Pynini: A Python library for weighted finite-state grammar compilation

Kyle Gorman
Google, Inc.
111 8th Avenue, New York, NY 10011

Abstract

We present Pynini, an open-source library for the compilation of weighted finite-state transducers (WFSTs) and pushdown transducers (PDTs) from strings, context-dependent rewrite rules, and recursive transition networks. Pynini uses the OpenFst library for encoding, modifying, and applying WFSTs and PDTs. We describe the design of this library and the algorithms and interfaces used for compilation, optimization, and application, and provide two illustrations of the library in use.

1 Introduction

Weighted finite-state transducers (WFSTs) are widely used in speech and language applications. The hypothesis space for tasks like automatic speech recognition (ASR) and optical character recognition can be be represented as a compact, efficiently searchable cascade of WFSTs (Mohri et al., 2002). Manually-generated grammatical resources such as pronunciation lexicons and phonological rules are also naturally represented as finite-state transducers. One advantage of an end-to-end WFST formulation in complex natural language problems like speech recognition is that WFSTs make it easy to combine "big data" statistical components like language models and "small batch" resources like hand-written grammars. For instance, WFST ASR models may be augmented with a series of weighted rewrite rules modeling pronunciation variation (Hazen et al., 2005) to reduce word error rate, or composed with an WFST which restores punctuation and capitalization to improve transcript readability (Shugrina, 2010).

2 Existing WFST libraries

There are a number of publicly available WFST libraries, most of them open-source. Roughly speaking, these libraries can be divided into three groups. First, libraries like Carmel (Knight and Graehl, 1998) and OpenFst (Allauzen et al., 2007) provide efficient implementations of key algorithms for combining, optimizing, and searching WFSTs. However, such libraries provide little support for users who wish to compile a lexicon or a list of grammatical rules into a WFST, so that even basic grammar-building tasks may be a challenge. A second group of libraries, including HFST (Lindén et al., 2013), Lextools (Sproat, 1995), Kleene (Beesley, 2012), and Thrax (Roark et al., 2012), focus on grammar compilation operations and rely upon the aforementioned libraries for core WFST algorithms. Finally, a third group of libraries, including Foma (Hulden, 2009) and XFST (Beesley and Karttunen, 2003), provide algorithms and compilation routines for finite-state transducers, though neither supports weighted transducers required by many real-world applications.

2.1 DSL-and-compiler interfaces

Table 1 summarizes key features of eight libraries which provide some form of grammar compilation support. Among these libraries, the most common interface is a compiler for a library-specific declarative, domain-specific language (DSL). There are some advantages for using a domain-specific language here; for instance, they allow for remarkably terse grammars. However, in our experience, DSL-and-compiler interfaces may also be a substantial impediment to rapid development.

First, these domain-specific languages all make extensive the use of novel operators; e.g., several employ `$` as a unary prefix operator denoting the containment of an WFST. Any new language

Proceedings of the ACL Workshop on Statistical NLP and Weighted Automata, pages 75–80,
Berlin, Germany, August 12, 2016.

	XFST	Lextools	Carmel	Foma	Kleene	Thrax	HFST	Pynini
Gratis	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Libre	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Weighted transducers	No	Yes	No	No	Yes	Yes	Yes	Yes
CDRR compilation	No	Yes	No	Yes	Yes	Yes	Yes	Yes
PDT compilation	No	No	No	No	No	Yes	No	Yes
Python bindings	No	No	No	Yes	No	No	Yes	Yes

Table 1: Key features of eight publicly available libraries for compilation of finite-state grammars.

comes with a learning curve, but the use of novel operators (with unfamiliar precedence) may make the curve particularly steep.

More generally, these DSLs lack the libraries, programming constructs, and tooling present in popular domain-general programming languages. A user who wishes to compile a pronunciation lexicon from data stored in an XML file, for example, has little choice but to write a script in some domain-general programming language to rewrite the XML data into a library-specific format supported by the compiler.

2.2 Introducing Pynini

We propose an alternative approach to this problem. We do not introduce yet another competing DSL-and-compiler standard, nor do we attempt to expose a DSL to another programming language, as do Foma and HFST, both of which provide basic Python bindings. Rather we make WFST algorithms and compilation routines "first-class citizens" of an existing domain-general multi-paradigm programming language. The result is a Python extension called *Pynini* (named in honor of Pāṇini, the renown Sanskrit grammarian). Pynini takes advantage of Python's substantial standard library, expressive syntax, and tools for debugging, linting, and interactive development.

Pynini is distributed freely as part of the OpenGrm toolkit under the Apache 2.0 license.

2.3 Outline

In what follows, we describe the design and implementation of the Pynini library, focusing on compilation and optimization routines (§3). We then present two examples illustrating the use of the library (§4–5), and then conclude with future directions for this project.

3 Design of the library

3.1 Software architecture

Pynini is based on OpenFst (Allauzen et al., 2007), an efficient weighted finite-state transducer library written in C++ 11.[1] At the lowest level, OpenFst provides a set of classes (representing WFSTs) and functions (representing WFST algorithms) templated on the semiring of the input FST(s).

A second layer, the OpenFst scripting API, uses virtual dispatch, function registration, and dynamic loading of shared objects to provide a common interface shared by FSTs of different semirings. OpenFst also includes a Python extension module, `pywrapfst`, which exposes the entire scripting API with little additional "syntactic sugar".[2]

Pynini extends this architecture at all three levels. It is implemented with C++ template functions (some of which are shared with Thrax), a semiring-independent scripting interface, and a Python module which extends `pywrapfst`.

3.2 Compilation

Pynini provides a Python class called `Fst`, which represents a mutable WFST with a user-specified semiring (by default, the tropical semiring). The `epsilon_machine` function creates a one-state FST that accepts only the empty string. The `acceptor` function compiles a string into a (deterministic, epsilon-free) FSA. The user may specify how the arcs of the resulting FSA are to be labeled; by default, each arc in the FSA corresponds to a byte in the input string, but the string may also be interpreted as a UTF-8-encoded string—in which case each arc label corresponds to a codepoint in the Unicode Basic Multilingual Plane—or according to a user-provided symbol table. As in

[1] At time of writing Pynini depends on OpenFst 1.5.3.

[2] For more information on these APIs, visit `http://openfst.org`.

Thrax, a string enclosed in square brackets are interpreted as a single *generated* symbol rather than as sequences of bytes or codepoints.

Nearly all Pynini functions permit a string to be passed in place of an `Fst`, in which case `acceptor` then is used to compile the string. One such function is `transducer`, which takes two FSA (or string) arguments and compiles a transducer that represents their cross-product. The union of many such string pairs can be compiled using the `string_file` and `string_map` functions. The former reads pairs of strings from a tab-separated values (TSV) file, whereas the latter extracts string pairs from a Python dictionary, list, or tuple. Both functions produce a compact prefix tree representation of a map (or multimap) such as a pronunciation dictionary.

The `cdrewrite` function performs compilation of context-dependent rewrite rules (CDRRs) using an algorithm due to Mohri and Sproat (1996). The `replace` function compiles FSTs from recursive transition networks (RTNs; Woods 1970). An RTN is specified as a single *root* FST followed by a set of one or more *replacement* FSTs, each of which is passed as a keyword argument where the keyword represents the replacement's corresponding *nonterminal*. If an RTN contains any recursive replacements—i.e., if any FST in the RTN contains its nonterminal either directly or indirectly—it lacks a finite expansion and therefore cannot be compiled into an FST, but it can be compiled as a pushdown transducer (PDTs; Allauzen and Riley 2012) using the `pdt_replace` function.

Major functions for constructing FSTs are shown in Figure 1.

3.3 Visualization

Pynini provides several techniques for visualizing FSTs. Invoking Python's `print` statement on an `Fst` prints the FST's arc list, and the `draw` method renders an FST as a GraphViz[3] image. It is also possible to iterate over the states, arcs, and paths of an FST using the `states`, `arcs`, and `paths` methods, respectively.

3.4 Algorithms

All of the major WFST algorithms supported by OpenFst can be invoked via module-level functions which return an `Fst` instance (or raise an exception in the case of run-time failure). A subset of OpenFst WFST algorithms, including closure, inversion, and projection, operate destructively (i.e., they mutate their input in-place), and can also be invoked by calling the appropriate instance method on an `Fst` object. All destructive operations return their mutated input so as to allow chaining.

3.5 Optimization

Core FST operations such as composition, concatenation, and union tend to introduce many redundant arcs and states, and therefore it is desirable (and in some cases necessary) to optimize FSTs during grammar compilation. OpenFst provides four algorithms for this task: epsilon-removal, arc-sum mapping (which merges identically-labeled multiarcs), determinization, and minimization. However, there are complex restrictions on the application of these algorithms, making manual optimization something of a challenge.

Pynini provides an instance method `optimize` which applies these four procedures, subject to these algorithmic restrictions and the properties of the input FST. For instance, the minimization algorithm is guaranteed to find a minimal FST only in the case that the input is deterministic, and while any acyclic FST over a *zero-sum-free* semiring is determinizable, this is not necessarily true of FSTs with weighted cycles (Mohri, 2009). Therefore, if an FST is not known to be deterministic and weighted-cycles-free, the optimization routine performs both determinization and minizimation on the FST while it is encoded as if it were an unweighted acceptor (Allauzen et al., 2004).

4 Sample grammar: Finnish harmony

Koskenniemi (1983) provides a number of manually-compiled FSTs modeling Finnish morphophonological patterns. One of these concerns the well-known pattern of Finnish vowel harmony. Many Finnish suffixes have two allomorphs differing only in the backness specification of their vowel. For example, the adessive suffix is usually realized as *-llä* [lːæː] except when the preceding stem contains one of *u*, *o*, and *a* and there is no intervening *y*, *ö*, or *ä*; in this case, it is *-lla* [lːɑː]. For example, *käde* 'hand' has an adessive *kädellä*, whereas *vero* 'tax' forms the adessive *verolla* because the nearest stem vowel is *o* (Ringen and Heinämäki, 1999). Figure 2 provides a Pynini function (`make_adessive`) which generates the appropriate form of a noun stem.

[3] `http://graphviz.org`

```
a = epsilon_machine()
b = acceptor("Red Leicester")
c = transducer("Tilsit", "Never at the end of the week, sir")
d = string_map({"Stilton": "Sorry", "Gruyère": "No"})
e = replace("[COLOR] [CHEESE]",
            COLOR=union("Blue", "Red", "White"),
            CHEESE=union("Leicester", "Stilton", "Vinney", "Windsor"))
```

Figure 1: Examples of various FST construction functions in Pynini.

It first concatenates the stem with an abstract representation of the suffix, and then composes the result with a context-dependent rewrite rule `adessive_harmony`.

5 Sample application: T9 decoding

T9 (short for "Text on 9 keys"; Grover et al. 1998) is a patented predictive text entry system. In T9, each character in the "plaintext" alphabet is assigned to one of the 9 digit keys (0 is usually reserved to represent space) of the traditional 3x4 touch-tone phone grid. For instance, the message *GO HOME* is entered as *4604663*. Since the resulting "ciphertext" may be highly ambiguous—this sequence could also be read as *GO HOOD*, not to mention many nonsensical expressions—a hybrid language model/lexicon is used for decoding.

Figure 3 shows T9 encoding and decoding in Pynini. The first line reads in an language model represented as a weighted finite-state acceptor.[4] The second line reads in the encoder table from a `string_file`, a text file in which each line contains an alphabetic character and its corresponding digit key. By computing the concatenative `closure` of this map, we obtain an FST (`T9_ENCODER`) which can encode arbitrarily-long plaintext strings.

The `k_best` function first applies a ciphertext string (a bytestring of digits) to the inverted encoder FST (`T9_DECODER`) via composition and output-projection. This creates an intermediate lattice of all possible plaintexts consistent with the T9 ciphertext. This is then scored by—that is, composed with—the character LM. Finally, we generate the k shortest paths (i.e., the k highest-probability plaintexts) in the lattice. In the example given in the figure, the shortest path here is in fact identical to the plaintext.

[4]The language model used here is an 8-gram character language model with Witten-Bell smoothing, trained on the Wall St. Journal portion of the Penn Treebank using OpenGrm-NGram (Roark et al., 2012). This is somewhat different than the language models used in mobile phone T9 systems.

6 Conclusions

We have described and illustrated the design and implementation of an expressive and extensible open-source library for weighted finite-state grammar compilation. In future work, we hope to exploit this library to improve the user experience for developers of grammars and of WFST applications more generally.

Acknowledgements

Thanks to all those who provided valuable input during the design of Pynini. Richard Sproat originally suggested to me the idea of developing a Python based finite-state grammar library, discussed many design issues, and contributed the path iteration algorithm. Cyril Allauzen, Steven Bedrick, Myroslava Dzikovska, Mark Epstein, Toby Hawker, Michael Riley, and Ke Wu also provided a lot of useful input. To obtain Pynini, visit `http://pynini.opengrm.org`.

References

Cyril Allauzen and Michael Riley. 2012. A pushdown transducer extension for the OpenFst library. In *CIAA*, pages 66–77.

Cyril Allauzen, Mehryar Mohri, Michael Riley, and Brian Roark. 2004. A generalized construction of integrated speech recognition transducers. In *ICASSP*, pages 761–764.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *CIAA*, pages 11–23.

Kenneth R. Beesley and Lari Karttunen. 2003. *Finite state morphology*. CSLI, Stanford, CA.

Kenneth R. Beesley. 2012. Kleene, a free and open-source language for finite-state programming. In *10th International Workshop on Finite State Methods and Natural Language Processing*, pages 50–54.

```
back_vowel = u("u", "o", "a")
neutral_vowel = u("i", "e")
front_vowel = u("y", "ö", "ä")
vowel = u(back_vowel, neutral_vowel, front_vowel)
archiphoneme = u("A", "I", "E", "O", "U")
consonant = u("b", "c", "d", "f", "g", "h", "j", "k", "l", "m", "n", "p", "q",
              "r", "s", "t", "v", "w", "x", "z")
sigma_star = u(vowel, consonant, archiphoneme).closure()
adessive = "llA"
intervener = u(consonant, neutral_vowel).closure()
adessive_harmony = (cdrewrite(t("A", "a"), back_vowel + intervener, "",
                              sigma_star) *
                    cdrewrite(t("A", "ä"), "", "", sigma_star)).optimize()

def make_adessive(stem):
  return ((stem + adessive) * ur).stringify()
```

Figure 2: Finnish adessive suffix harmony, implemented with context-dependent rewrite rules.

```
LM = Fst.read("charlm.fst")
T9_ENCODER = string_file("t9.tsv").closure()
T9_DECODER = invert(T9_ENCODER)

def encode_string(plaintext):
  return (plaintext * T9_ENCODER).stringify()

def k_best(ciphertext, k):
  lattice = (ciphertext * T9_DECODER).project(True) * LM
  return shortestpath(lattice, nshortest=k, unique=True).paths()

pt = "THE SINGLE MOST POPULAR CHEESE IN THE WORLD"
ct = encode_string(pt)
for (_, opath, _) in k_best(ct, 5):
  print opath
```

Figure 3: T9 encoding and decoding with a character LM.

Dale L. Grover, Martin T. King, and Clifford A. Kushler. 1998. Reduced keyboard disambiguating computer. US Patent 5,818,437.

Timothy J. Hazen, I. Lee Hetherington, Han Shu, and Karen Livescu. 2005. Pronunciation modeling using a finite-state transducer representation. *Speech Communication*, 46(2):189–203.

Mans Hulden. 2009. Foma: A finite-state compiler and library. In *EACL*, pages 29–32.

Kevin Knight and Jonathan Graehl. 1998. Machine transliteration. *Computational Linguistics*, 24(4):599–612.

Kimmo Koskenniemi. 1983. *Two-level morphology: A general computational model for word-form recognition and production*. Ph.D. thesis, University of Helsinki.

Krister Lindén, Erik Axelson, Senka Drobac, Sam Hardwick, Juha Kuokkala, Jyrki Niemi, Tommi A. Pirinen, and Miikka Silfverberg. 2013. HFST: A system for creating NLP tools. In *SCFM*, pages 53–71.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *ACL*, pages 231–238.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16(1):69–88.

Mehryar Mohri. 2009. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of weighted automata*, pages 213–254. Springer, New York.

Catherine O. Ringen and Orvokki Heinämäki. 1999. Variation in Finnish vowel harmony: An OT account. *Natural Language and Linguistic Theory*, 17(2):303–337.

Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *ACL*, pages 61–66.

Maria Shugrina. 2010. Formatting time-aligned ASR transcripts for readability. In *NAACL*, pages 198–206.

Richard Sproat. 1995. Lextools: Tools for finite-state linguistic analysis. Technical Report 11522-951108-10TM, Bell Laboratories.

William A. Woods. 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.