OpenFst - a General and Efficient Weighted Finite-State Transducer Library

Cyril Allauzen - allauzen@cs.nyu.edu
Michael Riley - riley@google.com
Johan Schalkwyk - johans@google.com
Wojciech Skut - wojciech@google.com
Mehryar Mohri - mohri@{cs.nyu,edu, google.com}

July 17, 2007

OpenFst Library

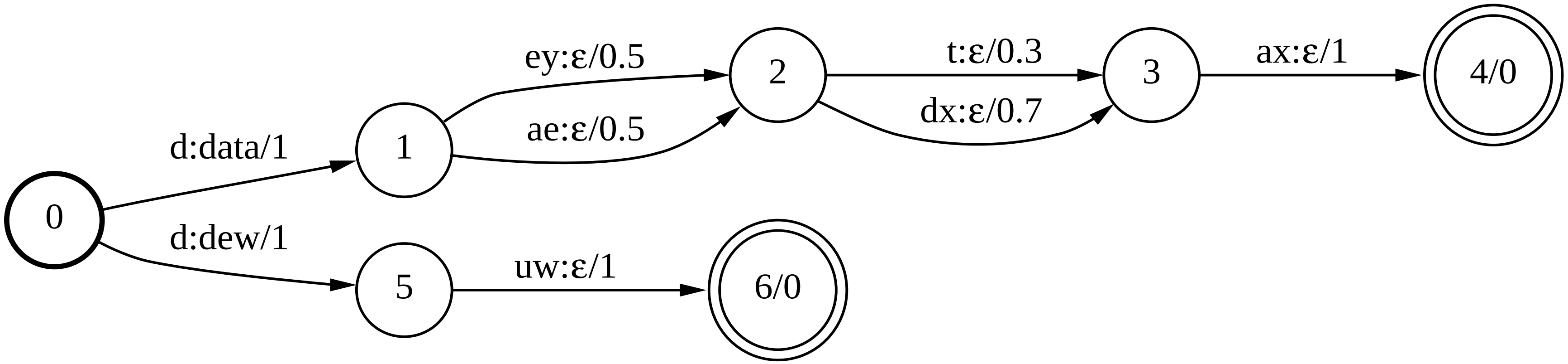
- C++ template library for constructing, combining, optimizing, and searching *weighted finite-states transducers (FSTs)*.
- Goals: Comprehensive, flexible, efficient and scale well to large problems.
- Origins: AT&T, merged efforts from Google and the NYU Courant Institute.
- Documentation and Download: http://www.openfst.org
- Released under the Apache license.
- Talk is not about new algorithms – uses previously published algorithms (many by the authors), but does discuss new simplified implementations of some of these algorithms.
- Talk is about a software library which we have found very useful and hope others do now that it is open-source.

Current OpenFst Applications

- Speech recognition (speech-to-text): lexicons, language models, phonetic context-dependency, recognizer hypothesis sets.
- Speech synthesis (text-to-speech): text normalization, pronunciation models
- Optical character recognition: lexicons, language models
- Information extraction: pattern matching, text processing
- Music identification: 'music phone' lexicon

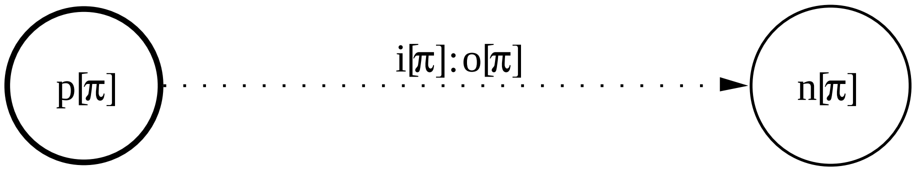
Weighted Transducers

- Finite automata with input labels, output labels, and weights.
- Example: *Pronunciation lexicon transducer:*



Definitions and Notation – Paths

- **Path π**
  - Origin or previous state: $p[\pi]$.
  - Destination or next state: $n[\pi]$.
  - Input label: $i[\pi]$.
  - Output label: $o[\pi]$.



- **Sets of paths**
  - $P(R_1, R_2)$: set of all paths from $R_1 \subseteq Q$ to $R_2 \subseteq Q$.
  - $P(R_1, x, R_2)$: paths in $P(R_1, R_2)$ with input label x.
  - $P(R_1, x, y, R_2)$: paths in $P(R_1, x, R_2)$ with output label y.

Definitions and Notation – Transducers

- Alphabets: input Σ, output Δ.
- States: Q, initial states I, final states F.
- Transitions: $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \mathbb{K} \times Q$.
- Weight functions:

  initial weight function $\lambda : I \rightarrow \mathbb{K}$

  final weight function $\rho : F \rightarrow \mathbb{K}$.
- Transducer $T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$ with for all $x \in \Sigma^*, y \in \Delta^*$:

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

Semirings

A *semiring* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ = a ring that may lack negation.

- **Sum**: to compute the weight of a sequence (sum of the weights of the paths labeled with that sequence).
- **Product:** to compute the weight of a path (product of the weights of constituent transitions).

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

with $\oplus_{\log}$ defined by: $x \oplus_{\log} y = -\log(e^{-x} + e^{-y})$.

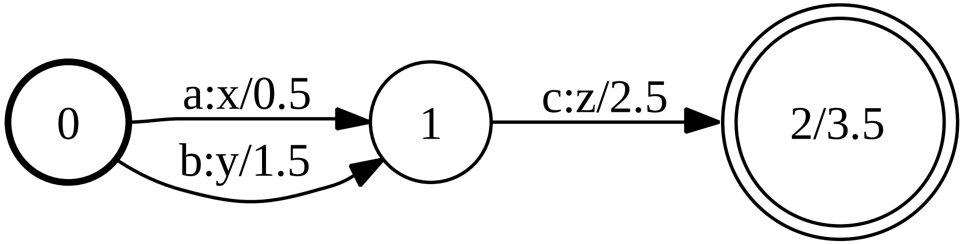
Finite-State Transducer Construction

Input Methods:

- Finite-state transducer:
  - Textual transducer file representation
  - C++ code
  - Graphical user interface
- Regular expressions
- “Context-free” rules
- “Context-dependent” rules

FST Textual File Representation

- **Graphical Representation** (`T.ps`):



- **Transducer File** (`T.txt`):

```
0   1   a   x   0.5
0   1   b   y   1.5
1   2   c   z   2.5
2  3.5
```

- **Input Symbols File** (`T.isyms`):

```
a  1
b  2
c  3
```

- **Output Symbols File** (`T.osyms`):

```
x  1
y  2
z  3
```

Compiling, Printing, Reading, and Writing FSTs

- **Compiling**

```
fstcompile -isymbols=T.isyms -osymbols=T.osyms T.txt T.fst
```

- **Printing**

```
fstprint -isymbols=T.isyms -osymbols=T.osyms T.fst >T.txt
```

- **Drawing**

```
fstdraw -isymbols=T.isyms -osymbols=T.osyms T.fst >T.dot
```

- **Reading**

```
Fst<Arc> *fst = Fst<Arc>::Read(‘‘T.fst’’)
```

- **Writing**

```
fst.Write(‘‘T.fst’’)
```

C++ FST Construction

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

// Add state 0 to the initially empty FST and make it the start 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
```

OpenFst Design: Arc (transition)

Labels and states may be any integral type; weights may be any class that forms a semiring:

```
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
};
```

OpenFst Design: Tropical Weight

A Weight class holds the set element and provides the semiring operations:

```
class TropicalWeight {
public:
  TropicalWeight(float f) : value_(f) {}
  static TropicalWeight Zero() { return TropicalWeight(kPositiveInfinity); }
  static TropicalWeight One() { return TropicalWeight(0.0); }
private:
  float value_;
};

TropicalWeight Plus(TropicalWeight x, TropicalWeight y) {
   return w1.value_ < w2.value_ ? w1 : w2;
};
```

OpenFst Design: Product Weight

This template allows easily creating the product semiring from two (or more) semirings.

```
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:
  float value1_;
  float value2_;
};
```

Operation Implementation Types

- **Destructive:** Modifies input; $O(|Q| + |E|)$:

```
StdFst *input = StdFst::Read("input.fst");
Invert(input);
```

- **Constructive:** Writes to output; $O(|Q| + |E|)$:

```
StdFst *input = StdFst::Read("input.fst");
StdVectorFst output;
ShortestPath(input, &output);
```

- **Lazy (or Delayed):** Creates new Fst; $O(|Q_{visit}| + |E_{visit}|)$:

```
StdFst *input = StdFst::Read("input.fst");
StdFst *output = new StdInvertFst(input);
```

Lazy implementations are useful in applications where the whole machine may not be visited, e.g. Dijsktra (positive weights), pruned search.

Rational Operations

- **Definitions**

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	$[\![T^*]\!](x,y) = \bigoplus_{n=0}^{\infty} [\![T]\!]^n(x,y)$

- **Implementations:** Lazy and non-lazy

Elementary Unary Operations

- **Definitions**

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

- **Implementations:** Non-lazy, for lazy see table.

Fundamental Binary Operations

- **Definitions**

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

- **Implementations:** Non-lazy and lazy.

Optimization Operations

- **Definitions**

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

- **Conditions**: There are specific semiring conditions for the use of these algorithms. Not all weighted transducers can be determinized using that algorithm.

- **Implementations**: Non-lazy, for lazy see table.

Normalization Operations

- **Definitions**

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 epsilon delay	Yes

- **Implementations:** Non-lazy, for lazy see table.

Search Operations

- **Definitions**

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

- **Implementations:** Non-lazy.

Example: FST Application - Shell-Level

```
# 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.
$ fstarcsort --sort_type=olabel input.fst input_sorted.fst
$ fstarcsort --sort_type=ilabel model.fst model_sorted.fst

# Creates the composed FST
$ fstcompose input_sorted.fst model_sorted.fst comp.fst

# Just keeps the output label
$ fstproject --project_output comp.fst result.fst

# Do it all in a single command line
$ fstarcsort --sort_type=ilabel model.fst |
fstcompose input.fst - | fstproject --project_output result.fst
```

Example: FST Application - C++

```
// 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);
```

Example: Shortest-Distance with Various Semirings

- 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> SA;
Fst<SA> *input = Fst::Read("input.fst");
vector<SA::Weight> distance;
ShortestDistance(*input, &distance);
```

- Left String Semiring:

```
ERROR: ShortestDistance:  Weights need to be right distributive
```

Transition Representation

- We have represented a transition as:

$$e \in Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \mathbb{K} \times Q.$$

  - Treats input and output symmetrically
  - Space-efficient single output-label per transition
  - Natural representation for composition algorithm

- Alternative representation of a transition:

$$e \in Q \times (\Sigma \cup \{\epsilon\}) \times \Delta^* \times \mathbb{K} \times Q.$$

  or equivalently,

$$e \in Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{K}' \times Q, \qquad \mathbb{K}' = \Delta^* \times \mathbb{K}.$$

  - Treats string and $\mathbb{K}$ outputs uniformly
  - Natural representation for weighted transducer determinization, minimization, label pushing, and epsilon normalization.

Using the Alternative Transition Representation

- We can use the alternative transition representation with:

```
typedef ProductWeight<StringWeight, TropicalWeight> CompositeWeight;
```

- Weighted transducer determinization becomes:

```
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);
```

- Efficiency is not sacrificed given the lazy computation and an efficient string semiring representation.
- Weighted transducer minimization, label pushing and epsilon normalization are similarly implemented easily using the generic (acceptor) weighted minimization, weight pushing, and epsilon removal algorithms.

Example: 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 :

$$\begin{aligned}(x_1, y_1) \oplus (x_2, y_2) &= (x_1 + x_2, y_1 + y_2)\\(x_1, y_1) \otimes (x_2, y_2) &= (x_1 x_2, x_1 y_2 + x_2 y_1)\end{aligned}$$

The system $(\mathbb{K}, \oplus, \otimes, (0, 0), (1, 0))$ defines a commutative semiring.

This semiring combined with the composition and shortest-distance algorithms can be used to compute the relative entropy between probabilistic automata [C. Cortes, M. Mohri, A. Rastogi, and M. Riley. On the Computation of the Relative Entropy of Probabilistic Automata. *International Journal of Foundations of Computer Science*, 2007.]:

$$D(A \| B) = \sum_x [\![A]\!](x) \log [\![A]\!](x) - \sum_x [\![A]\!](x) \log [\![B]\!](x).$$

This algorithm is trivially implemented in the OpenFst Library.

OpenFst Design: Fst (generic)

```
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);
}
```

OpenFst Design: State Iterator

```
template <class F>
class StateIterator {
public:
  explicit StateIterator(const F &fst);
  virtual ~StateIterator();
  virtual bool Done();                        // States exhausted?
  virtual StateId Value() const;              // Current state Id
  virtual void Next();                        // Advance a state
  virtual void Reset();                       //Start over
}
```

OpenFst Design: Arc Iterator

```
template <class F>
class ArcIterator {
public:
  explicit ArcIterator(const F &fst, StateId s);
  virtual ~ArcIterator();
  virtual bool Done();                        // Arcs exhausted?
  virtual const Arc &Value() const;           // Current arc
  virtual void Next();                        // Advance an arc
  virtual void Reset();                       // Start over
  virtual void Seek(size_t a);                // Random access
}
```

OpenFst Design: MutableFst

```
template <class Arc>
class MutableFst : public Fst<Arc> {
public:
  void SetStart(StateId s);                   // Set initial state
  void SetFinal(StateId s, Weight w);         // Set final weight
  void AddState();                            // Add a state
  void AddArc(StateId s, const Arc &arc);     // Add an arc
}
```

OpenFst Design: Mutable Arc Iterator

```
template <class F>
class MutableArcIterator {
public:
  explicit MutableArcIterator(F *fst, StateId s);
  virtual ~MutableArcIterator();
  virtual bool Done();                              // Arcs exhausted?
  virtual const Arc &Value() const;                 // Current arc
  virtual void Next();                              // Advance an arc
  virtual void Reset();                             // Start over
  virtual void Seek(size_t a);                      // Random access
  virtual void SetValue(const Arc &arc);            // Set current arc
}
```

OpenFst Design: Invert (Destructive)

```
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);
      }
  }
}
```

OpenFst Design: Invert (Lazy)

```
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;
}
```