The OpenGrm open-source finite-state grammar software libraries

Brian Roark[†] **Richard Sproat**[†°] **Cyril Allauzen**[°] **Michael Riley**[°] **Jeffrey Sorensen**[°] **& Terry Tai**[°]
[†]Oregon Health & Science University, Portland, Oregon [°]Google, Inc., New York

Abstract

In this paper, we present a new collection of open-source software libraries that provides command line binary utilities and library classes and functions for compiling regular expression and context-sensitive rewrite rules into finite-state transducers, and for n-gram language modeling. The OpenGrm libraries use the OpenFst library to provide an efficient encoding of grammars and general algorithms for building, modifying and applying models.

1 Introduction

The OpenGrm libraries[1] are a (growing) collection of open-source software libraries for building and applying various kinds of formal grammars. The C++ libraries use the OpenFst library[2] for the underlying finite-state representation, which allows for easy inspection of the resulting grammars and models, as well as straightforward combination with other finite-state transducers. Like OpenFst, there are easy-to-use command line binaries for frequently used operations, as well as a C++ library interface, allowing library users to create their own algorithms from the basic classes and functions provided.

The libraries can be used for a range of common string processing tasks, such as text normalization, as well as for building and using large statistical models for applications like speech recognition. In the rest of the paper, we will present each of the two libraries, starting with the Thrax grammar compiler and then the NGram library. First, though, we will briefly present some preliminary (informal) background on weighted finite-state transducers (WFST), just as needed for this paper.

[1]`http://www.opengrm.org/`
[2]`http://www.openfst.org/`

2 Informal WFST preliminaries

A weighted finite-state transducer consists of a set of states and transitions between states. There is an initial state and a subset of states are final. Each transition is labeled with an input symbol from an input alphabet; an output symbol from an output alphabet; an origin state; a destination state; and a weight. Each final state has an associated final weight. A path in the WFST is a sequence of transitions where each transition's destination state is the next transition's origin state. A valid path through the WFST is a path where the origin state of the first transition is an initial state, and the the last transition is to a final state. Weights combine along the path according to the semiring of the WFST.

If every transition in the transducer has the same input and output symbol, then the WFST represents a weighted finite-state automaton. In the OpenFst library, there are a small number of special symbols that can be used. The ϵ symbol represents the empty string, which allows the transition to be traversed without consuming any symbol. The ϕ (or failure) symbol on a transition also allows it to be traversed without consuming any symbol, but it differs from ϵ in only allowing traversal if the symbol being matched does not label any other transition leaving the same state, i.e., it encodes the semantics of *otherwise*, which is useful for language models. For a more detailed presentation of WFSTs, see Allauzen et al. (2007).

3 The Thrax Grammar Compiler

The Thrax grammar compiler[3] compiles grammars that consist of regular expressions, and context-dependent rewrite rules, into FST archives (fars) of weighted finite state transducers. Grammars may

[3]The compiler is named after Dionysius Thrax (170–90BCE), the reputed first Greek grammarian.

Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics, pages 61–66,
Jeju, Republic of Korea, 8-14 July 2012. ©2012 Association for Computational Linguistics

be split over multiple files and *imported* into other grammars. Strings in the rules may be parsed in one of three different ways: as a sequence of bytes (the default), as utf8 encodings, or according to a user-provided symbol table. With the `--save_symbols` flag, the transducers can be saved out into fars with appropriate symbol tables.

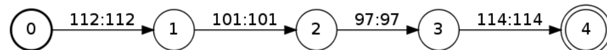
The Thrax libraries provide full support for different weight (semiring) classes. The command-line flag `--semiring` allows one to set the semiring, currently to one of: tropical (default), log or log64 semirings.

3.1 General Description

Thrax revolves around rules which, typically, construct an FST based on a given input. In the simplest case, we can just provide a string that represents a (trivial) transducer and name it using the assignment operator:

```
pear = "pear";
```

In this example, we have an FST consisting of the characters “p”, “e”, “a”, and “r” in a chain, assigned to the identifier `pear`:



This identifier can be used later in order to build further FSTs, using built-in operators or using custom functions:

```
kiwi = "kiwi";
fruits = pear | kiwi; # union
```

In Thrax, string FSTs are enclosed by double-quotes (`"`) whereas simple strings (often used as pathnames for functions) are enclosed in single-quotes (`'`).

Thrax provides a set of built-in functions that aid in the construction of more complex expressions. We have already seen the disjunction “|” in the previous example. Other standard regular operations are *expr**, *expr+*, *expr?* and *expr{m,n}*, the latter repeating *expr* between *m* and *n* times, inclusive. Composition is notated with “@” so that *expr1 @ expr2* denotes the composition of *expr1* and *expr2*. Rewriting is denoted with “:” where *expr1 : expr2* rewrites strings that match *expr1* into *expr2*. Weights can be added to expressions using the notation “<>”: thus, *expr<2.4>* adds weight *2.4* to *expr*. Various operations on FSTs are also provided by built-in functions, including `Determinize`, `Minimize`, `Optimize` and `Invert`, among many others.

3.2 Detailed Description

A Thrax grammar consists of a set of one or more source files, each of which must have the extension `.grm`. The compiler compiles each source file to a single *FST archive* with the extension `.far`. Each grammar file has sections: Imports and Body, each of which is optional. The body section can include statements interleaved with functions, as specified below. Comments begin with a single pound sign (#) and last until the next newline.

3.2.1 Imports

The Thrax compiler compiles source files (with the extension `.grm`) into *FST archive* files (with the extension `.far`). FST archives are an OpenFst storage format for a series of one or more FSTs. The FST archive and the original source file then form a pair which can be imported into other source files, allowing a Python-esque include system that is hopefully familiar to many. Instead of working with a monolithic file, Thrax allows for a modular construction of the final rule set as well as sharing of common elements across projects.

3.2.2 Functions

Thrax has extensive support for functions that can greatly augment the capabilities of the language. Functions in Thrax can be specified in two ways. The first is inline via the *func* keyword within `grm` files. These functions can take any number of input arguments and must return a single result (usually an FST) to the caller via the *return* keyword:

```
func DumbPluralize[fst] {
  # Concatenate with "s"...
  result = fst "s";
  # ...and then return to caller.
  return result;
}
```

Alternatively, functions can be written C++ and added to the language. Regardless of the function implementation method (inline in Thrax or subclassed in C++), functions are integrated into the Thrax environment and can be called directly by using the function name and providing the necessary arguments. Thus, assuming someone has written a function called `NetworkPluralize` that retrieves the plural of a word from some website, one could write a grammar fragment as follows:

```
apple = "apple";
plural_apple = DumbPluralize[apple];

plural_tomato = NetworkPluralize[
    "tomato",
    'http://server:port/...'];
```

3.2.3 Statements

Functions can be interleaved with grammar statements that generate the FSTs that are exported to the FST archive as output. Each statement consists of an assignment terminating with a semicolon:

```
foo = "abc";
export bar = foo | "xyz";
```

Statements preceded with the *export* keyword will be written to the final output archive. Statements lacking this keyword define temporaries that be used later, but are themselves not output.

The basic elements of any grammar are string FSTs, which, as mentioned earlier, are defined by text enclosed by double quotes ("), in contrast to raw strings, which are enclosed by single quotes ('). String FSTs can be parsed in one of three ways, which are denoted using a dot (.) followed by either *byte*, *utf8*, or an identifier holding a symbol table. Note that within strings, the backslash character (\) is used to escape the next character. Of particular note, '\n' translates into a newline, '\r' into a line feed, and '\t' into the tab character. Literal left and right square brackets also need escaping, as they are used to generate symbols (see below). All other characters following the backslash are uninterpreted, so that we can use \" and \' to insert an actual quote (double) quote symbol instead of terminating the string.

Strings, by default, are interpreted as sequences of bytes, each transition of the resulting FST corresponding to a single 1-byte character of the input. This can be specified either by leaving off the parse mode (`"abc"`) or by explicitly using the byte mode (`"abc".byte`). The second way is to use UTF8 parsing by using the special keyword, e.g.:

```
kimchi = "김치".utf8;
```

Finally, we can load a symbol table and split the string using the `fst_field_separator` flag (found in `fst/src/lib/symbol-table.cc`) and then perform symbol table lookups. Symbol tables can be loaded using the SymbolTable built-in function:

```
arctic_symbol_table =
  SymbolTable['/path/to/bears.symtab'];
pb = "polar bear".arctic_symbol_table;
```

One can also create temporary symbols on the fly by enclosing a symbol name inside brackets within an FST string. All of the text inside the brackets will be taken to be part of the symbol name, and future encounters of the same symbol name will map to the same label. By default, labels use "Private Use Area B" of the unicode table (0x100000 - 0x10FFFD), except that the last two code points 0x10FFFC and 0x10FFFD are reserved for the "[BOS]" and "[EOS]" tags discussed below.

```
cross_pos = "cross" ("" : "_[s_noun]");
pluralize_nouns = "_[s_noun]" : "es";
```

3.3 Standard Library Functions and Operations

Built-in functions are provided that operate on FSTs and perform most of the operations that are available in the OpenFst library. These include: closure, concatenation, difference, composition and union. In most cases the notation of these functions follows standard conventions. Thus, for example, for closure, the following syntax applies: fst* (accepts fst 0 or more times); fst+ (accepts fst 1 or more times); fst? (accepts fst 0 or 1 times) fst{x,y} (accepts fst at least x but no more than y times).

The operator "@" is used for composition: `a @ b` denotes `a` composed with `b`. A ":" is used to denote *rewrite*, where `a : b` denotes a transducer that deletes `a` and inserts `b`. Most functions can also be expressed using functional notation:

```
b = Rewrite["abc", "def"];
```

The delimiters < and > add a weight to an expression in the chosen semiring: `a<3>` adds the weight 3 (in the tropical semiring by default) to `a`.

Functions lacking operators (hence only called by function name) include: `ArcSort`, `Connect`, `Determinize`, `RmEpsilon`, `Minimize`, `Optimize`, `Invert`, `Project` and `Reverse`. Most of these call the obvious underlying OpenFst function.

One function in particular, `CDRewrite` is worth further discussion. This function takes a transducer and two context acceptors (and the alphabet machine), and generates a new FST that performs a context dependent rewrite everywhere in the provided contexts. The context-dependent rewrite algorithm used is that of Mohri and Sproat (1996), and

see also Kaplan and Kay (1994). The fourth argument (`sigma_star`) needs to be a minimized machine. The fifth argument selects the direction of rewrite; we can either rewrite left-to-right or right-to-left or simultaneously. The sixth argument selects whether the rewrite is optional.

```
CDRewrite[tau, lambda, rho,
          sigma_star,
          'ltr'|'rtl'|'sim',
          'obl'|'opt']
```

For context-dependent rewrite rules, two built-in symbols “[BOS]” and “[EOS]” have a special meaning in the context specifications: they refer to the beginning and end of string, respectively.

There are also built-in functions that perform other tasks. In the interest of space we concentrate here on the `StringFile` function, which loads a file consisting of a list of strings, or tab-separated *pairs* of strings, and compiles them to an acceptor that represents the union of the strings.

```
StringFile['strings_file']
```

While it is equivalent to the union of the individual string (pairs), `StringFile` uses an efficient algorithm for constructing a prefix tree (trie) from the list and can be *significantly* more efficient than computing a union for large lists. If a line consists of a tab-separated pair of strings *a*, *b*, a transducer equivalent to `Rewrite[a, b]` is compiled.

The optional keywords `byte` (default), `utf8` or the name of a symbol table can be used to specify the parsing mode for the strings. Thus

```
StringFile['strings_file', utf8, my_symtab]
```

would parse a sequence of tab-separated pairs, using utf8 parsing for the left-hand string, and the symbol table `my_symtab` for the right-hand string.

4 NGram Library

The OpenGrm NGram library contains tools for building, manipulating and using n-gram language models represented as weighted finite-state transducers. The same finite-state topology is used to encode raw counts as well as smoothed models. Here we briefly present this structure, followed by details on the operations manipulating it.

An n-gram is a sequence of n symbols: $w_1 \dots w_n$. Each state in the model represents a prefix *history* of the n-gram ($w_1 \dots w_{n-1}$), and transitions in the model represent either n-grams or backoff transitions following that history. Figure 1 lists conventions for states and transitions used to encode the n-grams as a WFST.

This representation is similar to that used in other WFST-based n-gram software libraries, such as the AT&T GRM library (Allauzen et al., 2005). One key difference is the implicit representation of <s> and </s>, as opposed to encoding them as symbols in the grammar. This has the benefit of including all start and stop symbol functionality while avoiding common pitfalls that arise with explicit symbols.

Another difference from the GRM library representation is explicit inclusion of failure links from states to their backoff states even in the raw count files. The OpenGrm n-gram FST format is consistent through all stages of building the models, meaning that model manipulation (e.g., merging of two

Figure 1: List of state and transition conventions used to encode collection of n-grams in WFST.

An n-gram is a sequence of n symbols: $w_1 \dots w_n$. Its proper prefixes include all sequences $w_1 \dots w_k$ for $k < n$.

- There is a *unigram* state in every model, representing the empty string.
- Every proper prefix of every n-gram in the model has an associated state in the model.
- The state associated with an n-gram $w_1 ... w_n$ has a backoff transition (labeled with ϵ) to the state associated with its suffix $w_2 ... w_n$.
- An n-gram $w_1 ... w_n$ is represented as a transition, labeled with w_n, from the state associated with its prefix $w_1 ... w_{n-1}$ to a destination state defined as follows:
  - If $w_1 ... w_n$ is a proper prefix of an n-gram in the model, then the destination of the transition is the state associated with $w_1 ... w_n$
  - Otherwise, the destination of the transition is the state associated with the suffix $w_2 ... w_n$.
- Start and end of the sequence are not represented via transitions in the automaton or symbols in the symbol table. Rather
  - The start state of the automaton encodes the “start of sequence” n-gram prefix (commonly denoted <s>).
  - The end of the sequence (often denoted </s>) is included in the model through state final weights, i.e., for a state associated with an n-gram prefix $w_1 ... w_n$, the final weight of that state represents the weight of the n-gram $w_1 ... w_n$</s>.

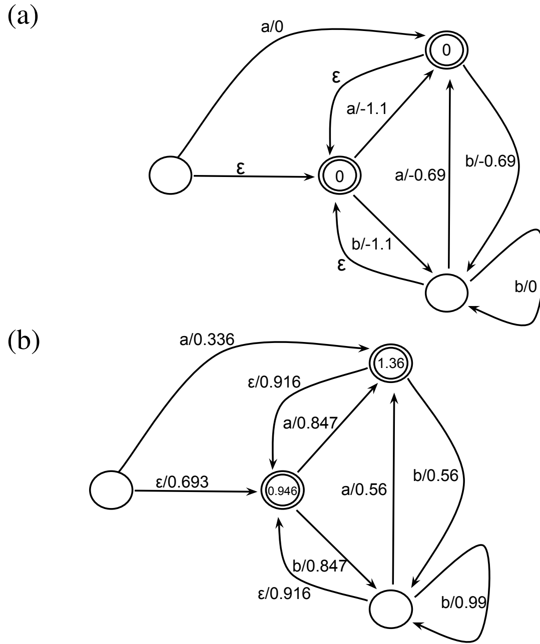


Figure 2: FST representations of (a) bigram and unigram counts; and (b) smoothed bigram model, when trained on the single string "`a b a b b a`"

models or count files, or pruning them) can be processed by the same operations. By convention, all counts and probabilities are stored as negative logs, and the FSTs are in the Tropical semiring. The symbol table provided during counting is kept with the model FSTs.

4.1 N-gram Counting

The command line binary `ngramcount` takes as input an FST archive (far) consisting of a collection of acyclic WFSTs and outputs an n-gram WFST of the specified order. The acyclic WFSTs can be linear automata representing strings from a corpus – easily compiled using the `farcompilestrings` command of OpenFst – or weighted word lattices output from, say, a speech recognition or machine translation system. In such a way, expected frequencies of n-grams can be counted. To count all trigrams, bigrams and unigrams in the compiled (far) corpus:

```
ngramcount -order=3 in.far >3g.cnt.fst
```

For example, counting with the `-order=2` flag (bigrams) from a corpus consisting of a single string "`a b a b b a`" yields the FST in Figure 2(a). Each state represents a prefix history: the leftmost state is the initial state, representing the <s> history; the central state is the *unigram* state, representing the ϵ history; the topmost state represents the history 'a'; and the bottom state represents the history 'b'. Since this is a bigram model, histories consist of at most one prior symbol from the vocabulary. Double circles represent final states, which come with a final weight encoding the negative log count of ending the string at that state. Only the 'a' history state and the unigram state are final states, since our example string ends with the symbol 'a'. (The unigram state is always final.) The ϵ transitions are backoff transitions, and the weights on each n-gram transition are negative log counts of that symbol occurring following the history that the state represents. Hence the bigram "`b b`" occurs once, yielding a negative log of zero for the transition labeled with 'b' leaving the state representing the history 'b'.

4.2 N-gram Model Parameter Estimation

Given counts, one can build a smoothed n-gram model by normalizing and smoothing, which is accomplished with the `ngrammake` command line binary. The library has several available smoothing methods, including Katz (1987), absolute discounting (Ney et al., 1994), Kneser-Ney (1995) and (the default) Witten-Bell (1991). See Chen and Goodman (1998) for a detailed presentation of these smoothing methods. Each of these smoothing methods is implemented as a relatively simple derived subclass, thus allowing for straightforward extension to new and different smoothing methods. To make a smoothed n-gram model from counts:

```
ngrammake 3g.cnt.fst >3g.mod.fst
```

Figure 2(b) shows the model built using the default Witten-Bell smoothing from the count FST in 2(a). The topology remains identical, but now the n-gram transition weights and final weights are negative log probabilities. The backoff transitions (labeled with ϵ) have the negative log backoff weights, which ensure that the model is correctly normalized.

Models, by default, are smoothed by interpolating higher- and lower-order probabilities. This is even true for methods more typically associated with backoff (rather than mixture) smoothing styles, such as Katz. While the smoothing values are estimated using interpolation, the model is encoded as a backoff model by pre-summing the interpolated probabilities, so that the backoff transitions are to be traversed only for symbols without transitions out of the current state. While these backoff transitions are labeled with ϵ, see Section 4.4 for discussion of applying them as failure transitions.

4.3 N-gram Model Merging and Pruning

Two n-gram count FSTs or two model FSTs can be merged into a single FST using `ngrammerge`, with command line flags to allow for scaling of each of the two, and to indicate whether to carry out full normalization. This approach allows for various sorts of MAP adaptation approaches for the n-gram models (Bacchiani et al., 2006). To merge two input FST models with no scaling:

```
ngrammerge in.mod1 in.mod2 >mrg.mod
```

N-gram model pruning is provided with three different methods: count pruning based on a threshold; the method from Seymore and Rosenfeld (1996); and relative entropy pruning of Stolcke (1998). Like smoothing, each of these pruning methods is implemented as a relatively simple derived subclass, thus allowing for straightforward extension to new and different pruning methods. To prune a smoothed n-gram model:

```
ngramshrink -theta=4 in.mod >prn.mod
```

4.4 N-gram Utilities

In addition to the above detailed core operations on language models, the OpenGrm NGram library has a number of utilities that make building and using the models very easy. There are utilities related to input and output, including `ngramsymbols`, which produces a symbol table from a corpus; `ngramread`, which reads in textual count files and models in ARPA format and encodes them as an FST; `ngramprint` which prints n-gram counts or ARPA format text files; and `ngraminfo` which displays information about the model, such as number of n-grams of various orders. There are also utilities related to the use of the models, including `ngramapply`, which applies the model to an input FST archive (far); `ngramrandgen` which randomly generates strings from the model; and `ngramperplexity` which calculates the perplexity of a corpus given the model. Note that `ngramapply` includes options for interpreting the backoff transitions as failure transitions.

Acknowledgments

This work was supported in part by a Google Faculty Research Award, NSF grant #IIS-0811745 and DARPA #HR0011-09-1-0041. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF or DARPA.

References

Cyril Allauzen, Mehryar Mohri, and Brian Roark. 2005. The design principles and algorithms of a weighted grammar library. *International Journal of Foundations of Computer Science*, 16(3):403–421.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Twelfth International Conference on Implementation and Application of Automata (CIAA 2007), Lecture Notes in Computer Science*, volume 4793, pages 11–23.

Michiel Bacchiani, Michael Riley, Brian Roark, and Richard Sproat. 2006. MAP adaptation of stochastic grammars. *Computer Speech and Language*, 20(1):41–68.

Stanley Chen and Joshua Goodman. 1998. An empirical study of smoothing techniques for language modeling. Technical report, TR-10-98, Harvard University.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20:331–378.

Slava M. Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recogniser. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 181–184.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238.

Hermann Ney, Ute Essen, and Reinhard Kneser. 1994. On structuring probabilistic dependences in stochastic language modeling. *Computer Speech and Language*, 8:1–38.

Kristie Seymore and Ronald Rosenfeld. 1996. Scalable backoff language models. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP)*.

Andreas Stolcke. 1998. Entropy-based pruning of backoff language models. In *Proc. DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274.

Ian H. Witten and Timothy C. Bell. 1991. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094.