.2 Commands using transducer files

As with grammars, you can simply provide a transducer file to Tiburon as input and it will return its contents to you.

```
tiburon comp2.rln

This is Tiburon, version 1.0

s

s.B(x0: x1:) -> D(t.x1 s.x0) # 1.000000

s.C(x0: x1:) -> H(s.x0 s.x1) # 0.600000

s.C(x0: x1:) -> H(s.x1 s.x0) # 0.400000

s.F -> L # 0.800000

s.G -> J # 0.700000

s.G -> K # 0.300000

t.B(x0: x1:) -> D(s.x0 s.x1) # 1.000000

t.F -> I # 0.200000
```

The -c flag works for transducers, too.

```
tiburon -c comp2.rln

This is Tiburon, version 1.0

Transducer info for input tree transducer comp2.rln:
```

```
2 states

8 rules
```

Analogous to the intersection of wrtgs, providing multiple transducers to Tiburon causes it to compose them. However, unlike the wrtg case, there are strict constraints on the classes of transducer that can be composed.

```
tiburon  wxtt.trans wlnt.trans

This is Tiburon, version 1.0

q0

q0.E(x0:) -> L # 0.500000

q0.A(Z(x0:) x1:) -> D(q13.x1 H(q0.x0 q14.x0)) # 0.180000

q0.A(Z(x0:) x1:) -> D(q13.x1 H(q14.x0 q0.x0)) # 0.120000

q13.E(x0:) -> I # 0.500000

q13.A(Z(x0:) x1:) -> D(H(q0.x0 q14.x0) q0.x1) # 0.180000

q13.A(Z(x0:) x1:) -> D(H(q14.x0 q0.x0) q0.x1) # 0.120000

q14.E -> J # 0.490000

q14.E -> K # 0.210000
```

Providing a wrtg and a transducer (or sequence of transducers) as arguments causes Tiburon to do application[7]. As before, subsequent operations (such as `-k`) are invoked on the resulting application wrtg or wcfg. In the following example we pass in a tree directly from the command line; the `-` represents where standard input is placed in the argument sequence.

[7]The currently released version only provides bucket brigade application

```
echo 'JJ(JJ("abhorrent") TO(TO("to") PRP("them")))' |
     ./tiburon  -k 5 - rot ins
This is Tiburon, version 1.0
JJ(TO(TO("to") PRP("them")) JJ("abhorrent")) # 0.344898
JJ(TO(PRP("them") TO("to")) JJ("abhorrent")) # 0.172449
JJ(JJ("abhorrent") TO(TO("to") PRP("them"))) # 0.114966
JJ(JJ("abhorrent") TO(PRP("them") TO("to"))) # 0.057483
JJ(TO(TO(nn-to INS) PRP("them")) JJ("abhorrent")) # 0.048218
```

As mentioned before, we can convert grammars to transducers.

```
tiburon -x XRS even.rtg | tee even.xrs
This is Tiburon, version 1.0
qe
qe.A(x0: x1:) -> qe.x0 qo.x1 # 0.100000
qe.A(x0: x1:) -> qo.x0 qe.x1 # 0.800000
qe.B(x0:) -> qo.x0 # 0.100000
qo.A(x0: x1:) -> qo.x0 qo.x1 # 0.600000
qo.A(x0: x1:) -> qe.x0 qe.x1 # 0.200000
qo.B(x0:) -> qe.x0 # 0.100000
qo.C -> C # 0.100000
```

This is a good way to build a parser! We can now pass a string in to the right of this transducer and form a wrtg that recognizes all (infinitely many) parses of the string with an even number of total nodes!

```
echo "C C C" | ./tiburon -k 5 a -

This is Tiburon, version 1.0

A(C A(C B(C))) # 0.000064

A(A(C C) B(C)) # 0.000048

A(C B(A(C C))) # 0.000048

B(A(A(C C) C)) # 0.000036

B(A(C A(C C))) # 0.000036
```

Finally, we can also train transducers. This transducer is an English-to-Japanese xNTs and thus we also need to set the character class with `-e euc-jp`. Here is a part of the transducer to be trained:

```
qTOP
r.DT(x0:"the") -> t.x0
r.PRP(x0:"he") -> t.x0
r.PRP(x0:"i") -> t.x0
...

r.VB(x0:PRP x1:VB x2:JJ) -> qVB.x2 qVB.x1 qVB.x0
r.PRP(x0:PRP x1:NN) -> qPRP.x1 qPRP.x0 @ 1
r.NN(x0:PRP x1:NN) -> qNN.x1 qNN.x0 @ 1
r.VB(x0:PRP x1:VB x2:NN x3:IN) -> qVB.x3 qVB.x2 qVB.x0 qVB.x1
...
qTOP.x0:VB -> i.x0 r.x0
qTOP.x0:VB -> r.x0 i.x0
qTOP.x0:VB -> r.x0
qVB.x0:RB -> i.x0 r.x0
qVB.x0:RB -> r.x0 i.x0
qVB.x0:RB -> r.x0
...
i.x0: -> "住ん"
t."school" -> "住ん"
t."five" -> "住ん"
t."english" -> "持っ"
t."cannot" -> "の"
t."abominate" -> "だ"
i.x0: -> "いる"
...
t."rises" -> *e*
t."above" -> *e*
t."cannot" -> *e*
```

Training looks like this:

```
tiburon -e euc-jp -t 5 corpus transducer > a
This is Tiburon, version 1.0
Cross entropy with normalized initial weights is 2.196;
   corpus prob is e^-474.308
Cross entropy after 1 iterations is 1.849; corpus prob is e^-399.584
Cross entropy after 2 iterations is 1.712; corpus prob is e^-369.819
Cross entropy after 3 iterations is 1.544; corpus prob is e^-333.421
Cross entropy after 4 iterations is 1.416; corpus prob is e^-305.830
```

6.5 Performance comparison

Tiburon is primarily intended for manipulation, combination, and inference of tree automata and transducers, but as these formalisms generalize string transducers and automata (see Figure 2.19), we can use it as a wfst toolkit, too. We may thus compare Tiburon's performance with other wfst toolkits. In this section we conduct performance and scalability tasks on wfsts and wfsas on three widely available wfst toolkits: Carmel version 6.2 (released May 4, 2010) [53], OpenFst version 1.1 (released June 17, 2009) [4], and FSM version 4.0 (released 2003) [102]. We repeat the same tests on wtt and wrtg equivalents in Tiburon version 1.0 (released concurrently with this thesis in August, 2010). The wfst toolkits were written in C or C++ and have been extensively used and tested over a number of years. Tiburon, by contrast, was written in Java and has received less testing and development time. The following sections demonstrate that Tiburon is

`t`
`(q (r A B .2))`
`(s (t *e* *e* .4))`
(a) fst rules and final state for Carmel.
	`q r A B 1.609`
	`s t *e* *e* 0.916`
	`t`
	(b) fst rules and final state for FSM and OpenFst.
		`q.A(x0:) -> B(r.x0) # .2`
		`s.x0: -> t.x0 # .4`
		`t.TIBEND -> TIBEND # 1`
		(c) Tree transducer rules and simulation of final state for Tiburon.

`A B`
(d) String representation in Carmel (`-i` switch converts to identity fst).
	`0 1 A A 0`
	`1 2 B B 0`
	`2`
	(e) String representation as identity fst in FSM and OpenFst.
		`A(B(TIBEND))`
		(f) String representation as monadic tree in Tiburon.

Figure 6.3: Comparison of representations in Carmel, FSM/OpenFst, and Tiburon.

generally slower and less scalable than its competitors on tasks designed for wfst toolkits. We ran these on a custom-built machine with four AMD Opteron 850 processors and 32g memory running Fedora Core 9. Times were calculated by summing the "user" and "sys" information reported by the `time` command and averaging over three repeated runs. All timing results are shown in Table 6.2. For Carmel and Tiburon we simply report the averaged time to perform the described operation. FSM and OpenFst operate by first converting text representations of transducers into a machine-readable binary format, then performing necessary operations, then finally converting back to text format. We thus separate out conversion from transducer manipulation operations for these toolkits in Table 6.2.

6.5.1 Transliteration cascades

We tested the toolkits' performance on transliterating Japanese katakana of English names through a cascade of transducers, similar to that described by Knight and Graehl

Order	Description	States	Rules
1	generates English words	1	50,001
2	pronounces English sounds	150,553	302,349
3	converts English sounds to Japanese sounds	99	283
4	slightly disprefers certain Japanese phonemes	5	94
5	combines doubled Japanese vowels	6	53
6	converts Japanese sounds to katakana	46	294

Table 6.1: Generative order, description, and statistics for the cascade of English-to-katakana transducers used in performance tests in Section 6.5.

[75].[8] We built equivalent versions of these transducers as well as a representation of katakana string glosses in formats suitable for the four toolkits. Figure 6.3 shows the difference between the various formats. Note that for Tiburon, a monadic tree replaces the katakana string. Also note that we represent weights in FSM and OpenFst in negative log space. We used these transducers to conduct performance experiments in simple reading, inference through a cascade, and k-best path search of an automaton.

6.5.1.1 Reading a transducer

The most basic task a toolkit can do is read in a file representing a transducer or automaton. We compared the systems' ability to read in and generate basic information about the large pronunciation transducer listed as line 2 in Table 6.1. OpenFst is about twice as slow as FSM and Carmel at reading in a transducer, while Tiburon is about two orders of magnitude worse than FSM and Carmel.

[8] We do not include a transducer modeling misspellings caused by OCR, as Knight and Graehl [75] do, and we include two additional transducers for technical reasons—lines 4 and 5 in Table 6.1.

6.5.1.2 Inference through a cascade

As described by Knight and Graehl [75], we can use the cascade of transducers that produce katakana from English words to decode katakana by passing a candidate string backward through the cascade, thereby obtaining a representation of all possible inputs as a (string or tree) automaton. We compared the systems' ability to perform this operation with the katakana gloss *a n ji ra ho re su te ru na i to*. Carmel is by far the fastest at this task—OpenFst is about 4.5 times slower, FSM is ten times slower than OpenFst, and Tiburon is another six times slower than FSM.

6.5.1.3 K-best

To complete the inference task it is important to obtain a k-best list from the result automaton produced in Section 6.5.1.2. Carmel can produce such a list, as does Tiburon (though in this case it is a list of monadic trees). FSM and OpenFst do not directly produce such lists but rather produce wfsas or identity wfsts that contain only the k-best paths; a post-process is needed to agglomerate labels and weights. Figure 6.4 shows the 2-best lists or their equivalent representations produced by each of the toolkits.

We compared the systems' abilities to generate a 20, 20,000, and 2,000,000-best list from the result automata they previously produced. For FSM and OpenFst we only considered the generation of the representative automata. FSM and Carmel took about the same amount of time for the 20-best list; OpenFst was an order of magnitude slower, and Tiburon, which as previously noted is very inefficient at reading in structures, was considerably worse. Tiburon was only twice as slow at obtaining the 20,000-best list as it was the 20-best list; this is because the overhead incurred for reading in the large

```
ANGELA FORRESTAL KNIGHT 2.60279825597665e-20
ANGELA FORRESTER KNIGHT 6.00711401244296e-21
```

(a) Carmel output.

```
ANGELA(FORRESTAL(KNIGHT(END))) # 2.602803E-20
ANGELA(FORRESTER(KNIGHT(END))) # 6.007097E-21
```

(b) Tiburon output.

```
0 1 *e*
0 28 *e*
1 2 *e*
2 3 *e* 0.0794004127
...
7 8 *e* 0.637409806
8 9 "ANGELA" 10.6754665
9 10 *e*
...
25 26 "KNIGHT" 8.59761715
26 27 *e*
27
```

(c) Partial FSM (OpenFst) output. The complete output has 53 arcs and two final states.

Figure 6.4: K-best output for various toolkits on the transliteration task described in Section 6.5.1.3. Carmel produces strings, and Tiburon produces monadic trees with a special leaf terminal. FSM and OpenFst produce wfsts or wfsas representing the k-best list and a post-process must agglomerate symbols and weights.

automaton dominates the 20-best operation. OpenFst, which also is comparatively slow at reading, was one order of magnitude slower than its 20-best operation at generating the 20,000-best list. Unlike in the 20-best case, FSM was much slower than Carmel at 20,000-best. This may be due to FSM's 32-bit architecture—the other three systems are all 64-bit. The 32-bit limitation was certainly the reason for FSM's inability to generate a 2,000,000-best list. It is unable to access more than 4g memory, while the other systems were able to take advantage of higher memory limits. Tiburon required explicit expansion of the Java heap in order to complete its task. It should be noted, though, that Carmel was able to obtain a 2,000,000-best list even when its 32-bit variant was run. This may be due to a choice of algorithm—Carmel uses the k-best algorithm of Eppstein [40], while OpenFst and FSM use the algorithm of Mohri and Riley [104].

6.5.2 Unsupervised part-of-speech tagging

As described by Merialdo [98], the EM algorithm can be used to train a part-of-speech tagging system given only a corpus of word sequences and a dictionary of possible tags for each word in a given vocabulary. The task is unsupervised, as no explicit tag sequences or observed tag-word pairs are given. An HMM is formed, where the states represent an n-gram tag language model, and an emission for each word from each state represents a parameter in a word-given-tag channel model. We follow Ravi and Knight [114] and use a bigram model rather than Merialdo's trigram model, as the former gives better performance and is easier to construct. As is standard practice, the language model is initially fully connected (all n-gram transitions are possible) and the channel model is bootstrapped by the tag dictionary. We instantiate the HMM as a single unweighted wfst and its tree transducer counterpart. A schematic of a sample HMM wfst for a language of two tags and two words is shown in Figure 6.5.

6.5.2.1 Training

We compared Tiburon with Carmel on EM training of the HMM described in Section 6.5.2.[9] The instantiated wfst/wtt has 92 states and 59,503 arcs. We trained both systems for 10 iterations of EM using a corpus of 300 sentences of 20 words or fewer as training. This is a reduction from the full Merialdo corpus of 1005 sentences of various lengths ranging from 1 to 108 words, which overwhelms Tiburon's memory, even when allowed the full 32g available on the test machine. Tiburon was slightly more than two orders of magnitude slower than Carmel. To demonstrate Carmel's ability to scale, we ran it

[9] FSM and OpenFst do not provide training functionality.

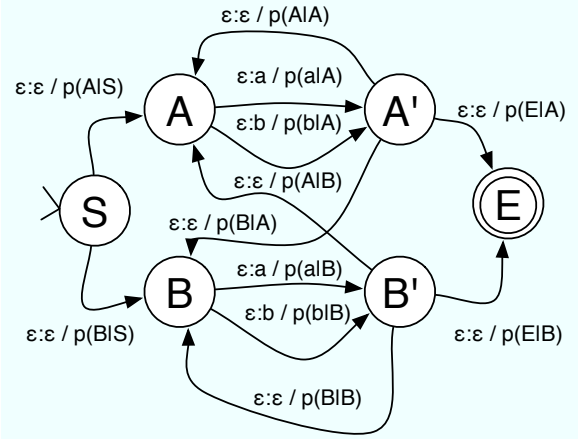


Figure 6.5: An example of the HMM trained in the Merialdo [98] unsupervised part-of-speech tagging task described in Section 6.5.2, instantiated as a wfst. Arcs either represent parameters from the bigram tag language model (e.g., the arc from A′ to B, representing the probability of generating tag B after tag A) or from the tag-word channel model (e.g., the topmost arc from A to A′, representing the probability of generating word a given tag A). The labels on the arcs make this wfst suitable for training on a corpus of (ε, word sequence) pairs to set language model and channel model probabilities such that the probability of the training corpus is maximized.

on the whole training data using the aforementioned model, and also created a more complicated model by splitting the tag space, in the spirit of Petrov and Klein [111], effectively doubling the state space and adding many more hidden variables for EM to model. This model as instantiated has 182 states and 123,057 arcs. Tiburon was unable to train this model, even on the 300-sentence reduced corpus, but Carmel was able to train this model on the entire Merialdo corpus easily.

6.5.2.2 Determinization

In order to use the trained transducers for tagging, the arcs representing the language model are modified such that their input symbol is changed from ε to the tag corresponding with the destination state. In the example of Figure 6.5, for instance, the arc label from A′ to B would be changed from ε:ε to b:ε. Backward application of the candidate

string can then be performed to form a result graph, and the most likely tag sequence calculated. In the original bigram formulation the result graph is deterministic and there is exactly one path for every distinct tag sequence. However, in the state split model discussed above, this is not the case, and it is important to determinize the result graph to ensure the most likely path and most likely sequence coincide. We used Carmel to build the state-split model above and to obtain a nondeterministic result graph for a 303-word sequence. The graph is highly nondeterministic—it has 2,624 states and 6,921 arcs and represents 1.8×10^{81} distinct taggings in 4.1×10^{157} paths. Carmel does not have a determinization function, but FSM, OpenFst, and Tiburon do, so we compared the ability of the three systems to determinize this result graph. FSM and OpenFst were quite fast at determinizing, even though the operation is potentially exponential in the input size, while Tiburon was again two orders of magnitude slower.

6.5.3 Discussion

We have seen that Tiburon runs consistently around two orders of magnitude slower than competing wfst toolkits on common tasks. Additionally, Tiburon does not scale with transducer and data size as well as the other systems do. Partial reasons for this may be unoptimized implementations, bugs, and the inherent advantage of compiled code. However, Tiburon has a key liability in its more general nature. As an example, consider the representation of a wfst arc in any of the wfst toolkits with a wxtt rule in Tiburon. A wfst arc may be represented by four integers and a float, denoting the input and output states and symbols and the arc weight, respectively. More complicated powers such as parameter tying and locking that are available in Carmel require an additional integer

Experiment	FSM		OpenFst		Carmel	Tiburon
	conv.	op.	conv.	op.		
reading	1.2s	.2s	2.6s	.4s	1.2s	87s
inference	2.0s	51.8s	5.4s	.2s	1.2s	326s
20-best	.09s	.01s	.4s	.01s	.02s	5.5s
20,000-best	1.6s	1.7s	1s	1.5s	.2s	10s
2,000,000-best	OOM		49s	112s	22s	1369s
train (reduced)	N/A		N/A		1.2s	159s
train (full)	N/A		N/A		2.9s	OOM
train (split)	N/A		N/A		9.0s	OOM
determinize	.05	.03	.08s	.06	N/A	232s

Table 6.2: Timing results for experiments using various operations across several transducer toolkits, demonstrating the relatively poor performance of Tiburon as compared with extant string transducer toolkits. The reading experiment is discussed in Section 6.5.1.1, inference in Section 6.5.1.2, the three *k*-best experiments in Section 6.5.1.3, the three training experiments in Section 6.5.2.1, and determinization in Section 6.5.2.2. For FSM and OpenFst, timing statistics are broken into time to convert between binary and text formats ("conv.") and time to perform the specified operation ("op."). N/A = this toolkit does not support this operation. OOM = the test computer ran out of memory before completing this experiment.

and boolean, but the memory profile of an arc is quite slim. A wxtt rule, on the other hand, is represented by an integer for the input state, a float for the weight, two trees for the input and output patterns, and a map linking the variables of the two sides together. The trees and the map are instantiated as objects, and not fixed-width fields, giving a single transducer rule a considerable minimum memory footprint. Additionally, overhead for reading, storing, and manipulating these more general structures is increased due to their variable size. A useful improvement to Tiburon would be a reimplementation of the fundamental data structures such that their size is fixed. This would allow more assumptions to be made about the objects and consequently more efficient processes, particularly in time-consuming I/O.

6.6 External libraries

We are grateful to the following external sources for noncommercial use of their Java libraries in Tiburon: Martian Software, for the JSAP command line parsing libraries, Stanford University's NLP group, for its implementation of heaps, and to the makers of the Gnu Trove, which provided several object container classes used in earlier versions of the software.

6.7 Conclusion

We have described Tiburon, a general weighted tree automata toolkit, and described some of its functions and their use in constructing natural language applications. Tiburon can be downloaded at `http://www.isi.edu/licensed-sw/tiburon/`.

Chapter 7

Concluding Thoughts and Future Work

7.1 Concluding thoughts

To recap, this thesis provided the following contributions:

- Algorithms for key operations of a weighted tree transducer toolkit, some of which previously only existed as proofs of concept, and some of which were novel.
- Empirical experiments, putting weighted tree transducers and automata to work to obtain machine translation and parsing improvements.
- Tiburon, a tree transducer toolkit that allows others to use these key operations and formalisms in their own systems.

The overall purpose of this thesis, some might say its "thesis", has been that weighted tree transducers and automata are useful formalisms for NLP models and that the development of practical algorithms and tangible software with implementations of those algorithms make these formal structures useful tools for actual NLP systems. As is usual, though, it is the journey to this thesis' end that is perhaps more important than the

destination. While I have provided a toolkit to enable development of syntax-infused systems as cascades of formal models, and used it to build some systems, the algorithm development component of this work seems to me to be the most interesting contribution. Heretofore, tree transducers generally existed as abstract entities. Many papers reasoned about them but little actual work was done using them, and this is chiefly because there was no real concrete way to actually use them. With Tiburon, we really can use these structures, and it is empirical work down the road that will actually test their long-term utility. But this is not a software engineering thesis, and Tiburon is not some exemplar of software design. No, the main contribution of this work is that before this thesis began Tiburon could not be constructed, as there were no clear-cut algorithms for nearly any desired operation. But now we not only have Tiburon, but the keys to extend it, refine it, even rebuild it if needed.

7.2 Future work

There are several different directions of useful future work that would enhance the results of this thesis. They broadly fall into the categories of *algorithms*, *models*, and *engineering*.

7.2.1 Algorithms

For most real-world systems, exact inference is an acceptable trade-off for speed. The key operations of Tiburon could be replaced with principled approximate versions that guarantee performance rates for a known amount of error. Approximate algorithms that admit a wider class of automaton or transducer at the expense of "limited incorrectness"

would also be useful. The following comprises a wish list of approximate variants of the algorithms presented in this thesis:

- A k-best algorithm for wrtgs and wcfgs that runs linear in k but may skip some paths.
- A polynomial-time determinization algorithm for wrtgs and wcfgs that returns an output automaton that does not recognize some low-weighted trees that were recognized by the input.
- A domain projection algorithm for wxNT that produces a wrtg recognizing a tree series with equivalent support to the true domain projection but only enforces that the weights of the trees be in the same relative order as they are in the true projection.
- A composition algorithm for wxLNT that produces a wxLNT that over- or under-generates the resulting weighted tree transformation.
- An on-the-fly application algorithm for a cascade, ending in a wxLNTs and a string, that has memory and speed guarantees at the expense of a known loss of the application wrtg's tree series.

7.2.2 Models

An important next step in this line of work is a thorough examination of the limitations of the tree transducer formalism at representing syntax models we care about. For example, Collins' parsing model [25] has a complicated back-off weighting scheme that does not

seem amenable to representation in the tree transducer domain. Additionally, as shown by DeNeefe and Knight [29], real-world systems built with tree transducer models have to use very large rule sets to model slight variations in the trees that can be produced (such as sentences with or without independent clauses). More flexible formalisms, such as synchronous tree adjoining grammars [120] may end up being key, and thus relevant extensions of the algorithms presented in this thesis may have to be designed.

7.2.3 Engineering

Tiburon has also not been extensively battle tested in ways that really bring it up to the level of a production-level toolkit. This would indeed require a software engineering thesis. Such an approach could also focus on improving the runtime of the algorithms presented here—many are quite impractical for large-scale efforts. Additionally, the benefits and limitations of general models such as tree transducers are exposed by application to a wide domain. I have limited discussion in this thesis to natural language experiments and scenarios, but tree-structured data exists in biological domains and may be quite useful in the study of financial systems. Treatments of wider genres of data are sure to provide insight into what challenges toward constructing a widely-used toolkit remain, in both the formal and engineering domains.

7.3 Final words

If nothing else it is my hope that this thesis has helped to reunite NLP practitioners and formal language theorists, such that the two fields can attempt to talk to each other in a common language and recognize how they may be of mutual benefit.

References

[1] Athanasios Alexandrakis and Symeon Bozapalidis. Weighted grammars and Kleene's theorem. *Information Processing Letters*, 24(1):1–4, January 1987.

[2] Cyril Allauzen and Mehryar Mohri. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics*, 8(2):117–144, 2003.

[3] Cyril Allauzen, Mehryar Mohri, and Brian Roark. A general weighted grammar library. In *Implementation and Application of Automata: Ninth International Conference (CIAA 2004)*, volume 3317 of *Lecture Notes in Computer Science*, pages 23–34, Kingston, Ontario, Canada, July 2004. Springer-Verlag.

[4] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skui, and Mehryar Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Ninth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23, Prague, 2007.

[5] André Arnold and Max Dauchet. Morhpismes et bimorphismes d'arbres. *Theoretical Computer Science*, 20:33–93, 1982.

[6] Brenda S. Baker. Composition of top-down and bottom-up tree transductions. *Information and Control*, 41(2):186–213, May 1979.

[7] Adam Berger, Peter Brown, Stephen Della Pietra, Vincent Della Pietra, John Gillett, John Lafferty, Robert Mercer, Harry Printz, and Luboš Ureš. The Candide system for machine translation. In *Human Language Technology*, pages 157–162, Plainsboro, New Jersey, March 1994.

[8] Jean Berstel and Christophe Reutenauer. Recognizable formal power series on trees. *Theoretical Computer Science*, 18(2):115–148, 1982.

[9] Daniel Bikel. Intricacies of Collins' parsing model. *Computational Linguistics*, 30(4):479–511, 2004.

[10] Rens Bod. A computational model of language performance: data oriented parsing. In *Proceedings of the fifteenth International Conference on Computational Linguistics (COLING-92)*, volume 3, pages 855–859, Nantes, France, 1992.

[11] Rens Bod. An efficient implementation of a new DOP model. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, pages 19–26, Budapest, 2003.

[12] Tugba Bodrumlu, Kevin Knight, and Sujith Ravi. A new objective function for word alignment. In *Proceedings of the NAACL HLT Workshop on Integer Linear Programming for Natural Language Processing*, pages 28–35, Boulder, Colorado, June 2009.

[13] Björn Borchardt. *The Theory of Recognizable Tree Series*. PhD thesis, Dresden University of Technology, 2005.

[14] Björn Borchardt and Heiko Vogler. Determinization of finite state weighted tree automata. *Journal of Automata, Languages and Combinatorics*, 8(3):417–463, 2003.

[15] Peter Borovansky, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, August 2002.

[16] Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

[17] Matthias Büchse, Jonathan May, and Heiko Vogler. Determinization of weighted tree automata using factorizations. In *Pre-proceedings of the Eight International Workshop on Finite-State Methods and Natural Language Processing*, July 2009.

[18] Matthias Büchse, Jonathan May, and Heiko Vogler. Determinization of weighted tree automata using factorizations. *Journal of Automata, Languages and Combinatorics*, 2010. submitted.

[19] Francisco Casacuberta and Colin de la Higuera. Computational complexity of problems on probabilistic grammars and transducers. In Arlindo L. Oliveira, editor, *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000*, pages 15–24, Lisbon, Portugal, September 2000.

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

[21] David Chiang. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting of the ACL*, pages 263–270, Ann Arbor, Michigan, June 2005.

[22] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[23] Noam Chomsky. *Syntactic Structures*. Mouton, 1957.

[24] Alexander Clark. Memory-based learning of morphology with stochastic transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 513–520, Philadelphia, PA, July 2002.

[25] Michael Collins. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid, Spain, July 1997.

[26] Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 111–118, Barcelona, Spain, July 2004.

[27] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2007. released October 12, 2007.

[28] Arthur Dempster, Nan Laird, and Donald Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[29] Steve DeNeefe and Kevin Knight. Synchronous tree adjoining machine translation. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 727–736, Singapore, August 2009.

[30] Steve DeNeefe, Kevin Knight, and Hayward Chan. Interactively exploring a machine translation model. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 97–100, Ann Arbor, Michigan, 2005.

[31] Steve DeNeefe, Kevin Knight, Wei Wang, and Daniel Marcu. What can syntax-based MT learn from phrase-based MT? In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 755–763, Prague, Czech Republic, June 2007.

[32] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[33] John Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4(5):406–451, October 1970.

[34] Frank Drewes and Renate Klempien-Hinrichs. Treebag. In Sheng Yu and Andrei Paun, editors, *Proc. 5th Intl. Conference on Implementation and Application of Automata (CIAA 2000)*, volume 2088 of Lecture Notes in Computer Science, pages 329–330, London, Ontario, Canada, 2001.

[35] Manfred Droste and Werner Kuich. Semirings and formal power series. In *Handbook of Weighted Automata*, chapter 1, pages 3–28. Springer-Verlag, 2009.

[36] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 6(8):451–455, 1970.

[37] Abdessamad Echihabi and Daniel Marcu. A noisy-channel approach to question answering. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Sapporo, Japan, July 2003.

[38] Jason Eisner. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 1–8, Philadelphia, Pennsylvania, USA, July 2002.

[39] Joost Engelfriet. Bottom-up and top-down tree transformations – a comparison. *Mathematical Systems Theory*, 9(2):198–231, 1975.

[40] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[41] Zoltán Ésik and Werner Kuich. Formal tree series. *Journal of Automata, Languages and Combinatorics*, 8(2):219–285, 2003.

[42] Alexander Fraser and Daniel Marcu. Semi-supervised training for statistical word alignment. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 769–776, Sydney, Australia, July 2006.

[43] Zoltán Fülöp, Andreas Maletti, and Heiko Vogler. Backward and forward application of weighted extended tree transducers. Unpublished manuscript, 2010.

[44] Zoltán Fülöp and H. Vogler. Weighted tree transducers. *J. Autom. Lang. Comb.*, 9(1), 2004.

[45] Zoltán Fülöp and Heiko Vogler. Weighted tree automata and tree transducers. In *Handbook of Weighted Automata*, chapter 9, pages 313–404. Springer-Verlag, 2009.

[46] Michel Galley, Jonathan Graehl, Kevin Knight, Daniel Marcu, Steve DeNeefe, Wei Wang, and Ignacio Thayer. Scalable inference and training of context-rich syntactic models. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 961–968, Sydney, Australia, July 2006.

[47] Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. What's in a translation rule? In Daniel Marcu Susan Dumais and Salim Roukos, editors, *HLT-NAACL 2004: Main Proceedings*, pages 273–280, Boston, May 2004.

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

[49] Ferenc Gécseg and Magnus Steinby. Tree languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Beyond Words*, volume 3 of *Handbook of Formal Languages*, chapter 1, pages 1–68. Springer-Verlag, Berlin, 1997.

[50] Thomas Genet, Valérie Viet, and Triem Tong. Reachability analysis of term rewriting systems with timbuk. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706, Havana, Cuba, December 2001. Springer-Verlag.

[51] Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. Fast decoding and optimal decoding for machine translation. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, pages 228–235, Toulouse, France, July 2001.

[52] Jonathan S. Golan. *Semirings and Their Applications*. Kluwer Academic, Dordrecht, The Netherlands, 1999.

[53] Jonathan Graehl. Carmel finite-state toolkit. http://www.isi.edu/licensed-sw/carmel, 1997.

[54] Jonathan Graehl. Context-free algorithms. Unpublished handout, July 2005.

[55] Jonathan Graehl and Kevin Knight. Training tree transducers. In *HLT-NAACL 2004: Main Proceedings*, pages 105–112, Boston, Massachusetts, USA, May 2004.

[56] Jonathan Graehl, Kevin Knight, and Jonathan May. Training tree transducers. *Computational Linguistics*, 34(3):391–427, September 2008.

[57] Udo Hebisch and Hans Joachim Weinert. *Semirings—Algebraic Theory and Applications in Computer Science*. World Scientific, Singapore, 1998.

[58] Jesper G. Henriksen, Ole J. L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm. MONA: Monadic second-order logic in practice. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 58–73, Department of Computer Science, University of Aarhus, May 1995. BRICS.

[59] Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64, Vancouver, British Columbia, Canada, October 2005.

[60] Aravind K. Joshi and Phil Hopely. A parser from antiquity. *Natural Language Engineering*, 2(4):291–294, 1996.

[61] Ed Kaiser and Johan Schalkwyk. Building a robust, skipping parser within the AT&T FSM toolkit. Technical report, Center for Human Computer Communication, Oregon Graduate Institute of Science and Technology, 2001.

[62] Stephan Kanthak and Hermaan Ney. FSA: An efficient and flexible C++ toolkit for finite state automata using on-demand computation. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 510–517, Barcelona, July 2004.

[63] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

[64] Ronald M. Kaplan and Martin Kay. Phonological rules and finite-state transducers. In *Linguistic Society of America Meeting Handbook, Fifty-Sixth Annual Meeting*, 1981. Abstract.

[65] Lauri Karttunen. The replace operator. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Cambridge, Massachusetts, USA, June 1995.

[66] Lauri Karttunen. Directed replacement. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 108–115, Santa Cruz, California, USA, June 1996.

[67] Lauri Karttunen and Kenneth R. Beesley. Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA, 1992.

[68] Lauri Karttunen and Kenneth R. Beesley. A short history of two-level morphology. Presented at the ESSLLI-2001 Special Event titled "Twenty Years of Finite-State Morphology", August 2001. Helsinki, Finland.

[69] Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, 1997.

[70] Lauri Karttunen, Tamás Gaál, and André Kempe. Xerox finite-state tool. Technical report, Xerox Research Centre Europe, 1997.

[71] Lauri Karttunen, Ronald M. Kaplan, and Annie Zaenen. Two-level morphology with composition. In *Proceedings of the fifteenth International Conference on Computational Linguistics (COLING-92)*, volume 3, pages 141–148, Nantes, France, 1992.

[72] Daniel Kirsten and Ina Mäurer. On the determinization of weighted automata. *Journal of Automata, Languages and Combinatorics*, 10(2/3):287–312, 2005.

[73] Kevin Knight. Capturing practical natural language transformations. *Machine Translation*, 21(2):121–133, June 2007.

[74] Kevin Knight and Yaser Al-Onaizan. Translation with finite-state devices. In David Farwell, Laurie Gerber, and Eduard Hovy, editors, *Machine Translation and the Information Soup: Third Conference of the Association for Machine Translation in the Americas AMTA '98 Langhorne, PA, USA, October 28-31, 1998 Proceedings*, volume 1529 of *Lecture Notes in Artificial Intelligence*, pages 421–437, Langhorne, Pennsylvania, USA, October 1998. Springer-Verlag.

[75] Kevin Knight and Jonathan Graehl. Machine transliteration. *Computational Linguistics*, 24(4):599–612, 1998.

[76] Kevin Knight and Jonathan Graehl. An overview of probabilistic tree transducers for natural language processing. In *Computational Linguistics and Intelligent Text Processing: 6th International Conference, CICLing 2005*, volume 3406 of *Lecture Notes in Computer Science*, pages 1–24, Mexico City, 2005. Springer Verlag.

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

[78] Philipp Koehn and Kevin Knight. Feature-rich statistical translation of noun phrases. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Sapporo, Japan, July 2003.

[79] Okan Kolak, William Byrne, and Philip Resnik. A generative probabilistic OCR model for NLP applications. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 55–62, Edmonton, Canada, May-June 2003.

[80] Kimmo Koskenniemi. Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki, 1983.

[81] Werner Kuich. Formal power series over trees. In Symeon Bozapalidis, editor, *Proceedings of the 3rd International Conference on Developments in Language Theory (DLT)*, pages 61–101, Thessaloniki, Greece, 1998. Aristotle University of Thessaloniki.

[82] Werner Kuich. Tree transducers and formal tree series. *Acta Cybernet.*, 14:135–149, 1999.

[83] Shankar Kumar and William Byrne. A weighted finite state transducer implementation of the alignment template model for statistical machine translation. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 63–70, Edmonton, Canada, May-June 2003.

[84] Irene Langkilde and Kevin Knight. The practical value of n-grams in generation. In *Proceedings of the Ninth International Workshop on Natural Language Generation*, pages 248–255, Niagara-on-the-Lake, Ontario, Canada, August 1998.

[85] Karim Lari and Steve Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[86] Andrej Ljolje and Michael D. Riley. Optimal speech recognition using phone recognition and lexical access. In *Proceedings of ICSLP-92*, pages 313–316, 1992.

[87] M. Magidor and G. Moran. Probabilistic tree automata. *Israel Journal of Mathematics*, 8:340–348, 1969.

[88] Andreas Maletti. The power of tree series transducers of type I and II. In Clelia De Felice and Antonio Restivo, editors, *Proceedings of the 9th International Conference on Developments in Language Theory (DLT), Palermo, Italy*, volume 3572 of *Lecture Notes in Computer Science*, pages 338–349, Berlin, 2005.

[89] Andreas Maletti. Compositions of tree series transformations. *Theoretical Computer Science*, 366:248–271, 2006.

[90] Andreas Maletti. Compositions of extended top-down tree transducers. *Information and Computation*, 206(9–10):1187–1196, 2008.

[91] Andreas Maletti, 2009. Personal Communication.

[92] Andreas Maletti, Jonathan Graehl, Mark Hopkins, and Kevin Knight. The power of extended top-down tree transducers. *SIAM Journal on Computing*, 39(2):410–430, 2009.

[93] Daniel Marcu and William Wong. A phrase-based, joint probability model for statistical machine translation. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 133–139, Philadelphia, July 2002.

[94] Lambert Mathias and William Byrne. Statistical phrase-based speech translation. In *IEEE Conference on Acoustics, Speech and Signal Processing*, pages 561–564, Toulouse, France, 2006.

[95] Jonathan May and Kevin Knight. A better n-best list: Practical determinization of weighted finite tree automata. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 351–358, New York City, June 2006.

[96] Jonathan May and Kevin Knight. Tiburon: A weighted tree automata toolkit. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *Proceedings of the 11th International Conference of Implementation and Application of Automata, CIAA 2006*, volume 4094 of *Lecture Notes in Computer Science*, pages 102–113, Taipei, Taiwan, August 2006. Springer.

[97] Jonathan May and Kevin Knight. Syntactic re-alignment models for machine translation. In Jason Eisner and Taku Kudo, editors, *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 360–368, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[98] Bernard Merialdo. Tagging english text with a probabilistic model. *Computational Linguistics*, 20(2):155–161, 1994.

[99] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312, June 1997.

[100] Mehryar Mohri. Generic ϵ–removal and input ϵ–normalization algorithms for weighted transducers. *International Journal of Foundations of Computer Science*, 13(1):129–143, 2002.

[101] Mehryar Mohri. Weighted automata algorithms. In *Handbook of Weighted Automata*, chapter 6, pages 213–254. Springer-Verlag, 2009.

[102] Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. A rational design for a weighted finite-state transducer library. In *Proceedings of the 7th Annual AT&T Software Symposium*, September 1997.

[103] Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231:17–32, January 2000.

[104] Mehryar Mohri and Michael Riley. An efficient algorithm for the *n*-best strings problem. In John H. L. Hansen and Bryan Pellom, editors, *7th International Conference on Spoken Language Processing (ICSLP2002 - INTERSPEECH 2002)*, pages 1313–1316, Denver, Colorado, USA, September 2002.

[105] Franz Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 160–167, Sapporo, Japan, July 2003. Association for Computational Linguistics.

[106] Franz Och and Hermann Ney. Improved statistical alignment models. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, pages 440–447, Hong Kong, October 2000.

[107] Franz Och and Hermann Ney. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449, 2004.

[108] Adam Pauls and Dan Klein. K-best A* parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 958–966, Suntec, Singapore, August 2009.

[109] Fernando Pereira and Michael Riley. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, chapter 15, pages 431–453. MIT Press, Cambridge, MA, 1997.

[110] Fernando Pereira, Michael Riley, and Richard Sproat. Weighted rational transductions and their application to human language processing. In *Human Language Technology*, pages 262–267, Plainsboro, NJ, March 1994. Morgan Kaufmann Publishers, Inc.

[111] Slav Petrov and Dan Klein. Learning and inference for hierarchically split PCFGs. In *AAAI 2007 (Nectar Track)*, 2007.

[112] Michael Rabin and Dana Scott. Finite automata and their decision properties. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[113] Alexander Radzievsky. Corrective modeling for parsing with semantic role labels. Master's thesis, Université de Genève, February 2008.

[114] Sujith Ravi and Kevin Knight. Minimized models for unsupervised part-of-speech tagging. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 504–512, Suntec, Singapore, August 2009. Association for Computational Linguistics.

[115] Giuseppe Riccardi, Roberto Pieraccini, and Enrico Bocchieri. Stochastic automata for language modeling. *Computer Speech & Language*, 10(4):265–293, 1996.

[116] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287, 1970.

[117] Arto Salomaa and Matti Soittola. *Automata-Theoretic Aspects of Formal Power Series*. Springer-Verlag, 1978.

[118] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[119] Stuart M. Shieber. Synchronous grammars as tree transducers. In *Proceedings of the Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+ 7)*, pages 88–95, Vancouver, British Columbia, Canada, May 2004.

[120] Stuart M. Shieber and Yves Schabes. Synchronous tree-adjoining grammars. In Hans Karlgren, editor, *Papers presented to the 13th International Conference on Computational Linguistics (COLING)*, volume 3, pages 253–258, Helsinki, 1990.

[121] Khalil Sima'an. Computational complexity of probabilistic disambiguation by means of tree-grammars. In *COLING 1996 Volume 2: The 16th International Conference on Computational Linguistics*, pages 1175–1180, 1996.

[122] Achim Siztus and Stefan Ortmanns. High quality word graphs using forward-backward pruning. In *Proceedings of the IEEE Conference on Acoustic, Speech and Signal Processing*, pages 593–596, Phoenix, Arizona, 1999.

[123] Richard Sproat, William Gales, Chilin Shih, and Nancy Chang. A stochastic finite-state word-segmentation algorithm for Chinese. *Computational Linguistics*, 22(3):377–404, September 1996.

[124] Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, June 1995.

[125] Joseph Tepperman. *Hierarchical methods in automatic pronunciation evaluation*. PhD thesis, University of Southern California, August 2009.

[126] Joseph Tepperman and Shrikanth Narayanan. Tree grammars as models of prosodic structure. In *Proceedings of InterSpeech ICSLP*, pages 2286–2289, Brisbane, Australia, September 2008.

[127] James W. Thatcher. Generalized^2 sequential machines. *Journal of Computer System Science*, pages 339–367, 1970.

[128] James W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, pages 143–172. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[129] Gertjan van Noord. Treatment of epsilon moves in subset construction. *Computational Linguistics*, 26(1):61–76, 2000.

[130] Gertjan van Noord and Dale Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Automata Implementation, 4th International Workshop on Implementing Automata, WIA'99*, volume 2214 of *Lecture Notes in Computer Science*, pages 122–139, Potsdam, Germany, 1999. Springer-Verlag.

[131] Stephan Vogel, Hermann Ney, and Christoph Tillmann. HMM-based word alignment in statistical translation. In *COLING96: Proceedings of the 16th International Conference on Computational Linguistics*, pages 836–841, Copenhagen, August 1996.

[132] Wei Wang, Jonathan May, Kevin Knight, and Daniel Marcu. Re-structuring, relabeling, and re-aligning for syntax-based machine translation. *Computational Linguistics*, 36(2), June 2010. To appear.

[133] William A. Woods. Cascaded ATN grammars. *American Journal of Computational Linguistics*, 6(1):1–12, January-March 1980.

[134] Dekai Wu. A polynomial-time algorithm for statistical machine translation. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 152–158, Santa Cruz, California, USA, June 1996. Association for Computational Linguistics.

[135] Dekai Wu. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23(3):377–404, 1997.

[136] Kenji Yamada and Kevin Knight. A syntax-based statistical translation model. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*, pages 523–530, Toulouse, France, July 2001.

[137] David Zajic, Bonnie Dorr, and Richard Schwartz. Automatic headline generation for newspaper stories. In *Proceedings of the ACL-02 Workshop on Text Summarization (DUC 2002)*, pages 78–85, Philadelphia, PA, July 2002. Association for Computational Linguistics.

[138] Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. Synchronous binarization for machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 256–263, New York City, June 2006.

[139] Andreas Zollmann and Khalil Sima'an. A consistent and efficient estimator for data-oriented parsing. *Journal of Automata, Languages and Combinatorics*, 10(2/3):367–388, 2005.