The MIT Finite-State Transducer Toolkit for Speech and Language Processing

Lee Hetherington

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

Abstract

We present the MIT Finite-State Transducer Toolkit and briefly describe research that it has benefitted. The toolkit is a collection of command-line tools and associated C++ API for manipulating finite-state transducers (FSTs) and acceptors (FSAs) and has been designed to enable research through its flexibility, yet remain efficient enough to aid real-world computationally demanding applications such as automatic speech recognition. The toolkit supports the construction, combination, optimization, and training of weighted FSTs and FSAs, and as such is useful in many areas of human language technology.

1. Introduction

Finite-state transducers (FSTs), possibly weighted, have long been utilized within a wide range of human language technologies including phonology, morphology, statistical language modeling, part-of-speech tagging, parsing, and speech recognition [1, 2]. FSTs can represent uncertain transformations from one level of representation to another, optionally weighting alternative interpretations or realizations probabilistically. FSTs allow a uniform representation, which in turn allows the use of a mathematical framework with powerful operations to construct, combine, and optimize them effectively. The alternative, using different representations, often interferes the effective composition of a total system from its components and subsequent optimization.

In this paper, we introduce The MIT FST Toolkit we have developed and are now making publicly available. The toolkit provides for the construction of various types of FSTs, their combination, optimization, and weight training. The operations are available at both the command-line executable level, operating on files or through pipes, and through an object-oriented C++ class library for closer integration with applications.

There are other toolkits available, most notably AT&T's FSM [3] and GRM [4] packages. However, at the time of this writing, these packages are publically available only in command-line executable form for research purposes and not as embeddable libraries or source code.

Development of this toolkit began in 1996 when we became frustrated with the use of different representations and algorithms scattered throughout our own speech recognition system. We had different representations for context-dependent model identification, phonological rules, lexicons, and language models. The different representations meant there was some duplication of algorithms for optimizing components, and combining different levels using different representations was problematic.

Inspired by work at AT&T [5], we sought to make use of FSTs for their uniform representation and uniform use of a single toolkit for constructing, combining, and optimizing components. At the time, there were no publicly available FST toolkits that provided an efficient C/C++ API enabling embedding within our speech recognizer. Thus, we developed our own, and by 1997 we had SUMMIT [6] operating with FSTs. We immediately saw gains in terms of efficiency and flexibility.

Over the past six years we have continued to refine and extend the toolkit to support research activities conducted by both students and staff, and it is now in use at a few other sites. In speech recognition, the toolkit has been instrumental in the following research: the compilation of phonological rules [7] and pronunciation weighting [8], the use of complex out-of-vocabulary word models [9], the use of an FST modeling phonetic confusions for non-native speakers [10], modeling of manner/place lingustic features [11], flexible language-model lookahead for large vocabularies [12], seamless integration of multiple languages/domains within a single recognition search [13], flexible recognition vocabularies [14], and multi-pass recognition to enable the verbal entry of new words through the use of spoken and spelled forms within the same utterance [15].

2. Toolkit Design

The three primary goals when designing this toolkit were: ease of use, flexibility, and efficiency. In our research group we have users with a wide range of programming abilities. The command-line interface is the easiest to use and allows many complex FST computations to be implemented by chaining operations together using Unix pipes. For embedding FST operations within larger tools for better efficiency (e.g., a speech recognition system) it is necessary to utilize the C++ class library. The toolkit's class library was designed so as to be flexible and extensible, allowing for the creation of new specialized FST types when needed. Finally, any FST toolkit must be efficient if it is to be utilized on real-world tasks. FSTs utilized in speech recognition can, for example, contain on the order of 10^7 or more states and transitions. The tools must be able to manage FSTs of that size effectively if they are to be genuinely useful in constructing and deploying human language technologies.

2.1. FST Representation

We have constrained FSTs to have a single initial state, one or more final states with weights, and transitions containing a single input symbol, a single output symbol, and a real-valued weight. See Figure 1 for a simple example. Input and output symbols are arbitrary character strings or ϵ (null symbol), and internally are rep-

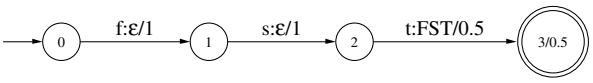

Figure 1: Sample FST mapping symbol sequence *f,s,t* to symbol *FST*. For the $(+, \times, 0, 1)$ semiring the resulting weight would be 0.25, and for the $(\min, +, \infty, 0)$ semiring it would be 3.

resented as integral indices with the mapping managed automatically by the toolkit. FSTs that allow strings of input/output symbols on a single arc can be represented by splitting such an arc into a sequence of arcs, without loss of generality. The use of simple real-valued weights is primarily to maintain speed and space efficiency in large-scale FSTs. Allowing a more general notion of weight supported by an appropriate semiring (e.g., Eisner's expectation semiring [16]) is unfortunately not currently supported by the toolkit.

There is no distinct finite-state acceptor (FSA) representation within the toolkit. FSAs are represented as FSTs with equal input and output labels (e.g., transition labels of the form $x : x$).

2.2. FST Implementations

The toolkit provides several general-purpose FST implementations or types that can be used with all the tools. These include mutable and immutable basic FSTs, caches, and memory-mapped files. The cache types aid efficiency by caching the transitions leaving states and are useful when working with FST operations that utilize lazy evaluation (expansion on demand). There are also special-purpose FST implementations including one supporting back-off n-gram word-class language models, a recursive transition network (RTN) implementation, and dynamic substitution. The use of a common interface allows most FST types to be used by most tools. However, not all types implement all methods (e.g., an immutable type will not support the methods to add states or transitions).

Back-off n-gram word-class language models in ARPA format can be converted to a specialized FST type for efficient storage and optionally deterministic access to transitions.[1] Java Speech API (JSAPI) grammars [17] can be imported into an RTN FST implementation that can be expanded via lazy evaluation if they are not finite-state or statically expanded and optimized if they are finite-state. We also have tools for constructing FSTs and FSAs from regular expressions.

In order to support dynamically changing vocabularies within speech recognition, we have created a specialized FST type that can dynamically expand "non-terminal" labels represented by other FSTs (e.g., a restaurant name class expanding to a set of restaurant names) while respecting context-dependent cross-word constraints across the splice boundaries. This is very similar to the work of Schalkwyk et. al [14].

2.3. FST Files

The basic FST type uses a simple ASCII format for file input/output. It was a design goal to keep this format as straightforward as possible so that FST files could be easily created from other tools (e.g., Perl scripts). There are binary formats as well to support more rapid file I/O.

Where command-line tools expect a file specification, they also accept "`-`" for standard input or output, supporting Unix-style pipes, as well as expressions invoking run-time FST operations. For example, the command

```
fst_compose a.fst 'det(re(<b.fst>))' -
```

computes the FST composition of two FSTs, the first contained in the file `a.fst`, and the second from the on-the-fly epsilon removal and determinization of the FST in the file `b.fst`, and writes the result to standard output. While this is often convenient, to have access to the full capabilities of the library you have to use the C++ API.

2.4. C++ API

The toolkit is implemented within a C++ class library, making all functionality available from command-line executables also available for use within applications. Central to this C++ API is a set of functions for manipulating FSTs, a common method interface for accessing and modifying state and transitions properties, and the use of a semiring interface for all weight manipulation. Transitions leaving from or arriving at a given state can optionally be filtered to match a given input or output symbol and/or sorted by input or output symbol, and an iterator is provided for traversing the result of such a selection. See Figure 2 for some sample code demonstrating use of the API.

2.5. Functionality

The toolkit provides a variety of operations for FST construction, composition, optimization, searching, pruning, and weight training, all available via command-line tools or the C++ API.

2.5.1. Weight Semirings

The manipulation of all weights within the toolkit is performed through semirings. This allows the user to choose how weights are combined in parallel and in series during composition and optimization. The primary semirings available are the real semiring $(+, \times, 0, 1)$, representative of probabilities (add in parallel, multiply in series), and the tropical semiring $(\min, +, \infty, 0)$, represen-

```
#include <fst/FST.h>

FSTGeneric swap_input_output(FSTGeneric in)
{
  FSTGeneric out = FSTBasic();
  out->add_states(in->n_states()); // not required
  out->set_initial(in->initial());
  for (int p = 0; p < in->n_states(); p++) {
    out->set_final(p, in->final_weight(p));
    for (FSTArcIter a = in->arcs_out(p); a.valid(); a++)
      out->add_arc(p, a->next,
                   a->output, a->input, a->weight);
  }
  return out;
}
```

Figure 2: Sample C++ code using the API to create copy of FST with input and output labels interchanged.

[1] Statically representing an n-gram as a deterministic FST can be very space inefficient due to every state or n-gram history requiring a transition for every word in the vocabulary.

tative of $-\log$ probabilities and Viterbi-style scoring (take best in parallel, add in series), the two most commonly used in speech and language processing applications. Other real-valued weight semirings could be implemented within the API.

2.5.2. Construction

The familiar regular operations $X \cup Y$ (union), XY (concatenation), and X^* and X^+ (Kleene closure) are available for constructing FSTs and FSAs. All of these operations support lazy evaluation. Other operations include reversing an FST and projecting its transition input/output labels (e.g., swapping labels or setting inputs or outputs to ϵ).

2.5.3. Composition

The FST composition operation $X \circ Y = Z$ is provided and supports lazy evaluation. Composition allows combining two transductions X and Y (e.g., X from phones to phonemes and Y from phonemes to words) into a single FST Z that combines both transductions (e.g., from phones to words). Composition is a basic building block when constructing models from different levels of representation. Applying composition to FSAs (i.e., FSTs with equal input and output labels) computes their intersection. An intermediate ϵ filter is used to eliminate redundant paths [18].

2.5.4. Optimization

Various operations are provided for optimizing FSTs. These operations are identity transformations; that is they do not change the transduction represented by an FST or the regular language accepted by an FSA. In general, the basic steps to compute a deterministic minimal FST or FSA are trimming unused states, ϵ removal, determinization, pushing weights and output labels, and finally minimization.

Trimming removes states that are not part of complete paths from the intial state to a final state. ϵ-removal removes ϵ transitions, potentially shifting output symbols in the process. Determinization combines transitions leaving a state with a common input or input/output symbol, and can cause output symbols to be delayed in order to share inputs. Pushing weights and output symbols away from final states aids in the identification of equivalent states in subsequent minimization. Finally, minimization collapses equivalent states.

The epsilon removal, determinization, and minimization algorithms are essentially those of Mohri [2]. Note that not all FSTs or FSAs can be determinized. For cyclic weighted FSTs and FSAs, weights in cycles can result in nontermination of the determization algorithm. Similarly, for FSTs, ambiguity of input/output mappings within cycles can also cause nontermination.

For example, if a lexicon FST mapping input phonemes to output words contains homophones, a given input phoneme sequence may yield more than one output word sequence (i.e., the mapping is ambiguous). We have augmented the determinization algorithm to force any pending ambiguity to be flushed out when a transition with a special disambiguation input label "#" is encountered. We then insert "#" input labels between words within a cyclic lexicon FST, enabling determinization to output such ambiguity rather than delaying it indefinitely.[2]

2.5.5. Searching

The toolkit supports various searches on FSTs including best path, N-best paths, and graph pruning. One tool uses an A^* search to generate N-best paths or performs pruning by outputting a graph (FST) containing all transitions and states within a score threshold of the best path. Another tool computes the best path through a composition $A \circ B$, where A's states are topologically sortable, and is similar to Viterbi decoding commonly used in speech recognition, including the availability of beam pruning.

2.5.6. EM Weight Training

We have implemented EM training on FSA and FST weights within the toolkit. Weighted FSAs can be trained from example sequences, and weighted FSTs can be trained from example sequence pairs (i.e., input and output sides), even in the middle of a cascade of FSTs [19]. The FST weights are initially trained to represent a joint probability distribution $P(I, O)$ between input sequence I and output sequence O, but it may be possible to transform the FST into one representing $P(I|O)$ or $P(O|I)$.

3. Toolkit Application

The first application of this toolkit was to the MIT SUMMIT speech recognition system [6], in which we factored linguistic constraints into the cascade of FSTs $R = C \circ P \circ L \circ G$, where G is a grammar or n-gram language model, L a lexicon constructed from phonemic regular expressions, P phonological transformations compiled from explicit phonological rules [7], and C a set of context-dependent model label transformations. The recognition search sees only the single FST R going from context-dependent model labels through word sequences, even though it may be statically compiled and optimized or parts of it expanded dynamically through lazy evaluation. Our recognizer is much more flexible than it was previously while at the same time more efficient through the use of FST optimization.

The FST framework has enabled the straightforward incorporation of complex out-of-vocabulary (OOV) word models that contain subword n-grams. These OOV models can then be incorporated within the lexicon L and subsequently the recognition FST R through straightforward use of FST operations [9]. An FST-based recognizer also enabled inserting an FST representing weighted phonetic confusions due to non-native speech into the recognition cascade R [10]. Tang et. al [11] explored the use of generalized manner and place lingustic features and how they could be integrated within the recognition cascade.

We have found that for very large vocabularies and very large trigram language models that we could achieve better time/accuracy tradeoffs using a factorization of language models [12]. The technique involves statically compiling and optimizing the recognition FST using a greatly pruned version of the n-gram and then applying the "rest" of the n-gram on the fly during recognition. The net effect is very similar to that of lexical trees with language model lookahead [20]. The use of the FST framework avoided having to use special-purpose search code. Willett and Katagiri [21] similarly saw time/accuracy gains when language model weights were smoothed during FST determinization, using our toolkit.

[2] An equivalent techique is to use a word-specific label ϵ_w at the end of each word w, determinize, and then change all ϵ_w to "#" or ϵ as desired.

Hazen et. al [13] used $R = R_1 \cup \cdots \cup R_n$ to combine recognizers R_i for different domains or languages into a single conversational system, allowing the different R_i to compete during the recognition search. The effect was improved efficiency vs. running separate recognizers in parallel, as non-competitive R_i could often be pruned early in the search.

FSTs played an integral role in configuring a "speak-and-spell" recognizer in which a new vocabulary entry could be spoken and spelled in the same utterance. The phonetic output of a general OOV model, through the use of a sound-to-letter FST, would be used to constrain the letter recognition elsewhere in the same utterance [15].

4. Conclusion and Availability

We have described a finite-state transducer toolkit, its design goals and functionality, and brief references to a subset of the research it has aided. It is hoped that by making the toolkit publically available as open source others will find it valuable to their research and will contribute to and improve its functionality.

See `http://www.sls.csail.mit.edu/ilh/fst/` for downloading the toolkit. It is known to compile with GNU `g++` 2.95–3.3 under Linux, but it is hoped that others will help port it to other platforms and/or compilers as needed.

5. Acknowledgements

Thanks to Mehryar Mohri for helping me to get started with this toolkit by patiently explaining his determinization algorithm. Finally, thanks to Victor Zue and Jim Glass for providing me with the time and resources to implement and maintain this toolkit within the Spoken Language Systems group at MIT.

6. References

[1] E. Roche and Y. Schabes, Eds., *Finite-State Language Processing*, The MIT Press, Cambridge, MA, 1997.

[2] M. Mohri, "Finite-state transducers in language and speech processing," *Computational Linguistics*, vol. 23, no. 2, pp. 269–312, June 1997.

[3] M. Morhi, F. C. N. Pereira, and M. D. Riley, "The AT&T FSM library: Finite-state machine library," http://www.research.att.com/sw/tools/fsm/, 2003.

[4] C. Allauzen, M. Morhi, and B. Roark, "The AT&T GRM library: Grammar library," http://www.research.att.com/sw/tools/grm/, 2003.

[5] F. Pereira, M. Riley, and R. Sproat, "Weighted rational transductions and their application to human language processing," in *Proc. of the ARPA Human Language Technology Workshop*, Plainsboro, NJ, Mar. 1994, pp. 262–267.

[6] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pao, T. J. Hazen, and L. Hetherington, "Jupiter: A telephone-based conversational interface for weather information," *IEEE Trans. Speech and Audio Processing*, vol. 8, no. 1, pp. 85–96, Jan. 2000.

[7] I. L. Hetherington, "An efficient implementation of phonological rules using finite-state transducers," in *Proc. of Eurospeech*, Aalborg, Sept. 2001, pp. 1599–1602.

[8] T. J. Hazen, L. Hetherington, H. Shu, and K. Livescu, "Pronunciation modeling using a finite-state transducer representation," in *Proc. of the ISCA Workshop on Pronunciation Modeling and Lexicon Adapation*, Estes Park, Sept. 2002, pp. 99–105.

[9] I. Bazzi and J. Glass, "Modeling out-of-vocabulary words for robust speech recognition," in *Proc. Intl. Conf. on Spoken Lang. Processing*, Beijing, Oct. 2000, pp. 401–404.

[10] K. Livescu and J. Glass, "Lexical modeling of non-native speech for automatic speech recognition," in *Proc. Intl. Conf. on Acoustics, Speech, and Signal Processing*, Instanbul, June 2000, pp. 1842–1845.

[11] M. Tang, S. Seneff, and V. Zue, "Modeling linguistic features in speech recognition," in *Proceedings of Eurospeech*, Geneva, Sept. 2003, pp. 2585–2588.

[12] H. Dolfing and L. Hetherington, "Incremental language models for speech recognition using finite-state transducers," in *Proc. IEEE Automatic Speech Recognition and Understanding Workshop*, Madonna de Campiglio, Dec. 2001.

[13] T. J. Hazen, I. L. Hetherington, and A. Park, "FST-based recognition techniques for multi-lingual and multi-domain spontaneous speech," in *Proc. of Eurospeech*, Aalborg, Sept. 2001, pp. 1591–1593.

[14] J. Schalkwyk, L. Hetherington, and E. Story, "Speech recognition with dynamic grammars using finite-state transducers," in *Proceedings of Eurospeech*, Geneva, Sept. 2003, pp. 1969–1972.

[15] G. Chung, S. Seneff, and C. Wang, "Automatic acquisition of names using speak and spell mode in spoken dialogue systems," in *Proc. HLT-NAACL 2003*, Edmonton, May 2003, pp. 32–39.

[16] J. Eisner, "Parameter estimation for probabilistic finite-state transducers," in *Proc. Assoc. of Computational Linguistics*, Philadelphia, July 2002.

[17] Sun Microsystems, "Java speech grammar format specification," http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/, 1998.

[18] F. Pereira and M. Riley, "Speech recognition by composition of weighted finite automata," in *Finite-State Language Processing*, E. Roche and Y. Schabes, Eds. The MIT Press, Cambridge, MA, 1997, also available as http://xxx.lanl.gov/pdf/cmp-lg/9603001.

[19] H. Shu and L. Hetherington, "EM training of finite-state transducers and its application to pronunciation modeling," in *Proc. of the Intl. Conf. on Speech and Lang. Processing*, Denver, Sept. 2002, pp. 1293–1296.

[20] S. Ortmanns, H. Ney, and A. Eiden, "Language-model look-ahead for large vocabulary speech recognition," in *Proc. Intl. Conf. on Spoken Lang. Processing*, Philadelphia, Oct. 1996, pp. 2095–2098.

[21] D. Willett and S. Katagiri, "Recent advances in efficient decoding combining on-line transducer composition and smoothed language model incorporation," in *Proc. Intl. Conf. on Acoustics, Speech, and Signal Processing*, Orlando, May 2002, pp. 713–716.