RC24829 (W0907-059) July 28, 2009
Computer Science

IBM Research Report

Designing a Non-Finite-State Weighted Transducer Toolkit

Stanley F. Chen
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Designing a Non-Finite-State Weighted Transducer Toolkit

Stanley F. Chen
IBM T.J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598
stanchen@watson.ibm.com

Abstract

Toolkits for weighted finite-state machines (WFSM's) have proven to be tremendously useful in a wide variety of speech and language applications. While WFSM's can directly represent finite-state statistical models such as hidden Markov models, this is not the case for many models of interest. In this paper, we consider extending a WFSM toolkit to a non-finite-state formalism. We select a formalism that is both useful and efficient to compute with, and analyze what finite-state operations can be extended to this automaton class. We describe a design for a toolkit for manipulating these automata, and give examples of how our toolkit can be used to quickly train and evaluate models for a variety of language tasks.

1 Introduction

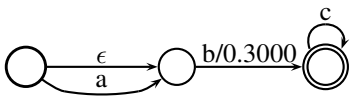
Weighted finite-state machine toolkits have shown themselves to be wonderfully useful and effective in speech and language tasks (Mohri et al., 1998). For example, WFSM's are a natural representation for finite-state statistical models, and WFSM toolkits make it easy to manipulate and compute with these types of models. However, there is a wide class of interesting statistical models that cannot be directly expressed as a WFSM, such as probabilistic context-free grammars (CFG's) and other non-finite-state formalisms. Even for statistical models that are technically finite-state, the size of the state space may be so large that direct representations of the model may be impractical to compute with.

In this paper, we ask the question: what would happen if we extended a WFSM toolkit to support a non-finite-state formalism? First, we outline extensions to WFSM's that would make it possible to express a rich class of graphical models. We analyze what finite-state operations can be extended to this automata class and argue that we can still do key operations efficiently. Next, we describe a design for a toolkit for manipulating these automata, and show how model training can be integrated into the toolkit. We present examples of how our toolkit can be used to quickly construct and evaluate interesting models for tasks such as part-of-speech tagging, language modeling, and parsing. Finally, we relate this toolkit to other software tools for statistical modeling.

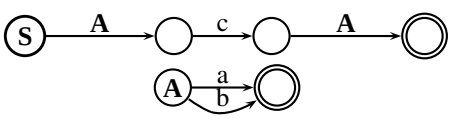
2 Extending Finite-State Automata

We begin by reviewing the definition of a weighted finite-state acceptor (WFSA). An example WFSA is shown in Figure 1(a); it consists of a finite set of states with a distinguished start state (bold circle) and set of final states (double circle), and a set of transitions labeled with elements

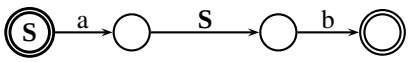
(a)



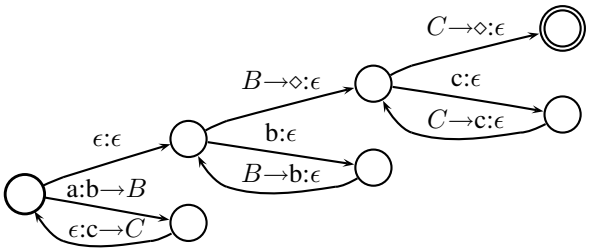
(b)



(c)



(d)



(e)

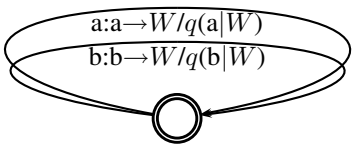


Figure 1: Example automata: (a) WFSA (b) RTN (c) $a^n b^n, n \geq 0$ (d) $a^n b^n c^n, n \geq 0$ (e) n-gram model.

from some finite alphabet or the empty string ϵ. In addition, transitions and final states have associated weights (the value after the '/', or 0 if omitted). We now consider several extensions to WFSA's; the utility of each on realistic tasks will be demonstrated in Section 6.

As context-free grammars are widely used in statistical modeling, we first propose to make FSA's as expressive as context-free grammars. The key difference between CFG's and FSA's is that CFG's allow grammar symbols to be defined recursively in terms of other symbols. We can add this functionality to FSA's by organizing the states of an FSA into a (usually disjoint) set of named sub-FSA's, each corresponding to the expansion of a single symbol in a CFG. Then, we allow transitions to be labeled with the name of a sub-FSA. For example, consider the automaton in Figure 1(b); this corresponds to the CFG containing the rules $S \Rightarrow AcA$ and $A \Rightarrow a|b$. The name for a sub-FSA is placed inside of its start state, and the bold circle is the start state for the "start" symbol. Another example is given in Figure 1(c), corresponding to the CFG $S \Rightarrow \epsilon|aSb$ which accepts the non-regular language $a^n b^n, n \geq 0$. This type of augmented FSA is called a *recursive transition network* (RTN) (Woods, 1970), and any FSA or CFG can be simply converted to an RTN. The set of languages accepted by RTN's is exactly the set of context-free languages.

The next change we propose is to add a *tape*, as in a Turing machine tape. In statistical models on sequential data, we generally need to condition on the past. To do this, it is very useful to have some sort of buffer, like a tape, where we can store previous symbols. For reasons to be given in Section 5.1, instead of supporting bidirectional read and write operations on tapes, we only consider tapes that behave like *stacks*. More concretely, we allow an automaton to have a finite number of

named stacks, and allow transitions to have labels of the form $a \rightarrow s$ or $s \rightarrow a$ corresponding to a "push" or "pop", respectively, of the symbol a from stack s. Initially, we assume stacks contain a single start symbol which we denote as '$\diamond$'. For example, consider the automaton in Figure 1(d) which uses the two stacks B and C and which accepts the non-context-free language $a^n b^n c^n, n \geq 0$. (In transition labels, symbols before and after the colon correspond to input and output symbols, respectively, possibly from/to a stack.) This demonstrates the power of stacks in remembering the past, and a finite-state automaton with as few as two stacks is equivalent to a Turing machine.

The final extension on our wish list is to allow *weight distribution values* to be placed on transitions in addition to simple weights. To motivate and explain this, consider an n-gram model. Models of this type can be represented using a WFSA that has one state for each n-gram history in the model. However, if we have a stack, we can store the last $n-1$ symbols on the stack, and we can reduce the number of needed states. Let us consider the 1-state automaton given in Figure 1(e) and see whether it can express an n-gram model (over the alphabet $\{a, b\}$). If the notation $q(a|W)$ represents a simple weight, we can only express a unigram model. However, what if $q(\cdot|\cdot)$ represents a conditional probability distribution (or more generally, a *weight* distribution), so that the value $q(a|W)$ can be an arbitrary function of a and the contents of stack W? By design, the stack W contains all previous symbols, so we can take $q(a|W)$ to be equal to the corresponding n-gram probability, and thus express an n-gram model using an automaton with a single state. As will be shown later, it is very useful to be able to shift complexity from an automaton's topology to within a weight distribution. Another motivation for supporting distributions is that it becomes straightforward to express many types of graphical models, as they may have arbitrary probability distributions at each node.

3 Multistack Pushdown Automata

Here, we provide a formal definition of the automaton type described in the last section, which can be considered to be a *multistack pushdown automaton* (MPDA). While our definition will not match traditional definitions exactly, it was chosen to closely match our implementation. We initially consider unweighted MPDA's that are acceptors; extension to weighted transducers will be discussed later. Then, an MPDA is a 9-tuple $(Q, \Sigma, N, \mathcal{S}, \delta, q_0, \diamond, F, \mathcal{N}_0)$ where Q is a finite set of states; Σ is a finite input alphabet; N is a set of nonterminal symbols (*i.e.*, sub-FSA names); S is a finite set of stacks; $\delta \subseteq Q \times (\Sigma \cup N \cup (\Sigma \times \mathcal{S} \times \{R, W\}) \cup \{\epsilon\}) \times Q$ is a set of transitions with source state, label, and destination state; $q_0 \in Q$ is the initial state; $\diamond \in \Sigma$ is the stack start symbol; $F \subseteq Q$ is a set of final states; and $\mathcal{N}_0 : N \rightarrow Q$ is a one-to-one mapping describing the start state for each nonterminal/sub-FSA. A transition label can either be a simple label $a \in \Sigma$; a nonterminal $n \in N$ (subroutine call); take the form (a, s, action) (read/write symbol $a \in \Sigma$ from/to stack $s \in \mathcal{S}$); or be empty (ϵ).

To describe which strings an MPDA accepts, we use the concept of an *instantaneous description* (ID) (Hopcroft and Ullman, 1979). The instantaneous description of an MPDA is a 4-tuple $(q, w, \gamma, \vec{\beta})$ where $q \in Q$ is the current state; w is the string of input symbols yet to be accepted; $\gamma \in Q^*$ is the list of states on the *call stack*; and $\vec{\beta}$ is the list of stack values for each stack in $\mathcal{S}$. We call γ the *call stack* as it is analogous to a call stack in a computer program.

Define $\vec{\beta} \cdot_s a$ to be $\vec{\beta}$ with the value a pushed on stack $s \in \mathcal{S}$. Then, we define the "reachability" relation $\vdash$ on ID's as follows: $(q, aw, \gamma, \vec{\beta}) \vdash (p, w, \gamma, \vec{\beta})$ if $(q, a, p) \in \delta$ (regular transition); $(q, w, \gamma, \vec{\beta}) \vdash (\mathcal{N}_0(n), w, \gamma p, \vec{\beta})$ if $(q, n, p) \in \delta$ (subroutine call); $(q, w, \gamma p, \vec{\beta}) \vdash (p, w, \gamma, \vec{\beta})$ if $q \in F$ (subroutine return); $(q, w, \gamma, \vec{\beta}) \vdash (p, w, \gamma, \vec{\beta} \cdot_s a)$ if $(q, (a, s, W), p) \in \delta$ (stack write); and $(q, w, \gamma, \vec{\beta} \cdot_s a) \vdash (p, w, \gamma, \vec{\beta})$ if $(q, (a, s, R), p) \in \delta$ (stack read). We define $\vdash^*$ to be the reflexive

and transitive closure of $\vdash$.

A string w is accepted by the automaton iff $(q_0, w, \epsilon, \vec{\beta}_0) \vdash^* (p, \epsilon, \epsilon, \vec{\beta})$ for some $p \in F$ and some $\vec{\beta}$, where $\vec{\beta}_0$ corresponds to each stack in $\mathcal{S}$ holding just '$\diamond$'. Note that an ID is final iff we are at a final state *and* the call stack is empty; the other stacks need not be empty. A useful generalization is to allow $\vec{\beta}_0$ to be defined in a task-specific manner.

We can extend MPDA's to weighted MPDA's (WMPDA's) in exactly the same way that FSA's have been extended to weighted FSA's (Mohri et al., 2002). Similarly, we can also extend MPDA's to be transducers. To add support for weight distributions, we augment WMPDA's by also specifying a finite list of distributions $(P_1, P_2, \ldots)$ and optionally attaching a weight distribution value $P_i(a|\vec{\beta})$ for $a \in \Sigma$ to each transition, where a distribution $P_i(\cdot|\cdot)$ is a function of a and $\vec{\beta}$ that returns a weight. The total weight of a transition is then its simple weight combined with $P_i(a|\vec{\beta})$ (or 0 if absent), where the stack state $\vec{\beta}$ used is the state on entry.

4 Computing with MPDA's

Here, we consider what finite-state operations can be efficiently extended to MPDA's. We consider the key operations of composition and automaton optimization, *i.e.*, determinization and minimization.

4.1 Composition

We first consider composition on unweighted pushdown automata (PDA's), MPDA's with no nonterminals and at most one stack. For acceptors, composition is equivalent to intersection. An algorithm for computing the intersection of an FSA and PDA is given by Bar-Hillel et al. (1961); the result is computed as a PDA. It is straightforward to generalize this algorithm to the composition of two MPDA's, where the composition of MPDA's with n and m stacks (including the call stack) can be computed as an MPDA with $n + m$ stacks. However, this is not as useful as it sounds, as many seemingly simple operations are intractable for MPDA's; *e.g.*, it is undecidable whether an arbitrary two-stack machine accepts no strings, and thus finding the lowest-weight path in an arbitrary MPDA is also undecidable.

What we would really like is for composition to yield an FSA if the result is a regular language, as FSA's are much easier to manipulate than general MPDA's. Note that an MPDA can be viewed as an FSA with a possibly infinite state space; states in this FSA would correspond to triples of the form $(q, \gamma, \vec{\beta})$. By simulating an MPDA using an FSA, we can frame FSA/MPDA composition as an instance of FSA/FSA composition, which yields an FSA as the result. Efficient composition implementations only access those states in an input automaton that are "reachable" given the other input machine, so this algorithm, which we refer to as *finite-state* composition, can be fast even if the MPDA corresponds to an infinite FSA. However, there is no guarantee that composition will terminate for arbitrary input automata; in general, it is undecidable whether even an arbitrary context-free language is regular. It is straightforward to extend both general MPDA/MPDA composition and finite-state composition to weighted transducers.

4.2 Automaton Optimization

It is undecidable whether an arbitrary PDA is determinizable. Since we cannot determinize arbitrary PDA's, neither can we compute the minimum state deterministic automaton for an arbitrary context-free language, as can be done for regular languages.

However, experience with FSM's suggests that whether an automaton can be minimized in theory is not important. In language processing, we often work with ambiguous weighted finite-state machines that cannot be determinized, and we cope by using heuristic methods, *e.g.*, (Allauzen and Mohri, 2003). Furthermore, while there is an efficient algorithm for finding the *minimum state deterministic* automaton for some classes of languages, this is not always the criterion we wish to optimize.

Just as we don't require compilers to produce the absolute smallest or fastest executable, neither do we need our optimization algorithms to produce truly optimal automata. Ultimately, what matters is whether we can build a useful set of optimization tools that work well in practice. While an in-depth discussion of automata optimization is beyond the scope of this paper, we observe that determinization and minimization algorithms for FSM's can still be profitably applied to MPDA's, and we provide an example in Section 6.3.

5 Toolkit Design

In this section, we discuss the design of the IBM Infinite-State Machine (ISM) Toolkit, which can be used to manipulate WMPDA's efficiently. This toolkit is still very much under construction.

As efficiency is of paramount concern, the basic design uses C++ and generic programming. Borrowing ideas from the C++ Standard Template Library (STL) and the Boost Graph Library (Siek et al., 2001), we defined interfaces, or *concepts*, for each automaton type (analogous to STL containers), and templatized algorithms that apply to one or more concepts (analogous to STL algorithms). In addition to the C++ interface to the library, we also used SWIG to generate a Python interface.

5.1 Technical Details

We implemented WMPDA's exactly as defined in Section 3. We defined a text syntax for describing WMPDA's similar to the syntax used by the AT&T FSM library, except that the transition label syntax was extended to support subroutine calls and stack operations. In addition, there is a syntax for specifying the start state for each sub-FSM, and weight distribution values can be attached to transitions.

We defined an abstract C++ interface for weight distributions to allow the list of distribution implementations to be easily expanded. Completed implementations include n-gram models and discrete exponential models.

To implement finite-state composition efficiently, it is necessary to store large collections of stacks compactly. We note that collections of "similar" stacks can be stored efficiently using a tree structure. Tapes that support bidirectional reads and writes do not share this property, which is why we chose to implement only stacks. Another important issue in composition is that if a stack holds unneeded state, efficiency can be drastically affected. For example, if a stack is used to hold the history for an n-gram model, the stack size should be bounded at $n - 1$ elements. To this end, we defined an abstract C++ interface for stacks to allow new implementations to be easily added, and currently support unbounded and fixed-size stacks. This feature allows the toolkit to achieve the same order of complexity for dynamic programming operations as task-specific implementations in many situations.

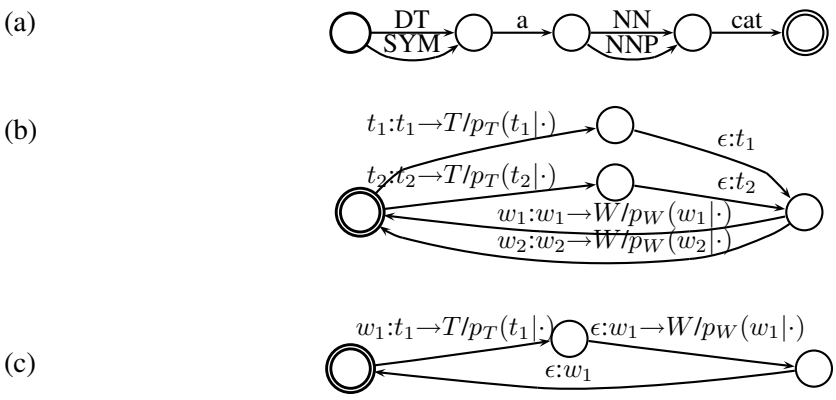


Figure 2: Tagging automata: (a) example input lattice (b) tagging WMPDA over tags t_1, t_2 and words w_1, w_2 (c) WMPDA for lattice rescoring, replicating whole loop for each word/tag pair in training set.

5.2 Training

Because of the relatively simple structure of weights in WFSM's, there is not much interesting weight training one can do without taking advantage of task-specific knowledge.[1] On the other hand, there are vast numbers of task-independent *distribution* trainers employing a diverse range of technologies (*e.g.*, maximum entropy, neural networks, SVM's, etc.). Given a WMPDA and a supervised training corpus,[2] it is straightforward to extract the "events" needed to train each weight distribution in a WMPDA. By feeding these events into an existing trainer and writing a weight distribution implementation that can read trained models of the given type, one can train and use new distribution types within the toolkit. Thus, supervised training can be done easily, potentially for a wide range of distributions.

6 Toolkit Examples

6.1 Part-of-Speech Tagging

Let's consider the example of building a part-of-speech tagger. Consider a classic trigram part-of-speech tagging model

$$p_1(T, W) = \prod_{i=1}^{n} p_T(t_i|t_{i-2}t_{i-1})p_W(w_i|t_i) \tag{1}$$

for a tag sequence T and word sequence W, where we take $\textit{tags}(W) = \arg\max_T p_1(T, W)$. First, we need to design an MPDA that implements this model. Here, we assume the tagger will be given an input lattice of the form given in Figure 2(a), where all possible tags for a word precede each word.[3]

Then, we can use an MPDA of the form given in Figure 2(b). Each tag and word read from the input is immediately written to stacks T and W, respectively, and assigned a probability of the

[1] One approach for this is given by Eisner (2002).

[2] A supervised training corpus is one where composition between the given WMPDA and each element in the corpus yields exactly one full path.

[3] For words in the training data, we restrict the possible tags to those which have labeled the word in the training.

form $p_T(t|T,W)$ or $p_W(w|T,W)$. In this example, both distributions only condition on stack T. In addition, each tag read is also written to output.

With the ISM toolkit, we can construct this model by first generating a text description of the MPDA. Then, we run one command to extract events suitable for our trainer, and then one command to train each of the distributions $p_T(t|\cdot)$ and $p_W(w|\cdot)$ using our exponential model trainer.[4] To use this MPDA for tagging, we wrote a script to generate input lattices of the form in Figure 2(a) for each input utterance. We then did lazy finite-state composition of the input lattice with the MPDA followed by beam pruning, and computed the highest probability path in the resulting lattice; each of these operations is a single call in the ISM library.

We ran experiments on Wall Street Journal (WSJ) data from the Penn Treebank 3; we used the same data splits as in (Toutanova et al., 2003). On a training set of about 1M words, training took less than three minutes. All told, it took a few hours from start to finish, including constructing the MPDA and writing the evaluation infrastructure, yielding a tagging accuracy of 94.61% on the development set. However, we can use the same infrastructure to evaluate other tagging models by just changing the weight distributions used in the MPDA. Note that the forms of the distributions $p_T(t|T,W)$ and $p_W(w|T,W)$ used in the model are quite general. For example, consider the model

$$\begin{aligned} p_2(T,W) &= \prod_{i=1}^{n} p_T(t_i|t_{i-2}t_{i-1}, w_{i-2}w_{i-1}) \times \\ &\quad p_W(w_i|t_i, w_{i-2}w_{i-1}) \end{aligned} \tag{2}$$

where we now consider dependencies between each tag and word and previous words. Again, we ran a single command to train each of these new distributions (taking <40 minutes); all told, it took less than an hour to build and evaluate this model (accuracy: 95.63%). Tagging speeds are 5000 words/sec and 3300 words/sec for the two models on a 2.8 GHz Xeon. For reference, Toutanova et al. (2003) achieved an accuracy of 97.15% on this test set; however, the above models do not use orthographic information to help tag words not in the training data. We can integrate this type of information by using a spelling-based model to assign tag weights for these words in the input lattices.

While the above models are *source-channel* models, it is also easy to build *direct* models of the form $p(T|W)$, or to use weights that are not probabilities at all. For example, consider the following direct model

$$p_3(T|W) = \prod_{i=1}^{n} p(t_i|w_i, t_{i-2}t_{i-1}) \tag{3}$$

where we condition the next tag on the current word and previous tags. In this model, it is more natural to input the current word before the current tag, so we modify the input lattice format in Figure 2(a) appropriately; we modify the MPDA in Figure 2(b) similarly and eliminate the distribution $p_W(w|\cdot)$. Then, we can use the same procedure as before to construct and evaluate this model (accuracy: 93.77%).

6.1.1 Class-Based Language Models

We observe that the tagging models given in eqs. (1) and (2) can also be used as language models (LM's); *i.e.*, $p(W) = \sum_T p(T,W)$, where in practice we use the approximation $p(W) \approx \max_T p(T,W)$. To evaluate the perplexity (PP) of the same WSJ development set as before, we

[4] Our trainer builds n-gram feature histories separately for each stack, and trains a model smoothed with a Gaussian prior (Chen and Rosenfeld, 2000) using a variant of iterative scaling.

can use the same procedure used in tagging. The tag sequence returned is $\arg\max_T p(T, W)$, from which we can directly compute our estimate of $p(W)$. The PP's of the models given by eqs. (1) and (2) are 520.9 and 193.0 (or 186.7 if we condition on three tags in the past rather than two).[5] In contrast, the PP of a baseline trigram model smoothed with modified Kneser-Ney smoothing is 218.7 (or 210.8 for a 5-gram model). Thus, we can reduce the PP over a baseline n-gram model by over 10% simply by using part-of-speech tags, which is the largest reduction of this kind that we are aware of.

One application of language models is word lattice rescoring in automatic speech recognition (ASR). Lattice rescoring can be implemented by composing a word lattice containing acoustic scores with a weighted automaton representing the LM, and thus is straightforward with our toolkit. However, input word lattices do not contain tags, as is expected by the MPDA used earlier, so we modify our MPDA topology as given in Figure 2(c). As a proof of concept, we rescored a 49k-word WSJ test set using the acoustic model described in (Chen, 2008) with a baseline Katz-smoothed word trigram model and with our tag-based model, yielding word-error rates of 28.0% and 25.5%, respectively.

While our tag-based models use "soft" classing in that a word may be assigned multiple tags, most class-based LM's use "hard" classing, *e.g.*, (Brown et al., 1992). Clearly, hard classes are a special case of soft classes, and can be implemented using the same techniques described above. For example, the class-based language modeling experiments in (Chen, 2008) were carried out using these tools.

6.2 Grammar-Based Letter Language Models

In this example, we demonstrate how a WMPDA can be used to combine grammatical and n-gram information in constructing a language model on letters, to model the spellings of individual words. English words are composed of a sequence of syllables, each of which consists of an optional *onset* (initial consonants), a *nucleus* (vowels), and optional *coda* (final consonants). We investigate whether we can use this structure to improve over a baseline n-gram model in perplexity and in syllabification accuracy. We used a 77k-word manually-syllabified training set and 5k-word test set; the test perplexity of a 7-gram model is 7.34. For syllabification, we trained an n-gram model on the training set with a distinguished boundary token inserted at each syllable boundary. To use this model to syllabify a word, we construct an input lattice consisting of the word spelling with an optional syllable boundary inserted at each position. We converted our n-gram model to a 1-state WMPDA as described in Section 2, and then decode as before. We achieve a recall and precision of 90.8% and 91.2%, respectively.

To use grammatical information, we start with a CFG backbone as follows:

$$\begin{array}{ccl} \textit{word} & \rightarrow & \textit{syl} \mid \textit{syl word} \\ \textit{syl} & \rightarrow & \textit{onset nucl coda} \mid \textit{onset nucl} \mid \ldots \\ \textit{onset} & \rightarrow & \textit{consonant} \mid \textit{consonant onset} \\ \textit{nucl} & \rightarrow & \textit{vowel} \mid \textit{vowel nucl} \\ \ldots & \ldots & \ldots \end{array}$$

We convert this to a WMPDA and place a weight distribution value on each branch of each branch point in the automaton. We first consider parameterizing this as a conventional probabilistic CFG, by conditioning each prediction only on its parent symbol.[6] We can train and evaluate this model

[5] These are actually upper bounds on the true PP.

[6] In practice, we did this by building a separate distribution for each parent symbol, but there are other ways of doing this.

using the same procedure described in Section 6.1. To use this model for syllabification, we convert the WMPDA to a transducer and output a syllable marker at the end of each syllable. This model achieves a PP of 18.56 and a recall and precision of 62.4% and 62.7%. As with many naive grammatical models, this model performs poorly because predictions aren't condition strongly on "lexical" information, as is done in n-gram models.

To address this, we can add a stack L that we push each letter on as it is read. Then, it is straightforward to condition on past letters in every prediction. We trained new distributions conditioning on the past five letters, and this yielded a PP of 7.56 and recall and precision of 89.2% and 89.8%. While these results do not surpass the n-gram baseline as hoped, this example demonstrates how our toolkit can be used to quickly build and evaluate interesting grammar-based models. The hardest part of this example was designing the WMPDA topology; training and evaluation involved running a few commands and writing a simple evaluation script.

6.3 Automaton Optimization

To demonstrate how automaton optimization algorithms can be applied to MPDA's, we use a contrived example. Consider the following grammar:

sentence	$\rightarrow$	*word* | *word sentence*
word	$\rightarrow$	*CC* | *DT* | *JJ* | ...
CC	$\rightarrow$	'a' 'n' 'd' | 'o' 'r' | 'n' 'o' 'r' | ...
DT	$\rightarrow$	't' 'h' 'e' | 'a' | 't' 'h' 'i' 's' | ...
...	...	...

(A similar grammar over phonemes rather than letters can be used to rescore phone lattices for ASR.) In this example, we take sentences split into characters and use the above grammar to transduce the input character sequence into a sequence of words with corresponding tags. To do this, we convert the grammar to a transducer and output the associated word and tag at the end of each word spelling.

Using the above grammar (built using all word/tag pairs in our WSJ training set) converted to an MPDA, we can transduce WSJ data (via lazy composition, pruning, and computing the best path) at a rate of 130 chars/sec. We can optimize the MPDA by determinizing and minimizing each sub-FSM separately, treating each as a normal FSM and using regular finite-state operations. After optimization, we can transduce at 2800 chars/sec. However, we can additionally optimize the grammar by using *inlining*, as in regular program optimization. We can substitute the expansion of each tag directly into the sub-FSM corresponding to the *word* symbol; instead of optimizing the word spellings for each tag separately, we can group together all words for all tags and optimize this as a single sub-FSM. By doing so, transduction goes to 16000 chars/sec. We have used the same optimization techniques on a grammar for the C++ language, and have successfully used the toolkit to parse C++ code.

We note that in conventional parsing, the output of interest is the parse structure. Optimization techniques such as inlining may disturb the grammar topology and alter the corresponding output parses. However, when using *transducers*, one generally takes the automata output to be the output of interest. Then, we have the advantage that we can do optimization without altering the target output.

7 Related Work

Obviously, the largest influence of this work is the AT&T finite-state transducer library (Mohri et al., 1998), which has had an immense impact on the field of language processing and others. Just as WFSM's are powerful because they can represent languages containing exponential (or infinite) numbers of strings using a small number of states, WMPDA's are powerful because they can represent WFSM's containing exponential (or infinite) numbers of states using a small number of stacks. For many models, this capability lets us transfer the complexity of managing a large state space from the toolkit user to the toolkit itself, greatly lessening a user's cognitive burden. Additionally, by encapsulating state structure within weight distributions, it becomes natural to integrate weight training within a toolkit. For example, consider n-gram model training. In a WFSM, n-gram selection amounts to topology induction, which is difficult to do without task-specific knowledge. In a WMPDA, n-gram model training is distribution training, which is straightforward. The most closely related work on extending a WFSM library is probably the work by Kempe et al. (2004) on *weighted multi-tape automata*. However, this work only considers tapes that are read-only or write-only.

We can also compare our toolkit with other tools for building statistical models. Perhaps the work with the most similar goals is Dyna, a weighted logic programming language (Eisner et al., 2005). A wide range of graphical models can be expressed compactly in Dyna, including models that are difficult to express as a WMPDA. Our design trades off flexibility for efficiency as compared to Dyna, in that our automaton operations are written directly in C++, while in Dyna one generally writes algorithms declaratively. Another category of related software is graphical modeling toolkits such as GMTK (Bilmes and Zweig, 2002) and BNT (Murphy, 2001). Again, these can do many of the same things as our toolkit (and much more). However, our toolkit can handle non-finite-state formalisms such as context-free grammars and related models. In addition, a significant benefit of automaton-based toolkits is that off-line automaton optimization is possible, which can drastically accelerate automaton operations (Mohri et al., 2002).

8 Discussion

A lot of work remains to be done. Of particular interest are integrating more distribution types and trainers; developing effective automaton optimization techniques for stack automata, including algorithms that can take advantage of the structure of weight distributions; developing tools to support unsupervised training; and eventually providing an open-source release of the toolkit.

Clearly, what software is available has a great impact on how we perceive research problems and on what research gets done. In particular, there is a strong tendency to focus on models that are "easy" to build, where "easiness" is determined by some magical combination of simplicity of use, fast execution, scalability to large data sets, documentation, stability, etc. While WFSM toolkits and language modeling toolkits have made it "easy" to build and manipulate finite-state models, it is less clear whether existing software for building more complex models has reached this threshold yet.

As demonstrated in Section 6, the ISM toolkit enables novel and interesting models to be trained and evaluated within a matter of hours. The toolkit can decode efficiently (*e.g.*, tag at 3k-5kw/sec) and distribution trainers that can handle very large data sets (*i.e.*, n-gram model toolkits) can already be used with the toolkit. We posit that we have moved a step closer towards making it possible for researchers to focus on *modeling* decisions (*e.g.*, model topology, parameterization) rather than *implementation* decisions (*i.e.*, what can be done quickly) in many contexts, and anticipate that

toolkits of this type will be widely useful in the field of language processing and others.

References

Cyril Allauzen and Mehryar Mohri. 2003. Generalized optimization algorithm for speech recognition transducers. In *Proceedings of ICASSP*, volume I, pages 352–355.

Yehoshua Bar-Hillel, Micha Perles, and Eliyahu Shamir. 1961. On formal properties of simple phrase structure grammars. *Z. Phonetik. Sprachwiss. Kommunikationsforsch*, 14:143–172.

Jeff Bilmes and Geoffrey Zweig. 2002. The graphical models toolkit. In *Proceedings of ICASSP*.

Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, Jennifer C. Lai, and Robert L. Mercer. 1992. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, December.

Stanley F. Chen and Ronald Rosenfeld. 2000. A survey of smoothing techniques for maximum entropy models. *IEEE Transactions on Speech and Audio Processing*, 8(1):37–50.

Stanley F. Chen. 2008. Performance prediction for exponential language models. Technical Report RC 24671, IBM Research Division, October.

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of HLT-EMNLP*, pages 281–290, October.

Jason Eisner. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Meeting of the ACL*, pages 1–8.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

Andre Kempe, Franck Guingne, and Florent Nicart. 2004. Algorithms for weighted multi-tape automata. Technical Report 2004/031, Xerox Research Centre Europe, June.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 1998. A rational design for a weighted finite-state transducer library. *Lecture Notes in Computer Science*, 1436.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16:69–88.

Kevin Murphy. 2001. The Bayes net toolbox for MATLAB. In *Computing Science and Statistics*, volume 33.

Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2001. *The Boost Graph Library*. Addison-Wesley Professional.

Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL*, pages 252–259.

William A. Woods. 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13:591–606.