OpenFst: A General and Efficient Weighted Finite-State Transducer Library

(Extended Abstract of an Invited Talk)

Cyril Allauzen[1], Michael Riley[2,⋆], Johan Schalkwyk[2], Wojciech Skut[2], and Mehryar Mohri[1]

[1] Courant Institute of Mathematical Sciences
251 Mercer ST, New York, NY 10012, USA
{allauzen,mohri}@cs.nyu.edu
[2] Google, Inc.
111 Eighth AV, New York, NY 10011, USA
{riley,johans,wojciech}@google.com

Abstract. We describe *OpenFst*, an open-source library for *weighted finite-state transducers* (WFSTs). OpenFst consists of a C++ template library with efficient WFST representations and over twenty-five operations for constructing, combining, optimizing, and searching them. At the shell-command level, there are corresponding transducer file representations and programs that operate on them. OpenFst is designed to be both very efficient in time and space and to scale to very large problems.

This library has key applications speech, image, and natural language processing, pattern and string matching, and machine learning.

We give an overview of the library, examples of its use, details of its design that allow customizing the labels, states, and weights and the lazy evaluation of many of its operations.

Further information and a download of the OpenFst library can be obtained from `http://www.openfst.org`.

Keywords: weighted automata, finite-state transducers, rational power series.

1 Introduction

A *weighted finite-state transducer* (WFST) is a finite automaton for which each transition has an input label, an output label, and a weight. Figure 1 depicts a weighted finite state transducer:

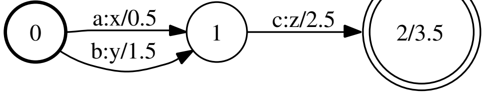

Fig. 1. An example weighted finite-state transducer

⋆ Corresponding author.

The initial state is labeled 0. The final state is 2 with final weight of 3.5. Any state with non-infinite final weight is a final state. There is a transition from state 0 to 1 with input label a, output label x, and weight 0.5. This machine transduces, for instance, the string ac to xz with weight 6.5 (the sum of the arc and final weights).

Weighted finite-state transducers have been used in speech recognition and synthesis, machine translation, optical character recognition, pattern matching, string processing, machine learning, information extraction and retrieval among others. Having a comprehensive software library of weighted transducer representations and core algorithms is key for using weighted transducers in these applications and for the development of new algorithms and applications.

To our knowledge, the first such software library was the AT&T FSM Library developed by Mohri, Pereira, and Riley for their work on transducer algorithms and applications [1]. It is available from AT&T for non-commercial use as executable binary programs. Since then, there have been various other weighted transducer library efforts [2,3,4,5]. Our motivation for OpenFst was to create a library as comprehensive and efficient as the AT&T FSM Library, but that was an open-source project. We also sought to make this library as flexible and customizable as possible given the wide range of applications WFSTs have enjoyed in recent years. It is a C++ template library, allowing it to be both very customizable and efficient.

This paper is an overview of this new library. Section 2 introduces some definitions and notation. Section 3 describes the representation and construction of transducers in this library. Section 4 briefly outlines the available algorithms. Section 5 provides examples of the library's use and discusses some new, simplified implementations of several algorithms. Section 6 gives more detail about the transducer representation and discusses the lazy evaluation of algorithms.

The OpenFst library is available for download from `http://www.openfst.org` and is released under the Apache license. Detailed documentation is also available at this site.

2 Definitions and Notation

The OpenFst Library closely parallels its mathematical foundations in the theory of rational power series [6,7,8]. The library user can define the alphabets and weights that label transitions. The weights may represent any set so long as they form a *semiring*.

A semiring $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is specified by a set of values $\mathbb{K}$, two binary operations $\oplus$ and $\otimes$, and two designated values $\overline{0}$ and $\overline{1}$. The operation $\oplus$ is associative, commutative, and has $\overline{0}$ as identity. The operation $\otimes$ is associative, has identity $\overline{1}$, distributes with respect to $\oplus$, and has $\overline{0}$ as annihilator: for all $a \in \mathbb{K}, a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$. If $\otimes$ is also commutative, we say that the semiring is *commutative*.

Table 1 lists some common semirings. All but the last are defined over subsets of the real numbers (extended with positive and negative infinity). In addition

Table 1. Semiring examples. $\oplus_{\log}$ is defined by: $x \oplus_{\log} y = -\log(e^{-x} + e^{-y})$

SEMIRING	SET	$\oplus$	$\otimes$	$\overline{0}$	$\overline{1}$
Boolean	$\{0,1\}$	$\vee$	$\wedge$	0	1
Probability	$\mathbb{R}_+$	$+$	$\times$	0	1
Log	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\oplus_{\log}$	$+$	$+\infty$	0
Tropical	$\mathbb{R} \cup \{-\infty, +\infty\}$	min	$+$	$+\infty$	0
String	$\Sigma^* \cup \{\infty\}$	lcp	$\cdot$	∞	ε

to the familiar Boolean semiring, and the probability semiring used to combine probabilities, two semirings often used in applications are the *log semiring* which is isomorphic to the probability semiring via the negative-log mapping, and the *tropical semiring* which is similar to the *log semiring* except the $\oplus$ operation is *min.* The *left (right) string semiring*, which is defined over strings, has longest common prefix (suffix) and concatenation as its operations, and has the (extended element) *infinite string* and the empty string for its identity elements. It only distributes on the left (right).

A *weighted finite-state transducer* $T = (\mathcal{A}, \mathcal{B}, Q, I, F, E, \lambda, \rho)$ over a semiring $\mathbb{K}$ is specified by a finite input alphabet $\mathcal{A}$, a finite output alphabet $\mathcal{B}$, a finite set of states Q, a set of initial states $I \subseteq Q$, a set of final states $F \subseteq Q$, a finite set of transitions $E \subseteq Q \times (\mathcal{A} \cup \{\varepsilon\}) \times (\mathcal{B} \cup \{\varepsilon\}) \times \mathbb{K} \times Q$, an initial state weight assignment $\lambda : I \to \mathbb{K}$, and a final state weight assignment $\rho : F \to \mathbb{K}$. $E[q]$ denotes the set of transitions leaving state $q \in Q$.

Given a transition $e \in E$, $p[e]$ denotes its origin or previous state, $n[e]$ its destination or next state, $i[e]$ its input label, $o[e]$ its output label, and $w[e]$ its weight. A *path* $\pi = e_1 \cdots e_k$ is a sequence of consecutive transitions: $n[e_{i-1}] = p[e_i]$, $i = 2, \ldots, k$. The functions n, p, and w on transitions can be extended to paths by setting: $n[\pi] = n[e_k]$ and $p[\pi] = p[e_1]$ and by defining the weight of a path as the $\otimes$-product of the weights of its constituent transitions: $w[\pi] = w[e_1] \otimes \cdots \otimes w[e_k]$. More generally, w is extended to any finite set of paths R by setting $w[R] = \bigoplus_{\pi \in R} w[\pi]$; if the semiring is closed, this is defined even for infinite R. We denote by $P(q, q')$ the set of paths from q to q' and by $P(q, x, y, q')$ the set of paths from q to q' with input label $x \in \mathcal{A}^*$ and output label $y \in \mathcal{B}^*$. These definitions can be extended to subsets $R, R' \subseteq Q$ by $P(R, R') = \cup_{q \in R,\, q' \in R'} P(q, q')$, $P(R, x, y, R') = \cup_{q \in R,\, q' \in R'} P(q, x, y, q')$.

A transducer T is *regulated* if the weight associated by T to any pair of input-output string (x, y) given by:

$$[\![T]\!](x, y) = \bigoplus_{\pi \in P(I, x, y, F)} \lambda[p[\pi]] \otimes w[\pi] \otimes \rho[n[\pi]] \tag{1}$$

is well-defined and in $\mathbb{K}$. If $P(I, x, y, F) = \emptyset$, then $T(x, y) = \overline{0}$. A weighted transducer without ε-cycles is regulated.

3 Transducer Representation and Construction

In the OpenFst Library, a transducer can be constructed from either the C++ level using class constructors and mutators or from a shell-level program using a textual file representation. We describe the former here.

This C++ code creates the transducer in Figure 1:

```
// A vector FST is a general mutable FST
VectorFst<StdArc> fst;

// Add state 0 to the initially empty FST and make it the initial state
fst.AddState(); // 1st state will be state 0 (returned by AddState)
fst.SetStart(0); // arg is state ID

// Add two arcs exiting state 0
// Arc constructor args: ilabel, olabel, weight, dest state ID
fst.AddArc(0, StdArc(1, 1, 0.5, 1)); // 1st arg is src state ID
fst.AddArc(0, StdArc(2, 2, 1.5, 1));

// Add state 1 and its arc
fst.AddState();
fst.AddArc(1, StdArc(3, 3, 2.5, 2));

// Add state 2 and set its final weight
fst.AddState();
fst.SetFinal(2, 3.5); // 1st arg is state ID, 2nd arg weight
```

The steps consist of first constructing an empty `VectorFst`, which is a general-purpose transducer that uses an adjacency list representation (stored in STL vectors). Next, its mutator member functions are used to add states and transitions ('arcs') and to set the intial state[1] and final weights. States are identified by integer labels. The result can be saved to a file with `fst.Write('out.fst')`.

The `VectorFst`, like all transducer representations and algorithms in this library, is templated on the transition type. This permits customization of the labels, state IDs and weights in a transducer. `StdArc` defines the library-standard transition representation:

```
struct StdArc {
  typedef int Label;
  typedef TropicalWeight Weight;
  typedef int StateId;

  Label ilabel;                 // Transition input label
  Label olabel;                 // Transition output label
  Weight weight;                // Transition weight
  StateId nextstate;            // Transition destination state
};
```

[1] Only one initial state is permitted and it has weight $\overline{1}$.

This uses 32-bit integer labels and state IDs and the class `TropicalWeight` for its weight.

A Weight class holds the set element and provides the semiring operations. `TropicalWeight` uses a single-precision float to hold the set element, has member functions that define the identity elements $\overline{0}$ and $\overline{1}$ and has associated functions `Plus` and `Times` that implement $\oplus$ and $\otimes$, respectively:

```
class TropicalWeight {
public:
  TropicalWeight(float f) : value_(f) {}
  static TropicalWeight Zero() { return TropicalWeight(Infinity); }
  static TropicalWeight One() { return TropicalWeight(0.0); }
private:
  float value_;
};
TropicalWeight Plus(TropicalWeight x, TropicalWeight y) {
   return w1.value_ < w2.value_ ? w1 : w2;
};
```

The class `LogWeight` is also defined in this library. Users may define their own weight classes and transition types. So long as the weights form a semiring, either a library algorithm will work correctly, or in the case where additional requirements are placed on the weights (such as commutivity), an error will be signalled if the condition is not met.

One simple customization is to use smaller or larger precision labels, state IDs or weights for a more compact or higher capacity representation, respectively. A less trivial extension is to use the `ProductWeight` template, also provided by the library:

```
template <typename W1, typename W2>
class ProductWeight {
public:
  ProductWeight(W1 w1, W2 w2) : value1_(w1), value2_(w2) {}
  static ProductWeight Zero() {
    return ProductWeight(W1::Zero(), W2::Zero());
  }
  static ProductWeight One() {
    return ProductWeight(W1::One(), W2::One());
  }
private:
  W1 value1_;
  W2 value2_;
};
```

For instance, the product semiring of the tropical semiring with itself can be created with

`ProductWeight<TropicalWeight, TropicalWeight>`.

Defined over $(\mathbb{R} \cup \{-\infty, +\infty\}) \times (\mathbb{R} \cup \{-\infty, +\infty\})$, this weight class could be used, for example, in speech recognition applications to store both the acoustic and language model scores in recognizer output hypothesis set ('lattice').

Another example of a semiring defined on pairs is the *expectation semiring*. Let $\mathbb{K}$ denote $(\mathbb{R}\cup\{+\infty, -\infty\}) \times (\mathbb{R}\cup\{+\infty, -\infty\})$. For pairs (x_1, y_1) and (x_2, y_2) in $\mathbb{K}$, define the following:

$$(x_1, y_1) \oplus (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$
$$(x_1, y_1) \otimes (x_2, y_2) = (x_1 x_2, x_1 y_2 + x_2 y_1)$$

The system $(\mathbb{K}, \oplus, \otimes, (0, 0), (1, 0))$ defines a commutative semiring. [9] show how to use this semiring to compute the relative entropy between probabilistic automata. This algorithm is trivially implemented in this library by adding the expectation semiring and using the intersection and shortest-distance library algorithms.

4 Transducer Algorithms

4.1 Algorithm Implementation Types

The algorithms in the library fall into three implementation types. The *destructive* algorithms modify in place. e.g., "`Invert(&fst)`". The *constructive* algorithms copy their result into a provided mutable transducer, e.g., "`Reverse(fst, &inverted_fst)`". Both have a complexity that is a function of the number of states and transitions in the result. The *lazy* (or *delayed*) algorithms are separate transducer C++ classes, e.g., "`InvertFst<StdArc> inverted_fst(fst)`". They do no computation on construction and their complexity is a function of the number of states and transitions *visited*. This is useful in applications where the whole result may not be visited, e.g., with Dijsktra's algorithm (with positive weights) or in a pruned search. In Section 6, the implementation of lazy algorithms is described.

4.2 Available Algorithms

The library provides over twenty-five transducer operations. We briefly describe some of them below.

Table 2 shows the library operations for the sum, product, and Kleene closure of weighted transducers. Both destructive implementations, using the Thompson construction, and lazy implementations are provided.

Table 3 shows the elementary unary operations for reversing the strings in an automaton, inverting a transduction, and projecting a transduction onto its domain or range. `Invert` and `Project` have both destrucive and lazy implementations, while Reverse has a constructive implementation. In Section 6, we outline the implementation of both forms of `Invert`.

Table 2. The rational operations

Operation	Definition
Union	$[\![T_1 \oplus T_2]\!](x,y) = [\![T_1]\!](x,y) \oplus [\![T_2]\!](x,y)$
Concat	$[\![T_1 \otimes T_2]\!](x,y) = \bigoplus_{x=x_1x_2, y=y_1y_2} [\![T_1]\!](x_1,y_1) \otimes [\![T_2]\!](x_2,y_2)$
Closure	$[\![T^*]\!](x,y) = \bigoplus_{n=0}^{\infty} [\![T]\!]^n(x,y)$

Table 3. Elementary unary operations

Operation	Definition and Notation	Lazy
Reverse	$[\![\widetilde{T}]\!](x,y) = [\![T]\!](\widetilde{x},\widetilde{y})$	No
Invert	$[\![T^{-1}]\!](x,y) = [\![T]\!](y,x)$	Yes
Project	$[\![A]\!](x) = \bigoplus_{y} [\![T]\!](x,y)$	Yes

Table 4 shows several binary operations based on the composition algorithm: the composition of transducers, the intersection of acceptors, and the difference of an acceptor and an unweighted, deterministic acceptor [6,7]. Composition is a fundamental operation used to apply or cascade transductions (see Section 5). All have lazy implementations.

Table 4. Fundamental binary operations

Operation	Definition and Notation	Condition
Compose	$[\![T_1 \circ T_2]\!](x,y) = \bigoplus_{z} [\![T_1]\!](x,z) \otimes [\![T_2]\!](z,y)$	$\mathbb{K}$ commutative
Intersect	$[\![A_1 \cap A_2]\!](x) = [\![A_1]\!](x) \otimes [\![A_2]\!](x)$	$\mathbb{K}$ commutative
Difference	$[\![A_1 - A_2]\!](x) = [\![A_1 \cap \overline{A_2}]\!](x)$	A_2 unweighted & deterministic

Table 5 shows operations that optimize transducers: trimming, epsilon-removal, and weighted determinization and minimization. Several of these algorithms have specific semiring conditions for their use. Not all weighted transducers can be determinized [10,11,12].

Table 5. Optimization operations

Operation	Description	Lazy
Connect	Removes non-accessible/non-coaccessible states	No
RmEpsilon	Removes ε-transitions	Yes
Determinize	Creates equivalent deterministic transducr	Yes
Minimize	Creates equivalent minimal deterministic transducer	No

Table 6 shows operations for sorting a transducer's transitions or states, for pushing their weights and labels toward the initial or final states, for placing all input ε's after the non-ε's and for synchronizing the ε delay [13,12].

Table 6. Normalization operations

Operation	Description	Lazy
TopSort	Topologically sorts an acyclic transducer	No
ArcSort	Sorts state's arcs given an order relation	Yes
Push	Creates equivalent pushed/stochastic machine	No
EpsNormalize	Places input ε's after non-ε's on paths	No
Synchronize	Produces monotone ε delay	Yes

Table 7 shows operations that search for shortest paths or distances in a weighted automaton or prune away states and transitions on paths that have weights that exceed a threshold [13].

Table 7. Search operations

Operation	Description
ShortestPath	Finds n-shortest paths
ShortestDistance	Finds single-source shortest-distances
Prune	Prunes states and transitions by path weight

Usage information, graphical examples, and complexities of all library operations are provided in the documentation available at `http://www.openfst.org`.

5 Examples

In this section, we give examples of the use of the library algorithms. First, we give a simple example of transducer application:

```
// Reads in an input FST.
StdFst *input = StdFst::Read("input.fst");

// Reads in the transduction model.
StdFst *model = StdFst::Read("model.fst");

// The FSTs must be sorted along the dimensions they will be joined.
// In fact, only one needs to be so sorted.
// This could have instead been done for “model.fst” when it was created.
ArcSort(input, StdOLabelCompare());
ArcSort(model, StdILabelCompare());

// Container for composition result.
StdVectorFst result;

// Create the composed FST
Compose(*input, *model, &result);

// Just keeps the output labels
Project(&result, PROJECT_OUTPUT);
```

An input automaton and a transducer to which the input will be applied are first read from files. Then these automata are sorted as required by the composition algorithm. Next, they are composed and the result is projected onto the output labels.

Next we give an example of using different semirings to compute the shortest distance from the initial state to each state q [13]:

```
// Tropical semiring
Fst<StdArc> *input = Fst<StdArc>::Read("input.fst");
vector<StdArc::Weight> distance;
ShortestDistance(*input, &distance);

// Log semiring
Fst<LogArc> *input = Fst::Read("input.fst");
vector<LogArc::Weight> distance;
ShortestDistance(*input, &distance);

// Right string semiring
typedef StringArc<TropicalWeight, STRING_RIGHT> SR;
Fst<SR> *input = Fst::Read("input.fst");
vector<SR::Weight> distance;
ShortestDistance(*input, &distance);

// Left string semiring
typedef StringArc<TropicalWeight, STRING_LEFT> SL;
Fst<SL> *input = Fst::Read("input.fst");
vector<SL::Weight> distance;
ShortestDistance(*input, &distance);
ERROR: ShortestDistance: Weights need to be right distributive
```

With the tropical semiring, the minimum path weight to q is computed. With the log semiring, the (log) sum of path weights to q is computed. With the right string semiring, the longest common suffix among paths to q is computed. With the left string semiring, an error is signalled, since the semiring is only left-distributive.

We have represented a transition as:

$$e \in Q \times (\mathcal{A} \cup \{\varepsilon\}) \times (\mathcal{B} \cup \{\varepsilon\}) \times \mathbb{K} \times Q.$$

This treats the input and output labels symmetrically, is space-efficient since there is a single output-label per transition, and is the natural representation for the composition algorithm whose efficiency is critical in many applications. However, an alternative representation of a transition is:

$$e \in Q \times (\mathcal{A} \cup \{\varepsilon\}) \times \mathcal{B}^* \times \mathbb{K} \times Q.$$

or equivalently,

$$e \in Q \times (\mathcal{A} \cup \{\varepsilon\}) \times \mathbb{K}' \times Q, \qquad \mathbb{K}' = \mathcal{B}^* \times \mathbb{K}.$$

This treats string and $\mathbb{K}$ outputs uniformly and is the natural representation for weighted transducer determinization, minimization, label pushing, and epsilon normalization [10,11,13,12]. Implementing these algorithms in the original transition representation is awkward and complex.

We can use the alternative transition representation in this library, combining each transitions output label and weight into a new product weight, with:

```
typedef ProductWeight⟨StringWeight,TropicalWeight⟩CompositeWeight;
```

The following shows how this composite weight is used to implement weighted determinization:

```
Fst<StdArc> *input = Fst::Read("input.fst");
// Converts into alternative transition representation
MapFst<StdArc, CompositeArc> composite(*input, ToCompositeMapper);
WeightedDeterminizeFst<CompositeArc> det(composite);
// Ensures only one output label per transition (functional input)
FactorWeightFst<CompositeArc> factor(det);
// Converts back from alternative transition representation
MapFst<CompositeArc> result(factor, FromCompositeMapper);
```

First, the input is converted to the alternate representation using an operation that maps a conversion function (object) across all transitions. Next, generic weighted (acceptor) determiniztaion is applied, and then the result is converted back to the original transition representation. Performance is not sacrificed given efficient lazy computation and string semiring implementations. Weighted transducer minimization, label pushing and epsilon normalization are easily implemented in a similar way using this change of transition representation and the generic (acceptor) weighted minimization, weight pushing, and ε-removal algorithms.

6 Transducer Class Design and Lazy Evaluation

In this section, the details of the transducer representations used and the implementation of lazy evaluation are described. In this library there are many transducer represenations. Some are mutable containers like `VectorFst`, others are immutable containers like `ConstFst`, while others implement the lazy evaluation of operations like `InvertFst`. They all share the abstract base `Fst` class:

```
template <class Arc>
class Fst {
public:
  virtual StateId Start() const = 0;             // Initial state
  virtual Weight Final(StateId) const = 0;       // State's final weight
  static Fst<Arc> *Read(const string filename);
}
```

Two companion classes, `StateIterator` and `ArcIterator`, are defined that each have methods `Next`, `Done`, and `Value`. These classes provide the minimum information needed to specify a transducer. This includes its initial state, final weights, and operations to step through the states of the transducer and the transitions of a state. Any class that derives from `Fst` and correctly implements these methods can be used with any algorithm that accepts this base type as its argument.

The destructive algorithms mutate their input. They use as arguments another abstract class, `MutableFst`, that derives from `Fst`. This class adds methods `SetStart`, `SetFinal`. `AddState`, and `AddArc` and has a companion class `MutableArcIterator` that adds method `SetValue` to `ArcIterator`.

The following shows the implementation of destructive `Invert` using these classes. The states and transitions are traversed and the input and output swapped in place:

```
template <class Arc> void Invert(MutableFst<Arc> *fst) {
  for (StateIterator< MutableFst<Arc> > siter(*fst);
    !siter.Done();
    siter.Next()) {
      StateId s = siter.Value();
      for (MutableArcIterator< MutableFst<Arc> > aiter(fst, s);
        !aiter.Done();
        aiter.Next()) {
          Arc arc = aiter.Value();
          Label l = arc.ilabel;
          arc.ilabel = arc.olabel;
          arc.olabel = l;
          aiter.SetValue(arc);
      }
  }
}
```

The following shows the implementation of lazy `InvertFst`:

```
template <class Arc> class InvertFst : public Fst<Arc> {
public:
  virtual StateId Start() const { return fst_->Start(); }
  ...
private:
  const Fst<Arc> *fst_;
}

template <class F> Arc ArcIterator<F>::Value() const {
  Arc arc = arcs_[i_];
  Label l = arc.ilabel;
  arc.ilabel = arc.olabel;
  arc.olabel = l;
  return arc;
}
```

This is a new class that derives from `Fst`. It forwards most of its methods to the input transducer. However, its companion `ArcIterator` swaps the input and output labels when a transition is requested. Note the input transducer is not modified and no computation is performed until requested. While the lazy inversion operation is trivial, more complex operations like composition and determinization are naturally implemented in this way as well.

7 Conclusion

This paper has presented an overview of a new comprehensive and flexible weighted finite-state transducer library whose source code is freely available. We encourage readers to visit `http://www.openfst.org` to download the library. There is an easy-to-use shell-level interface that we did not describe here, but that is documented on the web site and is a good place to start. It is our hope that others will find this library useful in the years to come.

Acknowledgments

The research of Cyril Allauzen and Mehryar Mohri was partially supported by the New York State Office of Science Technology and Academic Research (NYSTAR). This project was also sponsored in part by the Department of the Army Award Number W81XWH-04-1-0307. The U.S. Army Medical Research Acquisition Activity, 820 Chandler Street, Fort Detrick MD 21702-5014 is the awarding and administering acquisition office. The content of this material does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

1. Mohri, M., Pereira, F., Riley, M.: The design principles of a weighted finite-state transducer library. Theoretical Computer Science 231, 17–32 (2000)
2. Adant, A.: WFST: a finite-state template library in C++ (2000), `http://membres.lycos.fr/adant/tfe`
3. Hetherington, L.: The MIT finite-state transducer toolkit for speech and language processing. In: Proceedings of the ICSLP, Jeju, South Korea (2004)
4. Kanthak, S., Ney, H.: FSA: An efficient and flexible C++ toolkit for finite state automata using on-demand computation. In: Proceedings of 42nd Meeting of the ACL, pp. 510–517 (2004)
5. Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing VAUCANSON. Theoretical Computer Science 328, 77–96 (2004)
6. Salomaa, A., Soittola, M.: Automata-Theoretic Aspects of Formal Power Series. Springer, New York (1978)
7. Kuich, W., Salomaa, A.: Semirings, Automata, Languages. Number 5 in EATCS Monographs on Theoretical Computer Science. Springer, Germany (1986)
8. Berstel, J., Reutenauer, C.: Rational Series and Their Languages. Springer, New York (1988)

9. Cortes, C., Mohri, M., Rastogi, A., Riley, M.: On the computation of the relative entropy of probabilistic automata. International Journal of Foundations of Computer Science (2007)
10. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23 (1997)
11. Mohri, M.: Minimization algorithms for sequential transducers. Theoretical Computer Science 234, 177–201 (2000)
12. Mohri, M.: Generic epsilon-removal and input epsilon-normalization algorithms for weighted transducers. International Journal of Foundations of Computer Science 13, 29–143 (2002)
13. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7, 321–350 (2002)