OpenFst Library
# Table of Contents

**OpenFst Library**.................................................................................................................................................................1

**OpenFst Authors**..................................................................................................................................................................2

Principal Contacts....................................................................................................................................................................2
Contributors........................................................................................................................................................................2

**OpenFst Background Material**................................................................................................................................................3

**OpenFst Quick Tour**............................................................................................................................................................4
Finding and Using the Library.........................................................................................................................................................4
Example FST.............................................................................................................................................................................4
Creating FSTs............................................................................................................................................................................4
Accessing FSTs...........................................................................................................................................................................6
FST Operations..........................................................................................................................................................................8
FST Weights.............................................................................................................................................................................12

**ArcSort**..............................................................................................................................................................................14
Description............................................................................................................................................................................14
Usage...................................................................................................................................................................................14
Examples..............................................................................................................................................................................14
Complexity...........................................................................................................................................................................15

**Closure**............................................................................................................................................................................16
Description............................................................................................................................................................................16
Usage...................................................................................................................................................................................16
Examples..............................................................................................................................................................................16
Complexity...........................................................................................................................................................................17

**Compose**...........................................................................................................................................................................18
Description............................................................................................................................................................................18
Usage...................................................................................................................................................................................18
Examples..............................................................................................................................................................................18
Complexity...........................................................................................................................................................................19
Caveats................................................................................................................................................................................19

**Concat**............................................................................................................................................................................20
Description............................................................................................................................................................................20
Usage...................................................................................................................................................................................20
Examples..............................................................................................................................................................................20
Complexity...........................................................................................................................................................................21
Caveats................................................................................................................................................................................21

**Connect**............................................................................................................................................................................22
Description............................................................................................................................................................................22
Usage...................................................................................................................................................................................22
Examples..............................................................................................................................................................................22
Complexity...........................................................................................................................................................................22
# OpenFst Library

## Table of Contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determinize</td>
<td>23</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>23</td>
</tr>
<tr>
<td>Examples</td>
<td>23</td>
</tr>
<tr>
<td>Complexity</td>
<td>23</td>
</tr>
<tr>
<td>References</td>
<td>24</td>
</tr>
<tr>
<td>Difference</td>
<td>25</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>25</td>
</tr>
<tr>
<td>Examples</td>
<td>25</td>
</tr>
<tr>
<td>Complexity</td>
<td>26</td>
</tr>
<tr>
<td>Caveats</td>
<td>26</td>
</tr>
<tr>
<td>Encode/Decode</td>
<td>27</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>27</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>27</td>
</tr>
<tr>
<td>Complexity</td>
<td>27</td>
</tr>
<tr>
<td>References</td>
<td>29</td>
</tr>
<tr>
<td>EpsNormalize</td>
<td>30</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>30</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>30</td>
</tr>
<tr>
<td>Complexity</td>
<td>30</td>
</tr>
<tr>
<td>References</td>
<td>30</td>
</tr>
<tr>
<td>Equivalent</td>
<td>31</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>31</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>31</td>
</tr>
<tr>
<td>Complexity</td>
<td>31</td>
</tr>
<tr>
<td>References</td>
<td>31</td>
</tr>
<tr>
<td>Intersect</td>
<td>32</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>32</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td>32</td>
</tr>
<tr>
<td>Complexity</td>
<td>33</td>
</tr>
<tr>
<td>Caveats</td>
<td>33</td>
</tr>
<tr>
<td>Invert</td>
<td>34</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>34</td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td>34</td>
</tr>
<tr>
<td>Complexity</td>
<td>34</td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
</tbody>
</table>
# Table of Contents

Map.................................................................................................................................36
  Description....................................................................................................................36
  Usage.............................................................................................................................36
  Example.......................................................................................................................36
  Complexity..................................................................................................................37

Minimize.........................................................................................................................38
  Description....................................................................................................................38
  Usage.............................................................................................................................38
  Examples......................................................................................................................38
  Complexity..................................................................................................................39
  References..................................................................................................................39

Project............................................................................................................................40
  Description....................................................................................................................40
  Usage.............................................................................................................................40
  Examples......................................................................................................................40
  Complexity..................................................................................................................41

Prune...............................................................................................................................42
  Description....................................................................................................................42
  Usage.............................................................................................................................42
  Complexity..................................................................................................................42

Push...............................................................................................................................43
  Description....................................................................................................................43
  Usage.............................................................................................................................43
  Examples......................................................................................................................43
  Complexity..................................................................................................................44

RandEquivalent............................................................................................................46
  Description....................................................................................................................46
  Usage.............................................................................................................................46
  Example.......................................................................................................................46

RandGen.........................................................................................................................47
  Description....................................................................................................................47
  Usage.............................................................................................................................47
  Example.......................................................................................................................47
  Complexity..................................................................................................................48

Relabel...........................................................................................................................49
  Description....................................................................................................................49
  Usage.............................................................................................................................49
  Examples......................................................................................................................49
  Complexity..................................................................................................................51
# Table of Contents

- **Replace** .......................................................... 52
  - Description ....................................................... 52
  - Usage ............................................................... 52
  - Examples .......................................................... 52
  - Complexity ...................................................... 54
  - Caveats ............................................................ 54

- **Reverse** .......................................................... 55
  - Description ....................................................... 55
  - Usage ............................................................... 55
  - Examples .......................................................... 55
  - Complexity ...................................................... 55

- **Reweight** .......................................................... 56
  - Description ....................................................... 56
  - Usage ............................................................... 56
  - Complexity ...................................................... 56

- **RmEpsilon** .......................................................... 57
  - Description ....................................................... 57
  - Usage ............................................................... 57
  - Examples .......................................................... 57
  - Complexity ...................................................... 57
  - References ...................................................... 58

- **ShortestDistance** .................................................. 59
  - Description ....................................................... 59
  - Usage ............................................................... 59
  - Examples .......................................................... 59
  - Complexity ...................................................... 60
  - References ...................................................... 60

- **ShortestPath** ...................................................... 61
  - Description ....................................................... 61
  - Usage ............................................................... 61
  - Examples .......................................................... 61
  - Complexity ...................................................... 61
  - References ...................................................... 62

- **Synchronize** ...................................................... 63
  - Description ....................................................... 63
  - Usage ............................................................... 63
  - Examples .......................................................... 63
  - Complexity ...................................................... 63
  - References ...................................................... 64
# Table of Contents

**OpenFst Library**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topsort</strong></td>
<td>65</td>
</tr>
<tr>
<td>Description</td>
<td>65</td>
</tr>
<tr>
<td>Usage</td>
<td>65</td>
</tr>
<tr>
<td>Examples</td>
<td>65</td>
</tr>
<tr>
<td>Complexity</td>
<td>65</td>
</tr>
<tr>
<td><strong>Union</strong></td>
<td>66</td>
</tr>
<tr>
<td>Description</td>
<td>66</td>
</tr>
<tr>
<td>Usage</td>
<td>66</td>
</tr>
<tr>
<td>Examples</td>
<td>66</td>
</tr>
<tr>
<td>Complexity</td>
<td>67</td>
</tr>
<tr>
<td><strong>Verify</strong></td>
<td>68</td>
</tr>
<tr>
<td>Description</td>
<td>68</td>
</tr>
<tr>
<td>Usage</td>
<td>68</td>
</tr>
<tr>
<td>Complexity</td>
<td>68</td>
</tr>
<tr>
<td><strong>OpenFst Conventions</strong></td>
<td>69</td>
</tr>
<tr>
<td>Object Conventions</td>
<td>69</td>
</tr>
<tr>
<td>Coding Conventions</td>
<td>69</td>
</tr>
<tr>
<td><strong>FST Weight Requirements</strong></td>
<td>70</td>
</tr>
<tr>
<td><strong>OpenFst Advanced Usage</strong></td>
<td>71</td>
</tr>
<tr>
<td>Arc Iterators</td>
<td>71</td>
</tr>
<tr>
<td>Arc Filters</td>
<td>71</td>
</tr>
<tr>
<td>Arcs</td>
<td>72</td>
</tr>
<tr>
<td>Base Fsts</td>
<td>73</td>
</tr>
<tr>
<td>Caching</td>
<td>73</td>
</tr>
<tr>
<td>Command Line Flags</td>
<td>74</td>
</tr>
<tr>
<td>Composition Filters</td>
<td>75</td>
</tr>
<tr>
<td>Expanded Fsts</td>
<td>76</td>
</tr>
<tr>
<td>FST Input/Output</td>
<td>77</td>
</tr>
<tr>
<td>Mappers</td>
<td>78</td>
</tr>
<tr>
<td>Matchers</td>
<td>80</td>
</tr>
<tr>
<td>Mutable Fsts</td>
<td>82</td>
</tr>
<tr>
<td>Natural Orders</td>
<td>82</td>
</tr>
<tr>
<td>Operation Options</td>
<td>83</td>
</tr>
<tr>
<td>Properties</td>
<td>83</td>
</tr>
<tr>
<td>State Iterators</td>
<td>84</td>
</tr>
<tr>
<td>State Queues</td>
<td>85</td>
</tr>
<tr>
<td>State Tables</td>
<td>86</td>
</tr>
<tr>
<td>Symbol Tables</td>
<td>87</td>
</tr>
<tr>
<td>User-defined Fst Arcs and Weights</td>
<td>87</td>
</tr>
<tr>
<td>User-defined Fst Classes</td>
<td>87</td>
</tr>
<tr>
<td>Visitors</td>
<td>88</td>
</tr>
<tr>
<td>Weights</td>
<td>88</td>
</tr>
</tbody>
</table>
Table of Contents

OpenFst Glossary.................................................................................................................................90
OpenFst Library

*OpenFst* is a library for constructing, combining, optimizing, and searching *weighted finite-state transducers* (FSTs). Weighted finite-state transducers are automata where each transition has an input label, an output label, and a weight. The more familiar finite-state acceptor is represented as a transducer with each transition's input and output label equal. Finite-state acceptors are used to represent sets of strings (specifically, *regular* or *rational sets*); finite-state transducers are used to represent binary relations between pairs of strings (specifically, *rational transductions*). The weights can be used to represent the cost of taking a particular transition.

FSTs have key applications in speech recognition and synthesis, machine translation, optical character recognition, pattern matching, string processing, machine learning, information extraction and retrieval among others. Often a weighted transducer is used to represent a probabilistic model (e.g., an *n-gram model*, *pronunciation model*). FSTs can be optimized by *determinization* and *minimization*, models can be applied to hypothesis sets (also represented as automata) or cascaded by finite-state *composition*, and the best results can be selected by *shortest-path* algorithms.

This library was developed by contributors from Google Research and NYU's Courant Institute. It is intended to be comprehensive, flexible, efficient and scale well to large problems. It has been extensively tested. It is an open source project distributed under the Apache license.

- Background Material
- Quick Tour
  - Creating FSTs
  - Accessing FSTs
  - FST Operations
    - Calling Operations
    - Example -- FST Application
    - Available Operations
  - FST Weights
- Advanced Usage
- Conventions
- Glossary
- Download
- Forum
- Contributed and related projects
OpenFst Authors

Principal Contacts:

<table>
<thead>
<tr>
<th>Cyril Allauzen</th>
<th><a href="mailto:allauzen@google.com">allauzen@google.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael Riley</td>
<td><a href="mailto:riley@google.com">riley@google.com</a></td>
</tr>
</tbody>
</table>

Contributors:

<table>
<thead>
<tr>
<th>Boulos Harb</th>
<th><a href="mailto:harb@google.com">harb@google.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Johan Schalkwyk</td>
<td><a href="mailto:johans@google.com">johans@google.com</a></td>
</tr>
<tr>
<td>Masha Shugrina</td>
<td><a href="mailto:shumash@google.com">shumash@google.com</a></td>
</tr>
<tr>
<td>Mehryar Mohri</td>
<td><a href="mailto:mohri@cs.nyu.edu">mohri@cs.nyu.edu</a></td>
</tr>
<tr>
<td>Richard Sproat</td>
<td><a href="mailto:rws@uiuc.edu">rws@uiuc.edu</a></td>
</tr>
<tr>
<td>Wojtek Skut</td>
<td><a href="mailto:wojciech@google.com">wojciech@google.com</a></td>
</tr>
</tbody>
</table>


OpenFst Background Material

The following material is provided as background reading about finite state transducers. However, it is not necessary to read this material before using the OpenFst Library.


The AT&T FSM Library shares many of the same goals as the OpenFst Library.


Additional references are given with the description of individual algorithms.
OpenFst Quick Tour

Below is a brief tutorial on the OpenFst library. After reading this, you may wish to browse the Advanced Usage topic for greater detail and read the library Conventions topic to ensure its correct use.

Finding and Using the Library

The OpenFst library is a C++ template library. From C++, include `<fst/fstlib.h>` in the installation include directory and link to `libfst.so` in the installation library directory. (You may instead use just those include files for the classes and functions that you will need.) All classes and functions are in the `fst` namespace; the examples below assume you are within that namespace for brevity. (Include `<fst/fst-decl.h>` if forward declaration of the public OpenFst classes is needed.)

As an alternative interface, there are shell-level commands in the installation `bin` directory that operate on file representations of FSTs. The command-line flag `--help` will give usage information.

Example FST

The following picture depicts a finite state transducer:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>
```

The initial state is label 0. There can only be one initial state. The final state is 2 with final weight of 3.5. Any state with non-infinite final weight is a final state. There is an arc (or transition) from state 0 to 1 with input label `a`, output label `x`, and weight 0.5. This FST transduces, for instance, the string `ac` to `xz` with weight 6.5 (the sum of the arc and final weights). Note we have assumed the library default Weight type for this description.

Creating FSTs

FSTs can be created with constructors and mutators from C++ or from text files at the shell-level. We will show how to create the above example FST both ways.

Creating FSTs Using Constructors and Mutators From C++

The following code will create our example FST within C++:

```cpp
// A vector FST is a general mutable FST
StdVectorFst fst;

// Adds 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

// Adds two arcs exiting state 0.
// Arc constructor args: ilabel, olabel, weight, dest state ID.
```
We can save this FST to a file with:

```plaintext
fst.Write("binary.fst");
```

### Creating FSTs Using Text Files from the Shell

FSTs can be specified using a text file in the AT&T FSM format. Supporting this format permits interoperability with the AT&T FSM binary tools, which can be downloaded for non-commercial purposes.

We can create the text FST file for our example as follows:

```plaintext
# arc format: src dest ilabel olabel [weight]
# final state format: state [weight]
# lines may occur in any order except initial state must be first line
# unspecified weights default to 0.0 (for the library-default Weight type)
cat >text.fst <
```

The internal representation of an arc label is an integer. We must provide the mapping from symbols to integers explicitly with a *symbol table* file, also in AT&T format:

```plaintext
$ cat >isyms.txt <<EOF
<eps> 0
a 1
b 2
c 3
EOF
$ cat >osyms.txt <<EOF
<eps> 0
x 1
y 2
z 3
EOF
```

You may use any string for a label; you may use any non-negative integer for a label ID. The zero label ID is reserved for the *epsilon* label, which is the empty string. We have included 0 in our table, even though it is not used in our example. Since subsequent FST operations might add epsilons, it is good practice to include a symbol for it.

This text FST must be converted into a binary FST file before it can be used by the OpenFst library.

```plaintext
# Creates binary Fst from text file.
# The symbolic labels will be converted into integers using the symbol table files.
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt text.fst binary.fst
```

# As above but the symbol tables are stored with the FST.

Creating FSTs
OpenFst Library

$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt --keep_isymbols --keep_osymbols \ text.fst binary.fst

If the labels are represented as non-negative integers in the text FST, then the symbol table files can be omitted. In any case, the internal representation of the FST is:

```
0 ---1:1.5 ---2:2/1.5
      3:3/2.5
```

Once a binary FST is created, it can be used with the other shell-level programs. It can be loaded inside C++ with:

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

(See here for more information on FST I/O.)

Accessing FSTs

FSTs can be examined from C++ accessors or from shell-level commands that read the binary files.

Accessing FSTs from C++

Here is the standard representation of an arc:

```
struct StdArc {
    typedef int Label;
    typedef TropicalWeight Weight; // see "FST Weights" below
    typedef int StateId;

    Label ilabel;
    Label olabel;
    Weight weight;
    StateId nextstate;
};
```

Here are some example accesses of an FST:

```
typedef StdArc::StateId StateId;

# Gets the initial state; if == kNoState => empty FST.
StateId initial_state = fst.Start();

# Get state i's final weight; if == Weight::Zero() => non-final.
Weight weight = fst.Final(i);

# Iterates over the FSTs states.
for (StateIterator siter(fst); !siter.Done(); siter.Next())
    StateId state_id = siter.Value();

# Iterates over state i's arcs.
```

Accessing FSTs
for (ArcIterator aiter(fst, i); !aiter.Done(); aiter.Next())
    const StdArc &arc = aiter.Value();

# Iterates over state i's arcs that have input label l (FST must support this - # in the simplest cases, true when the input labels are sorted).
Matcher matcher(fst, MATCH_INPUT);
matcher.SetState(i);
if (matcher.Find(l))
    for (; !matcher.Done(); matcher.Next())
        const StdArc &arc = matcher.Value();

More information on state iterators, arc iterators, and matchers are linked here.

There are various conventions that must be observed when accessing FSTs.

**Printing, Drawing and Summarizing FSTs from the Shell**

The following command will print out an FST in AT&T text format:

```bash
# Print FST using symbol table files.
$ fstprint --isymbols=isyms.txt --osymbols=osyms.txt binary.fst text.fst
```

If the symbol table files are omitted, the FST will be printed with numeric labels unless the symbol tables are stored with the FST (e.g., with `fstcompile --keep_isymbols --keep_osymbols`).

The following command will draw an FST using Graphviz dot format.

```bash
# Draw FST using symbol table files and Graphviz dot:
$ fstdraw --isymbols=isyms.txt --osymbols=osyms.txt binary.fst binary.dot
$ dot -Tps binary.dot >binary.ps
```

Summary information about an FST can be obtained with:

```bash
$ fstinfo binary.fst
fst type                        vector
arc type                        standard
input symbol table              isyms.txt
output symbol table             osyms.txt
# of states                     3
# of arcs                       3
initial state                   0
# of final states               1
# of input/output epsilons      0
# of input epsilons             0
# of output epsilons            0
# of accessible states          3
# of coaccessible states        3
# of connected states           3
# of strongly conn components   3
expanded                        y
mutable                        y
acceptor                        y
input deterministic              y
output deterministic            y
input/output epsilons           n
input epsilons                  n
```
FST Operations

Calling FST Operations

The FST operations can be invoked either at the C++ level or from shell-level commands.

Calling FST Operations from C++

To invoke FST operations from C++, the FST class hierarchy must first be introduced:

The FST interface hierarchy consists of the following abstract class templates:

- **Fst<Arc>:** supports access operations described above
- **ExpandedFst<Arc>:** an Fst that additionally supports NumStates()
- **MutableFst<Arc>:** an ExpandedFst that supports the various mutating operations like AddStates() and SetStart().

Specific, non-abstract FSTs include these class templates:

- **VectorFst<Arc>:** a general-purpose mutable FST
- **ConstFst<Arc>:** a general-purpose expanded, immutable FST
- **ComposeFst<Arc>:** an unexpanded, delayed composition of two FSTs

(See here for how to define your own FST classes if desired.)

These classes are templated on the arc to allow customization. The class StdFst is a typedef for Fst<StdArc>. Similar typedefs exist for all the above templates.

For the state and arc iterators, you will get the greatest efficiency if you specify the most specific FST class as the iterator template argument (e.g., ArcIterator<StdVectorFst> rather than ArcIterator<StdFst> for a known StdVectorFst).

The C++ FST operations come in three general forms:

- **Destructive:** When an operation, like Connect, modifies its input, it has the form:

  ```cpp
  void Connect(MutableFst<Arc> *fst);
  ```

- **Constructive:** When an operation, like Reverse, creates a new expanded Fst, it has the form:
void Reverse(const Fst<Arc>& infst, MutableFst<Arc>* outfst);

- **Delayed**: When an operation, like ComposeFst, creates a lazy-evaluated Fst, it is a new unexpanded Fst class of the form:

  ComposeFst<Arc>(const Fst<Arc>& fst1, const Fst<Arc>& fst2);

Delayed Fsts have constant time-class constructors. When components of delayed Fsts are accessed through the Fst interface, the automaton is built dynamically, just enough to respond to the accesses requested. It is important that the object access conventions are observed for correct operation.

Several operations, like Union, come in more than one of the above forms.

### Calling FST Operations from the Shell

The shell-level FST operations typically read one or more input binary FST files, call internally the corresponding C++ operation and then write an output binary FST file. If the output file is omitted, standard output is used. If the input file is also omitted (unary case) or is "-", then standard input is used. Specifically, they have the form:

- **Unary Operations**:

  ```
  fstunaryop in.fst out.fst
  fstunaryop <in.fst >out.fst
  ```

- **Binary Operations**:

  ```
  fstbinaryop in1.fst in2.fst out.fst
  fstbinaryop - in2.fst <in1.fst >out.fst
  ```

### Example Use: FST Application

One of the most useful finite-state operations is *composition*, which produces the relational composition of two transductions. It can be used, for example, to apply a transduction to some input:

#### FST Application from C++

```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);
```
FST Application from the Shell

# 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

Available FST Operations

Click on operation name for additional information.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArcSort</td>
<td>ArcSort(&amp;A, compare);</td>
<td>sorts arcs using compare function object</td>
</tr>
<tr>
<td></td>
<td>ArcSortFst&lt;Arc, Compare&gt;(A, compare);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fstarcsort [--sort_type=$type] in.fst out.fst</td>
<td></td>
</tr>
<tr>
<td>Closure</td>
<td>Closure(&amp;A, type);</td>
<td>A* = {ε} ∪ A ∪ AA ∪ ....</td>
</tr>
<tr>
<td></td>
<td>ClosureFst&lt;Arc&gt;(A, type);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fstclosure in.fst out.fst</td>
<td></td>
</tr>
<tr>
<td>Compose</td>
<td>Compose(A, B, &amp;C);</td>
<td>composition of binary relations A and B</td>
</tr>
<tr>
<td></td>
<td>ComposeFst&lt;Arc&gt;(A, B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fstcompose a.fst b.fst out.fst</td>
<td></td>
</tr>
<tr>
<td>Concat</td>
<td>Concat(&amp;A, B);</td>
<td>contains the strings in A followed by B</td>
</tr>
<tr>
<td></td>
<td>Concat(A, &amp;B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConcatFst&lt;Arc&gt;(A,B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fstconcat a.fst b.fst out.fst</td>
<td></td>
</tr>
<tr>
<td>Connect</td>
<td>Connect(&amp;A);</td>
<td>removes states and arcs not on a path from the start to a final state</td>
</tr>
<tr>
<td></td>
<td>fstconnect in.fst out.fst</td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>Decode(&amp;A, encoder);</td>
<td>decodes previously encoded Fst</td>
</tr>
<tr>
<td></td>
<td>DecodeFst(A, encoder);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fstencode --decode in.fst encoder out.fst</td>
<td></td>
</tr>
<tr>
<td>Determinize</td>
<td>Determinize(A, &amp;B);</td>
<td>creates equiv. FST with no state with two arcs with the same input label</td>
</tr>
<tr>
<td></td>
<td>DeterminizeFst&lt;Arc&gt;(A);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fstdetermine in.fst out.fst</td>
<td></td>
</tr>
<tr>
<td>Difference</td>
<td>Difference(A, B, &amp;C);</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Command</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>DifferenceFst</strong>&lt;sup&gt;(Arc)&lt;/sup&gt;(A, B);</td>
<td>unweighted DifferenceFst(A, B);</td>
<td>fstdifference a.fsa b.dfa out.fsa</td>
</tr>
<tr>
<td><strong>Encode</strong></td>
<td>combines input labels with output labels and/or weights into new input labels</td>
<td>Encode(&amp;A, encoder);</td>
</tr>
<tr>
<td><strong>EncodeFst</strong>&lt;sup&gt;(Arc)&lt;/sup&gt;(A, encoder);</td>
<td></td>
<td>fstencode --encode_labels [--encode_weights] in.fst encoder out.fst</td>
</tr>
<tr>
<td><strong>EpsNormalize</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>creates equiv. FST with any input (output) epsilons at path ends</td>
<td>EpsNormalize(A, &amp;B, type);</td>
</tr>
<tr>
<td><strong>Equivalent</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>determines if acceptors A and B accept the same strings with the same weights</td>
<td>Equivalent(A, B)</td>
</tr>
<tr>
<td><strong>Intersect</strong></td>
<td>contains strings both in A and B</td>
<td>Intersect(A, B, &amp;C);</td>
</tr>
<tr>
<td><strong>Invert</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>inverse binary relation; exchanges input and output labels</td>
<td>Invert(&amp;A);</td>
</tr>
<tr>
<td><strong>Map</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>transforms arcs in an FST</td>
<td>Map(&amp;A, mapper);</td>
</tr>
<tr>
<td><strong>Minimize</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>transforms to equiv. deterministic FSA with fewest states and arcs</td>
<td>Minimize(&amp;A);</td>
</tr>
<tr>
<td><strong>Project</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>creates acceptor of just the input or output strings</td>
<td>Project(&amp;A, type);</td>
</tr>
<tr>
<td><strong>Prune</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>removes paths outside a threshold of best path</td>
<td>Prune(&amp;A, threshold);</td>
</tr>
<tr>
<td><strong>Push</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>creates equiv. FST pushing weights and/or output labels toward initial or final states</td>
<td>Push(A, Type&gt;(&amp;A, flags);</td>
</tr>
<tr>
<td><strong>RandEquivalent</strong>&lt;sup&gt;+&lt;/sup&gt;</td>
<td>checks if transducers A and B transduce the same randomly-generated string pairs with the same weights</td>
<td>RandEquivalent(A, B, npath);</td>
</tr>
</tbody>
</table>
### FST Weights

An arc weight in an FST gives the cost of taking that transition. The OpenFst library supports multiple types of weights -- in fact, any C++ class that meets various properties can be used as the `Weight` type specified in the Arc template parameter of an Fst. Several `Weight` types are predefined in the library that will normally meet your needs. Among a weight's properties, it must have associated binary operations $\oplus$ and $\otimes$ and elements 0 and 1. These are implemented by a `Weight` type with functions `Plus(x, y)` and `Times(x, y)`.
and static member functions `Zero()` and `One()`, respectively. These must form a semiring; see here for a further description of the constraints on these operations and other properties of weights. ⊕ is used to combine the weight of two identically labeled alternative paths, while ⊗ is used to combine weights along a path or when matching paths in composition or intersection. A state is final if and only its final weight is not equal to 0. A transition with weight 1 is, in essence, "free". A path with weight 0 is not allowed (since such paths present technical problems with some algorithms).

The following are some useful weight types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Set</th>
<th>⊕  (Plus)</th>
<th>⊗  (Times)</th>
<th>0 (Zero)</th>
<th>1 (One)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>{0, 1}</td>
<td>∨</td>
<td>∧</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Real</td>
<td>[0, ∞)</td>
<td>+</td>
<td>*</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Log</td>
<td>[-∞, ∞)</td>
<td>-log(e^x + e^y)</td>
<td>+</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>Tropical</td>
<td>[-∞, ∞)</td>
<td>min</td>
<td>+</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

The `boolean` weight type is used for the familiar unweighted automata (but see `tropical` below). The `real` weight type is appropriate when the transition weights represent probabilities. The `log` weight type is appropriate when the transition weights represent negative log probabilities (more numerically stable than the isomorphic, under log(), `real` weight type). The `tropical` weight type is appropriate for shortest path operations and is identical to the log except it uses `min` for the `Plus` operation.

The OpenFst library predefines `TropicalWeight` and `LogWeight` as well as the corresponding `StdArc` and `LogArc`. These weight classes represent their weight in a single precision float that is a constructor argument. That float can be directly accessed with member function `Value()`. For unweighted automata, it is convenient and customary in this library to use `TropicalWeight` restricted to `Zero` and `One`. `StdArc` is the default arc type for the FST binaries. The `Boolean` and `Real` weight types not currently pre-defined. See here for all pre-defined weight types.

From the shell-level, the FST arc type can be specified to `fstcompile` with the `--arc_type` flag; `StdArc` is the default.

(See here for how to your define your own FST arcs and weights if desired.)
**ArcSort**

**Description**

This operation sorts the arcs in an FST per state.

At the C++ level, the sort order is determined by a function object `compare` of type `Compare`. Comparison function objects `ILabelCompare` and `OLabelCompare` are provided by the library. In general, `Compare` must meet the requirements for an STL sort comparison function object. It must also have a member `Properties(uint64)` that specifies the known properties of the sorted Fst; it takes as argument the input Fst's known properties before the sort.

At the shell level, the sort order is determined by the `-sort_type` flag, which can have values `ilabel` and `olabel`.

**Usage**

```cpp
template <class Arc, class Compare>
void ArcSort(MutableFst<Arc> *fst, Compare compare);

template <class Arc, class Compare> ArcSort<Arc, Compare>::
ArcSortFst(const Fst<Arc> &fst, const Compare &compare);

fstarcsort [--opts] a.fst out.fst
    --sort_type: ilabel (def) | olabel
```

**Examples**

**A:**

![Diagram of input label sort of A]

**Input Label Sort of A:**

![Diagram of input label sort of A]

ArcSort(&A, ILabelCompare<Arc>());
ArcSortFst<Arc, ILabelCompare<Arc> >(A, ILabelCompare<Arc>());
fstarcsort --sort_type=ilabel a.fst out.fst

**Output Label Sort of A:**

```
0 2 3 4 1
```

ArcSort(A, OLabelCompare<Arc>());
ArcSortFst<Arc, OLabelCompare<Arc> >(A, OLabelCompare<Arc>());
fstarcsort --sort_type=olabel a.fst out.fst

**Complexity**

**ArcSort:**

- Time: $O(V + D \log D)$
- Space: $O(D)$

where $V =$ # of states and $D =$ maximum out-degree.

**ArcSortFst:**

- Time: $O(v + d \log d)$
- Space: $O(d)$

where $v =$ # of states visited, $d =$ maximum out-degree of the states visited. Constant time and space to visit an input state or arc is assumed and exclusive of caching.
Closure

Description

This operation computes the concatenative closure. If \( A \) transduces string \( x \) to \( y \) with weight \( a \), then the closure transduces \( x \) to \( y \) with weight \( a \), \( xx \) to \( yy \) with weight \( a \otimes a \), \( xxx \) to \( yyy \) with weight \( a \otimes a \otimes a \), etc. If \( \text{closure_type} \) is \( \text{CLOSURE_STAR} \), then the empty string is transduced to itself with weight 1 as well.

Usage

```cpp
enum ClosureType { CLOSURE_STAR, CLOSURE_PLUS };

template<class Arc>
void Closure(MutableFst<Arc> *fst, ClosureType type);
template <class Arc> ClosureFst<Arc>::ClosureFst(const Fst<Arc> &fst, ClosureType type);
```

Examples

A:

```
A
```

A*:

```
A*
```

```
Closure(&A, CLOSURE_STAR);
ClosureFst<Arc>(A, CLOSURE_STAR);
```

A+:

```
A+
```

```
Closure(&A, CLOSURE_PLUS);
ClosureFst<Arc>(A, CLOSURE_PLUS);
```

```
Complexity

Closure:

- Time: $O(V)$
- Space: $O(V)$

where $V = \#$ of states and $E = \#$ of arcs.

ClosureFst:

- Time: $O(v)$
- Space: $O(v)$

where $v = \#$ of states visited, $e = \#$ of arcs visited. Constant time to visit an input state or arc is assumed and exclusive of caching.
### Compose

#### Description

This operation computes the composition of two transducers. If \( A \) transduces string \( x \) to \( y \) with weight \( a \) and \( B \) transduces \( y \) to \( z \) with weight \( b \), then their composition transduces string \( x \) to \( z \) with weight \( a \odot b \).

The output labels of the first transducer or the input labels of the second transducer must be sorted (or the FSTs otherwise support appropriate matchers). The weights need to form a commutative semiring (valid for \( \text{TropicalWeight} \) and \( \text{LogWeight} \) for instance).

Versions of this operation (not all shown here) accept options that allow choosing the matcher, composition filter, state table and, when delayed, the caching behavior used by composition.

#### Usage

```cpp
template <class Arc>
void Compose(const Fst<Arc> &fst1, const Fst<Arc> &fst2, MutableFst<Arc> *ofst);
template <class Arc> ComposeFst<Arc>::ComposeFst(const Fst<Arc> &fst1, const Fst<Arc> &fst2);
nfstcompose [--opts] a.fst b.fst out.fst
   --connect: Trim output (def: true)
```

#### Examples

**A:**

![Diagram of A]

**B:**

![Diagram of B]

**A \circ B:**

![Diagram of A \circ B]
Compose(A, B, &C);
ComposeFst<Arc>(A, B);
fstcompose a.fst b.fst out.fst

**Complexity**

Assuming the first FST is unsorted and the second is sorted:

**Compose:**

- Time: $O(V_1 V_2 D_1 (\log D_2 + M_2))$
- Space: $O(V_1 V_2 D_1 M_2)$

where $V_i =$ # of states, $D_i =$ maximum out-degree and $M_i =$ maximum multiplicity for the $ith$ FST.

**ComposeFst:**

- Time: $O(v_1 v_2 d_1 (\log d_2 + m_2))$,
- Space: $O(v_1 v_2)$

where $v_i =$ # of states visited, $d_i =$ maximum out-degree, and $m_i =$ maximum multiplicity of the states visited for the $ith$ FST. Constant time and space to visit an input state or arc is assumed and exclusive of caching.

**Caveats**

Compose and fstcompose trim their output, ComposeFst does not (since it is a delayed operation).

The efficiency of composition can be strongly affected by several factors:

- the choice of which transducer is sorted - prefer sorting the FST that has the greater average out-degree;
- the amount of non-determinism;
- the presence and location of epsilon transitions - avoid epsilon transitions on the output side of the first transducer or the input side of the second transducer or prefer placing them later in a path since they delay matching and can introduce non-coaccessible states and transitions.
Concat

Description

This operation computes the concatenation \((product)\) of two FSTs. If \(A\) transduces string \(x\) to \(y\) with weight \(a\) and \(B\) transduces string \(w\) to \(v\) with weight \(b\), then their concatenation transduces string \(xw\) to \(yv\) with weight \(a \otimes b\).

Usage

\[
\begin{align*}
\text{template } \langle \text{class Arc}\rangle \\
\text{void Concat(MutableFst}<\text{Arc}\rangle \ *\text{fst1, const Fst}<\text{Arc}\rangle \ &\text{fst2}); \\
\text{template } \langle \text{class Arc}\rangle \\
\text{void Concat(const Fst}<\text{Arc}\rangle \ &\text{fst1, MutableFst}<\text{Arc}\rangle \ *\text{fst2}); \\
\text{template } \langle \text{class Arc}\rangle \ Conc\text{atFst}<\text{Arc}>:: \\
\text{Conc}\text{atFst(const Fst}<\text{Arc}\rangle \ &\text{fst1, const Fst}<\text{Arc}\rangle \ &\text{fst2}); \\
\text{fstconcat } a.\text{fst } b.\text{fst } \text{out}.\text{fst}
\end{align*}
\]

Examples

**A:**

![Diagram of FST A]

**B:**

![Diagram of FST B]

**AB:**

![Diagram of FST AB]

\[
\text{Concat}(\&A, B); \\
\text{Concat}(A, \&B); \\
\text{ConcatFst}<\text{Arc}>(A, B); \\
\text{fstconcat } a.\text{fst } b.\text{fst } \text{out}.\text{fst}
\]
Complexity

Concat (&A, B):

- Time: $O(V_i + V_2 + E_2)$
- Space: $O(V_i + V_2 + E_2)$

where $V_i = \#$ of states and $E_i = \#$ of arcs of the $ith$ FST.

Concat (A, &B):

- Time: $O(V_i + E_i)$
- Space: $O(V_i + E_i)$

where $V_i = \#$ of states and $E_i = \#$ of arcs of the $ith$ FST.

ConcatFst:

- Time: $O(v_i + e_i + v_2 + e_2)$
- Space: $O(v_i + v_2)$

where $v_i = \#$ of states visited and $e_i = \#$ of arcs visited of the $ith$ FST. Constant time and space to visit an input state or arc is assumed and exclusive of caching.

Caveats

When concatenating a large number of FSTs, one should use the prepending Concat (&A, B) instead of the appending Concat (A, &B) since the total cost of the concatenation operations would be linear in the sum of the size of the input FSTs for the former instead of quadratic for the latter.
Connect

Description

This operation trims an FST, removing states and arcs that are not on successful paths.

Usage

```cpp
template<class Arc>
void Connect(MutableFst<Arc> *fst);

fstconnect a.fst out.fst
```

Examples

A:

![Diagram of A]

Connect of A:

![Diagram of Connect(A)]

```cpp
Connect(&A);
fstconnect a.fst out.fst
```

Complexity

Connect:

- Time: $O(V + E)$
- Space: $O(V + E)$

where $V = \#$ of states and $E = \#$ of arcs.
Determinize

Description

This operation determinizes a weighted transducer. The result will be an equivalent FST that has the property that no state has two transitions with the same input label. For this algorithm, epsilon transitions are treated as regular symbols (cf. RmEpsilon).

The transducer must be functional. The weights must be (weakly) left divisible (valid for TropicalWeight and LogWeight for instance) and zero-sum-free.

Usage

```
template <class Arc>
void Determinize(const Fst<Arc> &fst, MutableFst<Arc> *ofst);
template <class Arc> DeterminizeFst<Arc>::
DeterminizeFst(const Fst<Arc> &fst);
```

Examples

A:

```
Determineze of A:
```

Complexity

Determinize:

- Determinizable: exponential (polynomial in the size of the output)
- Non-determinizable: does not terminate

DeterminizeFst:
• Determinizable: *exponential (polynomial in the size of the output)*
• Non-determinizable: *does not terminate*

The determinizable automata include all unweighted and all acyclic input.

**References**

Difference

Description

This operation computes the difference between two FSAs. Only strings that are in the first automaton but not in second are retained in the result.

The first argument must be an acceptor; the second argument must be an unweighted, epsilon-free, deterministic acceptor. The output labels of the first acceptor or the input labels of the second acceptor must be sorted (or the FSTs otherwise support appropriate matchers).

Usage

```cpp
template <class Arc>
void Difference(const Fst<Arc> &ifsa1, const Fst<Arc> &ifsa2, MutableFst<Arc> *ofsa);

template <class Arc> DifferenceFst<Arc>::
DifferenceFst(const Fst<Arc> &fsa1, const Fst<Arc> &fsa2);

fstdifference [--opts] a.fsa b.fsa out.fsa
   --connect: Trim output (def: true)
```

Examples

A:

![A diagram](image1)

B:

![B diagram](image2)

A - B:

![A - B diagram](image3)

Difference(A, B, &C);
DifferenceFst<Arc>(A, B);
fstdifference a.fsa b.fsa out.fsa
Complexity

Same as Compose.

Caveats

Same as Compose.


# Encode/Decode

## Description

The **Encode** operation allows the representation of a weighted transducer as a weighted automaton, an unweighted transducer or an unweighted automaton by considering the pairs (input label, output), the pairs (input label, weight) or the triple (input label, output label, weight) as a single label depending on the value of the encode flags: `kEncodeLabels`, `kEncodeWeights` or `kEncodeLabels|kEncodeWeights`. The encoding of each pair or triple of labels and/or weights as a unique key is performed by an `EncodeMapper` object.

The **Decode** operation takes as input an encoded FST and the corresponding `EncodeMapper` object and reverts the encoding.

## Usage

### EncodeMapper

```cpp
static const uint32 kEncodeLabels = 0x0001;
static const uint32 kEncodeWeights = 0x0002;
static const uint32 kEncodeFlags = 0x0003;
enum EncodeType { ENCODE = 1, DECODE = 2 };
template <class Arc> EncodeMapper<Arc>::EncodeMapper(uint32 flags, EncodeType type);
```

### Encode

```cpp
template <class Arc>
void Encode(MutableFst<Arc> *fst, EncodeMapper<Arc> *encoder);
template <class Arc>
EncodeFst<Arc>::EncodeFst<Arc>(const Fst<Arc> &fst, EncodeMapper<Arc> *encoder);
```

### Decode

```cpp
template <class Arc>
void Decode(MutableFst<Arc> *fst, const EncodeMapper<Arc> &encoder);
template <class Arc>
DecodeFst<Arc>::DecodeFst<Arc>(const Fst<Arc> &fst, EncodeMapper<Arc> *encoder);
```

### fstencode

```bash
fstencode [--encode_labels] [--encode_weights] in.fst encoder out.fst
fstencode --decode in.fst encoder out.fst
```

## Example
A:

Encode(&A, &encoder):

<table>
<thead>
<tr>
<th>encode_flags</th>
<th>kEncodeLabels</th>
<th>kEncodeWeights</th>
</tr>
</thead>
<tbody>
<tr>
<td>$flags</td>
<td>--encode_labels</td>
<td>--encode_weights</td>
</tr>
<tr>
<td>result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EncodeMapper<Arc> encoder(encode_flags, ENCODE);
Encode(&A, &encoder);
EncodeFst<Arc>(A, &encoder);

fstencode $flags a.fst encoder b.fst

Decode(&A, encoder):

Decode(&A, encoder);
DecodeFst<Arc>(A, encoder);
fstencode --decode a.fst encoder b.fst

Example
Complexity

Encode, Decode:

- Time: $O(V + E)$
- Space: $O(1)$

where $V =$ # of states, and $E =$ # of transitions in the input FST.

EncodeFst, DecodeFst:

- Time: $O(v + e)$
- Space: $O(1)$

where $v =$ # of states visited, $e =$ # of arcs visited Constant time and to visit an input state or arc is assumed and exclusive of caching.
EpsNormalize

Description

Returns an equivalent FST that is epsilon-normalized. An acceptor is epsilon-normalized if it is epsilon-removed. A transducer is input epsilon-normalized if additionally if on each path any epsilon input label follows all non-epsilon input labels. Output epsilon-normalized is defined similarly.

The input FST needs to be functional.

Usage

```
enum EpsNormalizeType { EPS_NORM_INPUT, EPS_NORM_OUTPUT };

template<class Arc>
void EpsNormalize(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, EpsNormalizeType type);
```

```
fstepsnormalize [--opts] a.fst out.fst
    --eps_norm_output: Normalize output epsilons (def: false)
```

Examples

A:

```
(Input) Epsilon Normalize of A:
```

```
Epsnormalize(A, &B, EPS_NORM_INPUT);
fstepsnormalize a.fst out.fst
```

Complexity

TBA

References

Equivalent

Description

This operation determines if two epsilon-free deterministic weighted acceptors are equivalent.

Usage

```
template <class Arc>
bool Equivalent(const Fst<Arc> &fst1,
    const Fst<Arc> &fst2,
    double delta = kDelta);
```

Example

TBA

Complexity

- **Equivalent**
  - Time:
    - Unweighted: quasi-linear, i.e. $O(n \cdot G(n))$
    - Weighted: complexity of unweighted + complexity of shortest-distance
  - Space: quasi-linear, i.e. $O(n \cdot G(n))$

where $n = V_1 + V_2$ with $V_i = \#$ of states and $G(n)$ is a very slowly growing function that can be approximated by 4 by all practical purposes.

References

Intersect

Description

This operation computes the intersection (Hadamard product) of two FSAs. Only strings that are in both automata are retained in the result.

The two arguments must be acceptors. One of the arguments must be label-sorted (or otherwise support an appropriate matcher). The weights need to form a commutative semiring (valid for TropicalWeight and LogWeight for instance).

Versions of this operation (not all shown here) accept options that allow choosing the matcher, composition filter, state table and, when delayed, the caching behaviour used by intersection.

Usage

```
template <class Arc> void Intersect(const Fst<Arc> &ifsa1, const Fst<Arc> &ifsa2, MutableFst<Arc> *ofsa);
template <class Arc> class IntersectFst<Arc>::IntersectFst(const Fst<Arc> &fsa1, const Fst<Arc> &fsa2);
```

```
fstintersect [--opts] a.fsa b.fsa out.fsa
--connect: Trim output (def: true)
```

Examples

A:

```
0 \rightarrow [0,2.5] 1 \rightarrow [1,0]
```

B:

```
0 \rightarrow [0,3] 1 \rightarrow [1,2.5] d/1.5
```

A $\cap$ B:
Intersect(A, B, &C);
IntersectFst<Arc>(A, B);
fstintersect a.fsa b.fsa out.fsa

Complexity

Same as Compose.

Caveats

Same as Compose.
Invert

Description

This operation inverts the transduction corresponding to an FST by exchanging the FST's input and output labels.

Usage

```cpp
template<class Arc>
void Invert(MutableFst<Arc> *fst);
template <class Arc> InvertFst<Arc>::
InvertFst(const Fst<Arc> &fst);
fstinvert a.fst out.fst
```

Examples

A:

![Diagram of A](image)

A⁻¹:

![Diagram of A⁻¹](image)

Invert(&A);
InvertFst<Arc>(A);
fstinvert a.fst out.fst

Complexity

Invert:

- Time: $O(V + E)$
- Space: $O(1)$

where $V = \# \text{ of states}$ and $E = \# \text{ of arcs}$.

InvertFst:

- Time: $O(v + e)$
- Space: $O(1)$
where $v =$ # of states visited, $e =$ # of arcs visited. Constant time and space to visit an input state or arc is assumed and exclusive of caching.
Map

Description

This operation transforms each arc and final state in the input FST. The transformation is specified by a function object called a mapper.

For instance, RmWeightMapper replaces the weight of every arc and final state by 1.

A list of available mappers and instructions on how to create them are given here.

Usage

```cpp
template <class Arc, class Mapper>
Map(MutableFst<Arc> *fst, Mapper *mapper);

template <class Arc, class Mapper>
Map(MutableFst<Arc> *fst, Mapper mapper);

template <class Arc, class Mapper>
Map(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, Mapper *mapper);

template <class Arc, class Mapper>
Map(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, Mapper mapper);

template <class Arc, class Mapper>
MapFst<Arc>::MapFst(const Fst<A> &fst, Mapper *mapper);

template <class Arc, class Mapper>
MapFst<Arc>::MapFst(const Fst<A> &fst, const Mapper &mapper);
```

Example

A:

```
Map(&A, RmWeightMapper());
```
Map(A, RmWeightMapper<StdArc>());
Map(A, &B, RmWeightMapper<StdArc>());
MapFst B(A, RmWeightMapper<StdArc>());

fstmap --mapper_type=rmweight a.fst b.fst

**Complexity**

**Map:**

- Time: $O(c^*(V + E))$
- Space: $O(m)$

where $V =$ # of states, $E =$ # of arcs in input FST, $c =$ cost of processing one arc by the mapper and $m =$ total memory usage for the mapper.

**MapFst:**

- Time: $O(c^*(v + e))$
- Space: $O(m)$

where $v =$ # of visited states, $e =$ # of visited arcs in input FST, $c =$ cost of processing one arc by the mapper and $m =$ total memory usage for the mapper. Constant time and space to visit an input state or arc is assumed and exclusive of caching.

For instance in the case of **RmWeightMapper**, we have $c = O(1)$ and $m = O(1)$.  

---

**Example** 37
Minimize

Description

This operation performs the minimization of deterministic weighted automata and transducers.

If the input FST $A$ is an automaton (acceptor), this operation produces the minimal automaton $B$ equivalent to $A$, i.e. the automata with a minimal number of states that is equivalent to $A$.

If the input FST $A$ is a transducer, this operation internally builds an equivalent transducer with a minimal number of states. However, this minimality is obtained by allowing transition having strings of symbols as output labels, this known in the literature as a real-time transducer. Such transducers are not directly supported by the library. By default, Minimize will convert such transducer by expanding each string-labeled transition into a sequence of transitions. This will results in the creation of new states, hence losing the minimality property. If a second output argument is given to Minimize, then the first output $B$ will be the minimal real-time transducer with each strings that is the output label of a transition being mapped to a new output symbol, the second output transducer $C$ represents the mapping between new output labels and old output labels. Hence, we will have that $A$ is equivalent to $B \circ C$.

Usage

```cpp
template<class Arc>
void Minimize(MutableFst<Arc> *fst, MutableFst<Arc> *sfst = 0);
```

```
fstminimize in.fst [out1.fst [out2.fst]]
```

Examples

Acceptor minimization

```
Minimize(A, &B);
fstminimize a.fst b.fst
```

Transducer minimization

TBA
Minimize(A, &B);
fstminimize a.fst b.fst

Minimize(A, &B, &C);
fstminimize a.fst b.fst c.fst

Complexity

Minimize

- Time:
  - Acyclic: $O(E)$
  - Unweighted: $O(E \log V)$
  - Weighted: complexity of shortest distance + complexity of unweighted minimization

where $V = \# \text{ of states and } E = \# \text{ of transitions in the input Fst.}$

References

Project

Description

This operation projects an FST onto its domain or range by either copying each arc’s input label to its output label or vice versa.

Usage

```c
enum ProjectType { PROJECT_INPUT, PROJECT_OUTPUT };

template<class Arc>
void Project(MutableFst<Arc> *fst, ProjectType type);

template <class Arc> ProjectFst<Arc>::
ProjectFst(const Fst<Arc> &fst, ProjectType type);

fstproject [--opts] a.fst out.fst
   --project_output: Project on output (def false)
```

Examples

**A:**

\[ \pi_1 (A) : \]

```
0 \xrightarrow{a \gamma 1} 1 \xrightarrow{c \gamma 5} 2 \xrightarrow{f \gamma 9} \_ 2
```

Project(&A, PROJECT_INPUT);
ProjectFst<Arc>(A, PROJECT_INPUT);
fstproject a.fst out.fst

\[ \pi_2 (A) : \]

```
0 \xrightarrow{a \gamma 1} 1 \xrightarrow{c \gamma 5} 2 \xrightarrow{f \gamma 9} \_ 2
```

Project(&A, PROJECT_OUTPUT);
ProjectFst<Arc>(A, PROJECT_OUTPUT);
fstproject --project_output=true a.fst out.fst

Complexity

**Project:**

- Time: $O(V + E)$
- Space: $O(1)$

where $V = \# \text{ of states}$ and $E = \# \text{ of arcs}$.

**ProjectFst:**

- Time: $O(v + e)$
- Space: $O(1)$

where $v = \# \text{ of states visited}$, $e = \# \text{ of arcs visited}$ Constant time and to visit an input state or arc is assumed and exclusive of caching.
Prune

Description

This operation deletes states and arcs in the input FST that do not belong to a successful path whose weight is no more (w.r.t the natural the natural semiring order) than the threshold $t \odot$-times the weight of the shortest path in the input FST.

Weights need to be commutative and have the path property. Both destructive and constructive implementations are available.

Usage

```cpp
template <class Arc>
void Prune(MutableFst<Arc> *fst, typename Arc::Weight threshold);
template <class Arc>
void Prune(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, typename Arc::Weight threshold);
```

```bash
fstprune [--opts] in.fst out.fst
  --weight: type = string, default = ""
`Weight parameter
A:
```

```
Prune of A with threshold 3:
```

Prune($A$, 3);
Prune($A$, $B$, 3);
fstprune --weight=3 a.fst out.fst

Complexity

Prune:

- Time: $O(V \log V + E)$
- Space: $O(V + E)$

where $V = \# \text{ of states and } E = \# \text{ of arcs.}$
Push

Description

This operation produces an equivalent transducer by pushing the weights and/or the labels towards the initial state or toward the final states.

Weight pushing

When pushing weights towards the initial state, the sum of the weight of the outgoing transitions and final weight at any non-initial state is equal to 1 in the resulting machine. When pushing weights towards the final states, the sum of the weight of the incoming transitions at any state is equal to 1.

Weight needs to be left distributive when pushing towards the initial state and right distributive when pushing towards the final states.

Label pushing

Pushing labels towards the initial state consists in minimizing at every state the length of the longest common prefix of the output labels of the outgoing paths. Pushing labels towards the final states consists in minimizing at every state the length of the longest common suffix of the output labels of the incoming paths.

Usage

```cpp
const uint32 kPushWeights = 0x0001;
const uint32 kPushLabels = 0x0002;

enum ReweightType { REWEIGHT_TO_INITIAL, REWEIGHT_TO_FINAL };

template <class Arc, ReweightType rtype>
void Push(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, uint32 ptype) {
    fstpush [--opts] a.fst out.fst
    --to_final: type = bool, default = false
    Push/reweight to final (vs. to initial) states
    --push_labels: type = bool, default = false
    Push output labels
    --push_weights: type = bool, default = false
    Push weights
}
```

Examples

A with TropicalWeight:

```
Push
```
**Push weights of A to initial state:**

![Diagram of push weights to initial state]

Push\(<\text{Arc, REWEIGHT_TO_INITIAL}>\)(A, &B, kPushWeights);
fstpush --push_weights a.fst out.fst

**Push labels of A to initial state:**

![Diagram of push labels to initial state]

Push\(<\text{Arc, REWEIGHT_TO_INITIAL}>\)(A, &B, kPushLabels);
fstpush --push_labels a.fst out.fst

**Push weights and labels of A to final states**

![Diagram of push weights and labels to final states]

Push\(<\text{Arc, REWEIGHT_TO_FINAL}>\)(A, &B, kPushWeights|kPushLabels);
fstpush --push_weights --push_labels --to_final a.fst out.fst

**Complexity**

**Push:**

- **Weight pushing:**
  - Time:
    - Acyclic: \(O(V + E)\)
    - Tropical semiring: \(O(V \log V + E)\)
    - General: exponential
  - Space: \(O(V + E)\)
- **Label pushing:**
  - Time: polynomial
  - Space: \(O(V^2 + E)\)
where $V = \# \text{ of states}$ and $E = \# \text{ of arcs.}$
RandEquivalent

Description

This operation test is two FSTs are equivalent by randomly generating \( N \) paths alternatively in each of the two FSTs.

For each randomly generated path, the algorithm computes for each of the two FSTs the sum of the weights of all the successful paths sharing the same input and output labels as the considered randomly generated path and checks that these two values are within \( \delta \).

The random generation can be specified in the same as for the RandGen operation.

Usage

```cpp
template <class Arc>
bool RandEquivalent(const Fst<Arc> &fst1, const Fst<Arc> &fst2,
                    ssize_t num_paths, float delta = kDelta,
                    int seed = time(0), int path_length = INT_MAX);
```

```cpp
template<class Arc, class ArcSelector>
bool RandEquivalent(const Fst<Arc> &fst1, const Fst<Arc> &fst2,
                    ssize_t num_paths, float delta,
                    const RandGenOptions<ArcSelector> &opts);
```

```
fstequivalent --random 
[-npath=$npaths$] 
[--delta=$delta$] 
[--seed=$seed$] 
[--max_length=$max_length$] in1.fst in2.fst
```

Example

![Diagram of FSTs A and B](image)

```cpp
RandEquivalent(A, B, npaths, kDelta, RandGenOptions<ArcSelector>();
```

```cpp
fstequivalent --random --npath=$npaths$ a.fst b.fst
```

<table>
<thead>
<tr>
<th>ArcSelector</th>
<th>npaths</th>
<th>RandEquivalent(A, B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniformArcSelector</td>
<td>10</td>
<td>false</td>
</tr>
<tr>
<td>LogProbArcSelector</td>
<td>10</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>false</td>
</tr>
</tbody>
</table>
RandGen

Description

This operation randomly generates a set of successful paths in the input FST. The operation relies on an ArcSelector object for randomly selecting an outgoing transition at a given state in the input FST. The default arc selector, UniformArcSelector, randomly selects a transition using the uniform distribution. LogProbArcSelector randomly selects a transition w.r.t. the weights treated as negative log probabilities after normalizing for the total weight leaving the state. In all cases, finality is treated as a transition to a super-final state.

Usage

```
template <class Arc> 
void RandGen(const Fst<Arc> &ifst, MutableFst<Arc> *ofst);

template <class Arc, class ArcSelector> 
void RandGen(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, 
             const RandGenOptions<ArcSelector> &opts);
```

```
$ fstrandgen --max_length=3 --npath=10 --seed=42 a.fst b.fst 
```

Example

A:

```
Relabel(A, &B) using UniformArcSelector:
```

```
0 \rightarrow f \rightarrow 1 \rightarrow g \rightarrow 2
```

```
RandGen(A, &B);
RandGen(A, &B, RandGenOptions<UniformArcSelector<Arc> >(UniformArcSelector<Arc>(())));
```

```
Relabel(A, &B) using LogProbArcSelector:
```

```
0 \rightarrow a \rightarrow 1 \rightarrow f \rightarrow 2
```

RandGen
OpenFst Library

RandGen(A, &B, RandGenOptions<LogProbArcSelector<Arc> >(LogProbArcSelector<Arc>()));

fstrandgen --select=log_prob a.fst b.fst

**Complexity:**

RandGen

- Time: $O(N * L * c_T)$
- Space: $O(N * L + c_S)$

where $N$ = number of paths to be generated, $L$ = expected length of a successful path according to the considered arc selector, $c_T$ = time required for randomly selecting an arc, and $c_S$ = space required for randomly selecting an arc.
Relabel

Description

This operation relabels the input and/or output labels of an FST. The input and/or output relabeling are specified by providing the corresponding sets of pairs \((old\_label, new\_label)\) of label IDs.

When the FST has input and/or output symbol tables, the relabeling can be specified by giving new input and/or output symbol tables. The operation then modifies the label IDs of each arc in such a way that the symbols are preserved.

Usage

```
template <class Arc> 
void Relabel(
    MutableFst<Arc> *fst, 
    const vector<pair<typename Arc::Label, typename A::Label> >& ipairs, 
    const vector<pair<typename Arc::Label, typename A::Label> >& opairs);
```

```
template<class Arc>
void Relabel(
    MutableFst<Arc> *fst, 
    const SymbolTable* new_isymbols, 
    const SymbolTable* new_osymbols);
```

```
template <class Arc> RelabelFst<Arc>:: 
RelabelFst(const Fst<A>& fst, 
    const vector<pair<Label, Label> >& ipairs, 
    const vector<pair<Label, Label> >& opairs); 
```

```
template <class Arc> RelabelFst<Arc>:: 
RelabelFst(const Fst<A>& fst, 
    const SymbolTable* new_isymbols, 
    const SymbolTable* new_osymbols) 

fstrelabel [--relabel_ipairs=$ipairs] [--relabel_opairs=$opairs] in.fst out.fst
```

Examples

A:

<table>
<thead>
<tr>
<th>S1</th>
<th>eps</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>g</td>
<td>6</td>
</tr>
</tbody>
</table>

Relabel
Relabel (&A):

\[
\begin{array}{c|cccc}
S2 & \epsilon & a & b & c \\
0 & 0 & 6 & 5 & 4 \\
1 & & & & \\
2 & & & & \\
3 & & & & \\
4 & & & & \\
5 & & & & \\
6 & & & & \\
\end{array}
\]

\[
\begin{array}{c|cccc}
B & \epsilon & a & b & c \\
0 & 0 & 6 & 5 & 4 \\
1 & & & & \\
2 & & & & \\
3 & & & & \\
4 & & & & \\
5 & & & & \\
6 & & & & \\
\end{array}
\]

Relabel (&A, S2, S2);
RelabelFst<Arc> B(A, S2, S2);

fstrelabel --relabel_isymbols=s2 --relabel_osymbols=s2 a.fst b.fst

\[
\begin{array}{c|cccc}
P & 0 & 0 & 1 & 6 \\
0 & 1 & 5 & 4 & 3 \\
1 & 2 & 4 & 3 & 5 \\
2 & 3 & 5 & 2 & 1 \\
\end{array}
\]

Relabel (&A, P, P);
RelabelFst<Arc> B(A, P, P);

fstrelabel --relabel_ipairs=p --relabel_opairs=p a.fst b.fst
Complexity

Relabel

- Time: $O(V + E + S_i + S_o)$
- Space: $O(S_i + S_o)$

where $V$ = # of states, $E$ = # of arcs of the input FST, $S_i$ = # of input symbols and $S_o$ = # of output symbols.

RelabelFst

- Time: $O(v + e + S_i + S_o)$
- Space: $O(S_i + S_o)$

where $v$ = # of states visited, $e$ = # of arcs visited of the input FST, $S_i$ = # of input symbols and $S_o$ = # of output symbols. Constant time and space to visit an input state or arc is assumed and exclusive of caching.
Replace

Description

This operation performs the dynamic replacement of arcs in one FST with another FST, allowing the definition of FSTs analogous to RTNs. It takes as input 1) a set of pairs formed by a non-terminal label and its corresponding FST, and 2) a label identifying the root FST in that set. The resulting FST is obtained by taking the root FST and recursively replacing each arc having a nonterminal as output label by its corresponding FST.

More precisely, an arc from state \( s \) to state \( d \) with output label the nonterminal \( n \) is replaced by redirecting this arc to the initial state of a copy \( F \) of the FST for nonterminal \( n \), replacing its output label by epsilon (and its input label by epsilon when the option \( \text{epsilon}_\text{on}_\text{replace} \) is set to true), and adding epsilon arcs from each final state of \( F \) to \( d \).

Usage

```cpp
template <class Arc>
void Replace(const vector<pair<typename Arc::Label, const Fst<Arc>*> >& label_fst_pairs, 
             MutableFst<Arc> *ofst, 
             typename Arc::Label root, 
             bool epsilon_on_replace);
```

```cpp
template <class Arc> ReplaceFst<Arc>::
ReplaceFst(const vector<pair<typename Arc::Label, const Fst<Arc>* > >& label_fst_pairs, 
           typename Arc::Label root);
```

```cpp
template <class Arc> ReplaceFst<Arc>::
ReplaceFst(const vector<pair<typename Arc::Label, const Fst<Arc>* > >& label_fst_pairs, 
           const ReplaceFstOptions<Arc> &opts);
```

```bash
fstreplace [--epsilon_on_replace] root.fst rootlabel [subfst1.fst label1 ....] [out.fst]
```

Examples

**Symbol table:**

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eps</td>
<td>0</td>
</tr>
<tr>
<td>$Root</td>
<td>1</td>
</tr>
<tr>
<td>$Name</td>
<td>2</td>
</tr>
<tr>
<td>$FirstName</td>
<td>3</td>
</tr>
<tr>
<td>$LastName</td>
<td>4</td>
</tr>
<tr>
<td>dial</td>
<td>5</td>
</tr>
<tr>
<td>please</td>
<td>6</td>
</tr>
<tr>
<td>johan</td>
<td>7</td>
</tr>
<tr>
<td>schalkwyk</td>
<td>8</td>
</tr>
<tr>
<td>google</td>
<td>9</td>
</tr>
<tr>
<td>michael</td>
<td>10</td>
</tr>
<tr>
<td>riley</td>
<td>11</td>
</tr>
</tbody>
</table>
**A1:**
root FST:

**A2:**
FST for nonterminal $Name$

**A3:**
FST for nonterminal $FirstName$

**A4:**
Fst for nonterminal $LastName$

**B:**

```
vector<pair<Label, const Fst<Arc>*>> > > label_fst_pairs;
label_fst_pairs.push_back(make_pair(1, A1.Copy()));
label_fst_pairs.push_back(make_pair(2, A2.Copy()));
label_fst_pairs.push_back(make_pair(3, A3.Copy()));
label_fst_pairs.push_back(make_pair(4, A4.Copy()));
Replace(label_fst_pairs, &B, 1, true);

ReplaceFst<Arc> B(label_fst_pairs, ReplaceFstOptions<Arc>(1, true));

fstreplace --epsilon_on_replace a1.fst 1 a2.fst 2 a3.fst 3 a4.fst 4 b.fst
```

Examples 53
Complexity

Replace

- Time: $O(V + E + N)$
- Space: $O(V + N)$

where $V = \# \text{ of states}$, $E = \# \text{ of transitions in the resulting FST}$, and $N = \# \text{ of nonterminals}$.

ReplaceFst

- Time: $O(v + e + N)$
- Space: $O(v + N)$

where $v = \# \text{ of visited states}$, $e = \# \text{ of visited transitions in the resulting FST}$, and $N = \# \text{ of nonterminals}$.
Constant time and space to visit an input state or arc is assumed and exclusive of caching.

Caveats

A cyclic dependency among a subset of nonterminals will lead to an infinite machine. The presence of such cyclic dependencies can be tested in time linear in the sum of the sizes of the input FSTs using the CyclicDependencies method of the ReplaceFst class.
Reverse

Description

This operation reverses an FST. If \( A \) transduces string \( x \) to \( y \) with weight \( a \), then the reverse of \( A \) transduces the reverse of \( x \) to the reverse of \( y \) with weight \( a.\text{Reverse}() \).

Typically, \( a = a.\text{Reverse}() \) and Arc = RevArc (e.g. for TropicalWeight or LogWeight). In general, e.g., when the weights only form a left or right semiring, the output arc type must match the input arc type except having the reversed Weight type.

Usage

```cpp
template<class Arc, class RevArc>
void Reverse(const Fst<Arc> &ifst, MutableFst<RevArc> *ofst);
```

Examples

**A:**

![Diagram of A]

**Reverse of A:**

![Diagram of Reverse of A]

Reverse(&A);
fstreverse a.fst out.fst

Complexity

Reverse:

- Time: \( O(V + E) \)
- Space: \( O(V + E) \)

where \( V \) = # of states and \( E \) = # of arcs.
**Reweight**

**Description**

This operation reweights an FST according to the potentials and in the direction specified by the user. An arc of weight $w$, with an origin state of potential $p$ and destination state of potential $q$, is reweighted by $p^{-1} \otimes (w \otimes q)$ when reweighting towards the initial state and by $(p \otimes w) \otimes q^{-1}$ when reweighting towards the final states.

The weights need to be left distributive when reweighting towards the initial state and right distributive when reweighting towards the final states (valid for TropicalWeight and LogWeight).

**Usage**

```cpp
enum ReweightType { REWEIGHT_TO_INITIAL, REWEIGHT_TO_FINAL };  // 1

template <class Arc>
void Reweight(MutableFst<Arc> *fst, vector<typename Arc::Weight> potential, ReweightType type)  // 2

fstreweight [--opts] a.fst potentials.txt out.fst
  --to_final: type = bool, default = false
  Push/reweight to final (vs. to initial) states
```

**Complexity**

Reweight:

- **Time:** $O(V + E)$
- **Space:** $O(V + E)$

where $V =$ # of states and $E =$ # of arcs.
RmEpsilon

Description

This operation removes epsilon-transitions (when both the input and output label are an epsilon) from a transducer. The result will be an equivalent FST that has no such epsilon transitions.

Usage

```cpp
template <class Arc>
void RmEpsilon(MutableFst<Arc> *fst);
template <class Arc> RmEpsilonFst<Arc>::RmEpsilonFst(const Fst<Arc>& fst);
```

```
fstrmepsilon [--opts] a.fst out.fst
--connect: Trim output (def: true)
```

Examples

A:

![Diagram of A]

```
RmEpsilon(&A);
RmEpsilonFst<Arc>(A);
fstrmepsilon a.fst out.fst
```

Complexity

RmEpsilon:

- **Time:**
  - Unweighted: $O(V^2 + VE)$
  - Acyclic: $O(V^2 + VE)$
  - Tropical semiring: $O(V^2 \log V + VE)$
  - General: exponential
- **Space:** $O(VE)$

where $V = \#$ of states and $E = \#$ of arcs.
RmEpsilonFst:

- Time:
  - Unweighted: $O(v^2 + v \cdot e)$
  - General: exponential
- Space: $O(v \cdot e)$

where $v =$ # of states visited, $e =$ # of arcs visited. Constant time to visit an input state or arc is assumed and exclusive of caching.

References

ShortestDistance

Description

This operation computes the shortest distance from the initial state to every state (when reverse is false) or from every state to the final states (when reverse is true). The shortest distance from $p$ to $q$ is the $\oplus$-sum of the weights of all the paths between $p$ and $q$.

The weights must be right (left) distributive if reverse is false (true) and $k$-closed (i.e., $1 \oplus x \oplus x^2 \oplus \ldots \oplus x^{k+1} = 1 \oplus x \oplus x^2 \oplus \ldots \oplus x^k$) (valid for TropicalWeight).

Usage

```cpp
template<class Arc>
void ShortestDistance(const Fst<Arc> &fst, vector<typename Arc::Weight> *distance, bool reverse = false);
```

Usage example:

```
fstshortestdistance [--opts] a.fst [distance.txt]
    --reverse: type = bool, default = false
    Perform in the reverse direction
```

Examples

$A$, over the tropical semiring:

```
<table>
<thead>
<tr>
<th>State</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Shortest distance from the initial state:

```
ShortestDistance(A, &distance);
fstshortestdistance a.fst
```

Shortest distance to the final states:

```
<table>
<thead>
<tr>
<th>State</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```
ShortestDistance(A, &distance, true);
fstshortestdistance --reverse A.fst

**Complexity**

ShortestDistance:

- Time:
  - Acyclic: $O(V + E)$
  - Tropical semiring: $O(V \log V + E)$
  - General: exponential
- Space: $O(V)$

where $V$ = # of states and $E$ = # of arcs.

**References**

**ShortestPath**

**Description**

This operation produces an FST containing the \( n \) -shortest paths in the input FST. The \( n \) -shortest paths are the \( n \) -lowest weight paths w.r.t. the natural semiring order. The single path that can be read from the ith of at most \( n \) transitions leaving the initial state of the resulting FST is the ith shortest path.

The weights need to be right distributive and have the path property. They also need to be left distributive as well for \( n \) -shortest with \( n > 1 \) (valid for TropicalWeight).

**Usage**

```cpp
template<class Arc>
void ShortestPath(const Fst<Arc> &ifst, MutableFst<Arc> *ofst, size_t n = 1);

fstshortestpath [--opts] a.fst out.fst
    --nshortest: type = int64, default = 1
    Return N-shortest paths
```

**Examples**

**A:**

![Diagram of A]

**Shortest path in A:**

![Diagram of shortest path in A]

**2-shortest paths in A:**

![Diagram of 2-shortest paths in A]

**Complexity**

ShortestPath:

- **1-shortest path:**
  - Time: \( O(V \log V + E) \)
  - Space: \( O(V) \)
• **n-shortest paths:**
  ♦ Time: $O(V \log V + n V + n E)$
  ♦ Space: $O(n V)$

where $V = \#$ of states and $E = \#$ of arcs.

**References**

• Mehryar Mohri and Michael Riley. An Efficient Algorithm for the n-best-strings problem, In *Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP ’02).*
Synchronize

Description

This operation synchronizes a transducer. The result will be an equivalent FST that has the property that during the traversal of a path, the delay is either zero or strictly increasing, where the delay is the difference between the number of non-epsilon output labels and input labels along the path.

For the algorithm to terminate, the input transducer must have bounded delay, i.e., the delay of every cycle must be zero.

Usage

```cpp
template <class Arc>
void Synchronize(const Fst<Arc> &fst, MutableFst<Arc> *ofst);
template <class Arc> SynchronizeFst<Arc>::
SynchronizeFst(const Fst<Arc>& fst);
fstsynchronize a.fst out.fst
```

Examples

A:

![Diagram of A]

Synchronize of A:

![Diagram of Synchronize of A]

Synchronize(A, &B);
SynchronizeFst<Arc>(A);
fstsynchronize a.fst out.fst

Complexity

Synchronize:

- A has bounded delay: Time and Space complexity is exponential
- A does not have bounded delay: does not terminate

SynchronizeFst:
OpenFst Library

- A has bounded delay: Time and Space complexity is *exponential*
- A does not have bounded delay: *does not terminate*

References

**Topsort**

**Description**
This operation topologically sorts its input if acyclic, modifying it. Otherwise, the input is unchanged. When sorted, all transitions are from lower to higher state IDs.

**Usage**

```cpp
template<class Arc>
void Topsort(MutableFst<Arc> *fst);

fsttopsort a.fst out.fst
```

**Examples**

**A:**

![Diagram of A]

**Topsort of A:**

![Diagram of Topsort of A]

```cpp
Topsort(*A);
fsttopsort a.fst out.fst
```

**Complexity**

**Topsort:**

- Time: $O(V + E)$
- Space: $O(V + E)$

where $V = \# \text{ of states}$ and $E = \# \text{ of arcs}$.
**Union**

**Description**

This operation computes the union (sum) of two FSTs. If \( A \) transduces string \( x \) to \( y \) with weight \( a \) and \( B \) transduces string \( w \) to \( v \) with weight \( b \), then their union transduces \( x \) to \( y \) with weight \( a \) and \( w \) to \( v \) with weight \( b \).

**Usage**

```cpp
template <class Arc>
void Union(MutableFst<Arc> *fst1, const Fst<Arc> &fst2);

template <class Arc> UnionFst<Arc>::UnionFst(const Fst<Arc> &fst1, const Fst<Arc> &fst2);
```

**Examples**

**A:**

```
[Diagram of FST A]
```

**B:**

```
[Diagram of FST B]
```

**A ∪ B:**

```
[Diagram of FST Union]
```

```cpp
Union(&A, B);
UnionFst<Arc>(A, B);
fstunion a.fst b.fst out.fst
```
Complexity

Union:

• Time: $O(V_2 + E_2)$
• Space: $O(V_2 + E_2)$

where $V_i =$ # of states and $E_i =$ # of arcs of the $ith$ FST.

UnionFst:

• Time: $O(v_1 + e_1 + v_2 + e_2)$
• Space: $O(v_1 + v_2)$

where $v_i =$ # of states visited and $e_i =$ # of arcs visited of the $ith$ FST. Constant time and space to visit an input state or arc is assumed and exclusive of caching.
Verify

Description

This operation checks the sanity of a FST's contents. It returns false if the transducer is incomplete or ill-formed (e.g., a non-trivial FST that has no initial state or transitions to non-existent destination states).

Usage

```cpp
template<class Arc>
bool Verify(const Fst<Arc> &fst);
```

Complexity

Verify:

- Time: $O(V + E)$
- Space: $O(1)$

where $V = \# \text{ of states}$ and $E = \# \text{ of arcs}$. 
OpenFst Conventions

The OpenFst Library has various conventions and assumptions about its objects and coding style.

Object Conventions

1. The StateIds of an FST are dense integers, numbered from 0 to NumStates() - 1.
2. A StateIterator returns StateIds in numerical order.
3. A user may not request info about a StateId from an FST unless the FST has already returned a StateId (e.g. from Start(), NumStates(), StateIterator, or ArcIterator->Value().nextstate).
4. The empty machine has start state kNoState.
5. Labels are non-negative integers except kNoLabel (or library internals). The label 0 is reserved for epsilon.
6. Weights satisfy the properties described here.
7. Arc weights satisfy the property that the sum of the weights of one or more paths from some state S to T is never Zero(). In particular, arc weights are never Zero().
8. State iterators are invalidated if the number of states is modified.
9. Arc iterators are invalidated if the number of states or arcs is modified.
10. A reference/pointer to an arc is invalidated at the next Fst, state or arc iterator operation.
11. State and arc iterators should be destroyed prior to destroying their component FSTs.
12. All Fst classes F implement a copy constructor F(const &F).
13. The copy constructor and Copy() method of an FST have constant time and space complexity (shallow copy) unless otherwise noted.

Coding Conventions

FST Application Code:

1. Your code should rarely use int or float when referring to FST components. Use:
   ♦ StateId for state Ids and number of states.
   ♦ Label for labels.
   ♦ size_t for other array lengths.
   ♦ Weight for weights.

New FST classes:

1. All Fst classes will implement working versions of all pure virtual member functions; writing dummy or error-raising versions is not permitted.
2. If class C is templated on an Arc, then C::Arc will give that type.
FST Weight Requirements

A *semiring* is specified by two binary operations $\oplus$ and $\otimes$ and two designated elements 0 and 1 with the following properties:

- $\oplus$: associative, commutative, and has 0 as its identity.
- $\otimes$: associative and has identity 1, distributes w.r.t. $\oplus$, and has 0 as an annihilator: $0 \otimes a = a \otimes 0 = 0$.

A left semiring distributes on the left; a right semiring is similarly defined.

A *Weight* class must have binary functions $\text{Plus}$ and $\text{Times}$ and static member functions $\text{Zero}()$ and $\text{One}()$ and these must form (at least) a left or right semiring.

In addition, the following must be defined for a *Weight*:

- **Member**: predicate on set membership.
- **>>**: reads textual representation of a weight.
- **<<**: prints textual representation of a weight.
- **Read(istream &)**: reads binary representation of a weight.
- **Write(ostream &)**: writes binary representation of a weight.
- **Hash**: maps weight to size_t.
- **ApproxEqual**: approximate equality (for inexact weights)
- **Quantize**: quantizes wrt delta (for inexact weights)
- **Divide**: $\forall a, b, c \text{ s.t. } \text{Times}(a, b) = c$
  
  $b' = \text{Divide}(c, a, \text{DIVIDE\_LEFT})$ if a left semiring, $b'.\text{Member}()$ and $\text{Times}(a, b') = c$
  
  $a' = \text{Divide}(c, b, \text{DIVIDE\_RIGHT})$ if a right semiring and $a'.\text{Member}()$ and $\text{Times}(a', b) = c$
  
  $b' = \text{Divide}(c, a) = \text{Divide}(c, a, \text{DIVIDE\_ANY}) = \text{Divide}(c, a, \text{DIVIDE\_LEFT}) = \text{Divide}(c, a, \text{DIVIDE\_RIGHT})$ if a commutative semiring,
  
  $b'.\text{Member}()$ and $\text{Times}(a, b') = \text{Times}(b', a) = c$

- **ReverseWeight**: the type of the corresponding reverse weight. Typically the same type as *Weight* for a (both left and right) semiring. For the left string semiring, it is the right string semiring.

- **Reverse**: a mapping from *Weight* to *ReverseWeight* s.t.
  
  $\text{Reverse}(\text{Reverse}(a)) = a$
  
  $\text{Reverse}(\text{Plus}(a, b)) = \text{Plus}(\text{Reverse}(a), \text{Reverse}(b))$
  
  $\text{Reverse}(\text{Times}(a, b)) = \text{Times}(\text{Reverse}(b), \text{Reverse}(a))$

  Typically the identity mapping in a (both left and right) semiring. In the left string semiring, it maps to the reverse string in the right string semiring.

- **Properties**: specifies properties that hold:
  
  - **LeftSemiring**: indicates weights form a left semiring
  - **RightSemiring**: indicates weights form a right semiring
  - **Commutative**: $\forall a, b: \text{Times}(a, b) = \text{Times}(b, a)$
  - **Idempotent**: $\forall a: a \oplus a = a$.
  - **Path**: $\forall a, b: a \oplus b = a$ or $a \oplus b = b$.  

---

FST Weight Requirements
OpenFst Advanced Usage

Below are a variety of topics covered in greater depth or of more specialized interest than found in the Quick Tour. Reading the Quick Tour first is recommended.

Arc Iterators

An arc iterator is used to access the transitions leaving an FST state. It has the form:

```cpp
template <class F>
class ArcIterator {
    typedef typename F::Arc Arc;
    typedef typename Arc::StateId StateId;

    public:
    ArcIterator(const &F fst, StateId s);
    // End of iterator?
    bool Done() const;
    // Current arc (when !Done)
    const Arc& Value() const;
    // Advance to next arc (when !Done)
    void Next();
    // Return current position
    size_t Position();
    // Return to initial position
    void Reset();
    // Arc access by position
    void Seek(size_t pos);
};
```

It is templated on the Fst class to allow efficient specializations but defaults to a generic version on the abstract base Fst class.

See here for conventions that arc iterator use must respect.

All current OpenFst library Seek() methods are constant time.

An example use of an arc iterator is shown here.

A MutableArcIterator is similar to an ArcIterator except its constructor takes a pointer to a MutableFst and it additionally has a SetValue() method.

Arc Filters

Arc filters are accepted by various operations to control which arcs are transitioned. An arc filter has the form:

```cpp
template <class Arc>
class SomeArcFilter {
    public:
    // Return true iff arc is to be transitioned.
    bool operator()(const Arc &arc) const;
};
```
Pre-defined arc filters include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnyArcFilter</td>
<td>Accept all arcs</td>
</tr>
<tr>
<td>EpsilonArcFilter</td>
<td>Accept only arcs with input and output epsilons</td>
</tr>
<tr>
<td>InputEpsilonArcFilter</td>
<td>Accept only arcs with input epsilons</td>
</tr>
<tr>
<td>OutputEpsilonArcFilter</td>
<td>Accept only arcs with output epsilons</td>
</tr>
</tbody>
</table>

**Arcs**

An Arc is a type that represents an FST transition from a given source state. It specifies an input label, an output label, a weight, and a destination state ID and it has a type name. In particular, it has the following form:

```cpp
default SomeArc {
    typedef W Weight;
    typedef L Label;
    typedef S StateId;

    static const string &Type();

    Label ilabel;
    Label olabel;
    Weight weight;
    StateId nextstate;
};
```

where W is a valid weight type, and L and S are signed integral types.

The following arc types are defined in the OpenFst library:

<table>
<thead>
<tr>
<th>Name</th>
<th>Label Type</th>
<th>State ID Type</th>
<th>Weight Type</th>
<th>Registered</th>
</tr>
</thead>
<tbody>
<tr>
<td>GallicArc&lt;A, S&gt;</td>
<td>A::Label</td>
<td>A::StateId</td>
<td>GallicWeight&lt;A::Label, A::Weight, S&gt;</td>
<td></td>
</tr>
<tr>
<td>LexicographicArc&lt;W1, W2&gt;</td>
<td>int</td>
<td>int</td>
<td>LexicographicWeight&lt;W1, W2&gt;</td>
<td>✔</td>
</tr>
<tr>
<td>LogArc</td>
<td>int</td>
<td>int</td>
<td>LogWeight</td>
<td>✔</td>
</tr>
<tr>
<td>MinMaxArc</td>
<td>int</td>
<td>int</td>
<td>MinMaxWeight</td>
<td></td>
</tr>
<tr>
<td>ProductArc&lt;W1, W2&gt;</td>
<td>int</td>
<td>int</td>
<td>ProductWeight&lt;W1, W2&gt;</td>
<td></td>
</tr>
<tr>
<td>StdArc</td>
<td>int</td>
<td>int</td>
<td>TropicalWeight</td>
<td>✔</td>
</tr>
<tr>
<td>StringArc&lt;S&gt;</td>
<td>int</td>
<td>int</td>
<td>StringWeight&lt;int, S&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Additional arc information:

- Corresponding weight types
- Elementary arc information
- Fst I/O: How to register arc types for I/O.
- User-defined arcs
Base Fsts

Every Fst must specify an initial state, the final weights, arc and epsilon counts per states, an Fst type name, the Fst's properties, how to copy, read and write the Fst, and the input and output symbol tables (if any). In particular, the base Fst class has the interface:

template <class A>
class Fst {
    public:
        typedef A Arc;
        typedef typename A::Weight Weight;
        typedef typename A::StateId StateId;

        // Initial state
        virtual StateId Start() const = 0;
        // States's final weight
        virtual Final(StateId) const = 0;
        // State's arc count
        virtual NumArcs(StateId) const = 0;
        // States's input epsilon count
        virtual NumInputEpsilons(StateId) const = 0;
        // State's output epsilon count
        virtual NumOutputEpsilons(StateId) const = 0;
        // If test=false, return stored properties bits for mask (some poss. unknown)
        // If test=true, return property bits for mask (computing o.w. unknown)
        virtual Properties(uint64 mask, bool test) const = 0;
        // Fst type name
        virtual const string& Type() const = 0;
        // Get a copy of this Fst
        virtual Fst<A> *Copy() const = 0;
        // Read an Fst from an input stream; returns NULL on error
        static Fst<A> *Read(istream &strm, const FstReadOptions &opts);
        // Read an Fst from a file; return NULL on error
        // Empty filename reads from standard input
        static Fst<A> *Read(const string &filename);
        // Write an Fst to an output stream; return false on error
        virtual bool Write(ostream &strm, const FstWriteOptions &opts);
        // Write an Fst to a file; return false on error
        // Empty filename writes to standard output
        virtual bool Write(const string &filename);
        // Return input label symbol table; return NULL if not specified
        virtual const SymbolTable* InputSymbols() const = 0;
        // Return output label symbol table; return NULL if not specified
        virtual const SymbolTable* OutputSymbols() const = 0;
    }
}

Fst is an abstract class (note the pure virtual methods). All OpenFst FSTs must meet this interface.

The companion state iterator and arc iterator classes provide access to the states and transitions of the FST.

Caching

Most of the delayed Fst classes use internal caching to save expanded states and arcs. This caching is controlled by this struct:

struct CacheOptions {

Base Fsts
// enable GC
bool gc;
// # of bytes allowed before GC
size_t gc_limit;

CacheOptions(bool g, size_t l) : gc(g), gc_limit(l) {};
CacheOptions() : gc(FLAGS_fst_default_cache_gc), gc_limit(FLAGS_fst_default_cache_gc_limit) {};

All OpenFst cached Fsts have constructors that accept this (or a class derived from it) as an argument. The member defaults are controlled by global flags. These options can be used for:

- **Maximal caching**: If $gc$ is false, then any expanded state will be cached for the extent of the FST. This case is useful when states are revisited and memory is not a concern.

- **Bounded caching**: If $gc$ is true, then the cache will be garbage-collected when it grows past $gc_limit$. This case is useful when states are revisited and memory is a concern. This is the default case (based on the global flags).

- **Minimal caching**: It is generally not possible to avoid all caching in such an FST since the cache is used to implement the arc iterators efficiently (creating an iterator computes and writes the state's arcs to the cache, iterating reads from the cache). However, if (1) $gc$ is true, (2) $gc_limit$ is 0, and (3) arcs iterators have been created (and then destroyed) only one state at a time, then only information for that state is cached and this case is especially optimized. This case is useful when states are not revisited (e.g. when a cached FST is simply being copied to a mutable FST).

### Command Line Flags

OpenFst has several global options in the library proper that most users can ignore, leaving them with their default values:

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAGS_fst_compat_symbols</td>
<td>bool</td>
<td>true</td>
<td>Require symbol tables to match when appropriate</td>
</tr>
<tr>
<td>FLAGS_fst_default_cache_gc</td>
<td>bool</td>
<td>true</td>
<td>Enable garbage collection of cached Fsts</td>
</tr>
<tr>
<td>FLAGS_fst_default_cache_gc_limit</td>
<td>int64</td>
<td>1048576</td>
<td>Byte size that triggers garbage collection of cached Fsts</td>
</tr>
<tr>
<td>FLAGS_fst_pair_parentheses</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Characters enclosing the first weight of a printed pair weight (and derived classes) to ensure proper I/O of nested pair weights; must have size 0 (none) or 2 (open and close parenthesis)</td>
</tr>
<tr>
<td>FLAGS_fst_pair_separator</td>
<td>string</td>
<td>&quot;,,&quot;</td>
<td>Character separator between printed pair weights; must be a single character</td>
</tr>
<tr>
<td>FLAGS_fst_verify_properties</td>
<td>bool</td>
<td>false</td>
<td>Verify Fst properties are correctly set when queried</td>
</tr>
</tbody>
</table>

The first ensures the arguments of binary FST operations (e.g. composition) have compatible symbol tables (e.g. output symbol table matches input symbol table for composition). The second two are used to control the...
caching of expanded state and arc information found in most delayed Fst classes; the default values should normally be satisfactory. The next two are used to control the text formatting of ProductWeight and other weight pairs. The last is used to ensure that the properties of an FST have been correctly set; it is used for debugging only since it incurs considerable computational cost.

In each of the Fst distribution installed binaries, the above options, as well as any of those defined specific to the binary, can be set from the command line using e.g. \texttt{--fst_default_cache_gc=false} or \texttt{--fst_pair_parenthesis=\{\}}. Additionally, the option \texttt{--help} and \texttt{--v=N} (where N = 0,1,2,...) will print out usage information and set the verbosity level of logging, respectively. The flag processing is modeled after the Google gflags package.

In a user-defined binary, the command line options processing will all also work if the user calls:

\begin{verbatim}
SetFlags(usage, argc, argv, true);
\end{verbatim}

In that case, the user can set his own flags as well, following the conventions in \texttt{<fst/flags.h>}. Alternatively, the user can process options in his own way and directly assign to any of the above global options if he wishes to modify their defaults.

\section*{Composition Filters}

A composition filter determines which matches are allowed to proceed in composition. The basic filters handle correct epsilon matching. In particular, they ensure that redundant epsilon paths, which would be incorrect with non-idempotent weights, are not created. More generally, composition filters can be used to block or modify composition paths for efficiency or specialized purposes. Their interface is:

\begin{verbatim}
template <class A>
    class SomeComposeFilter {
        public:
            typedef A Arc;
            typedef ... FilterState;

            // Required constructor.
            SomeComposeFilter(const Fst&lt A> &fst1, const Fst&lt ltA> &fst2);
            // Return start state of filter.
            FilterState Start() const;
            // Specifies current composition state.
            void SetState(StateId s1, StateId s2, const FilterState &f);

            // Apply filter at current composition state to these transitions.
            // If an arc label to be matched is kNolabel, then that side does not consume a symbol.
            // Returns the new filter state or, if disallowed, FilterState::NoState().
            FilterState FilterArc(A *arc1, A *arc2) const;

            // Apply filter at current composition state to these final weights
            // (cf. superfinal transitions). The filter may modify its inputs, e.g. for optimizations.
            void FilterFinal(Weight *final1, Weight *final2) const;
    }
\end{verbatim}

The filter's state is represented by the type \texttt{SomeComposeFilter::FilterState} and is stored in the composition state table tuple. It has the form:

\begin{verbatim}
OpenFst Library

Command Line Flags 75
\end{verbatim}
class SomeFilterState {
    public:
        // Required constructors
        SomeFilterState();
        SomeFilterState(const SomeFilterState &f);
        // An invalid filter state.
        static const SomeFilterState NoState();
        // Maps state to integer for hashing.
        size_t Hash() const;
        // Equality of filter states.
        bool operator==(const SomeFilterState &f) const;
        // Inequality of filter states.
        bool operator!=(const SomeFilterState &f) const;
        // Assignment to filter states.
        SomeFilterState& operator=(const SomeFilterState& f);
    
    The following composition filters are defined in the OpenFst library:

    | Name                        | Description                                             |
    |------------------------------|---------------------------------------------------------|
    | SequenceComposeFilter       | Requires epsilons on FST1 to be read before epsilons on FST2 |
    | AltSequenceComposeFilter    | Requires epsilons on FST2 to be read before epsilons on FST1 |
    | MatchComposeFilter          | Requires epsilons on FST1 to be matched with epsilons on FST2 whenever possible |

    SequenceComposeFilter is the default composition filter. It can be changed by using the version of ComposeFst that accepts ComposeFstOptions.

    Expanded Fsts

    An ExpandedFst is an Fst that has an additional method that specifies the state count as well as methods to copy and read the expanded FST. In particular, an ExpandedFst class has the interface:

    template <class A>
    class ExpandedFst : public Fst<class A> {
    public:
        typedef A Arc;
        typedef typename A::StateId StateId;

        // State count
        StateId NumStates();
        // Get a copy of this ExpandedFst
        virtual ExpandedFst<A> *Copy() const = 0;
        // Read an ExpandedFst from an input stream; returns NULL on error
        static ExpandedFst<A> *Read(istream &strm, const FstReadOptions &opts);
        // Read an ExpandedFst from a file; return NULL on error
        // Empty filename reads from standard input
        static ExpandedFst<A> *Read(const string &filename);
    
    ExpandedFst is an abstract class (note the pure virtual methods). Examples are VectorFst and ConstFst.
FST Input/Output

The following code:

```cpp
VectorFst<A> ifst;
...
ifst.Write("a.fst");
VectorFst<A> *ofst = VectorFst<A>::Read("a.fst");
```

writes and reads a defined Fst type (`VectorFst`) and arc type (`A`) to and from a file in a straight-forward way.

Library Registration

The call:

```cpp
Fst<Arc> *fst = Fst<A>::Read("a.fst");
```

reads the same `VectorFst` from the file as above, but returns a base `Fst`. This form, useful for code that works generically for different Fst types, can not work unless the Fst and arc type are *library-registered*. Some arc types (see here) are already registered for all the Fst types defined in the OpenFst library. Other arc type `A` and Fst type `F` pairs can be registered with the following call:

```cpp
REGISTER_FST(F, A);
```

To avoid code bloat in a given program, registering arc types, in particular, should be used sparingly.

Main Registration

In the above examples, the user provided the arc type as a template parameter. However, the call:

```bash
$ fstinfo a.fst
```

works e.g. for both `StdArc` and `LogArc` arcs. This is accomplished by calling in `main(argc, argv)`:

```cpp
return CALL_FST_MAIN(InfoMain, argc, argv);
```

where:

```cpp
template <class Arc>
int InfoMain(int argc; char **argv, istream &strm, const FstReadOptions &opts) {
    Fst<Arc> *fst = Fst<Arc>::Read(strm, opts);
    ...
    return 0;
}
```

is a templated `main` function that does the arc-specific work. `CALL_FST_MAIN` passes to `InfoMain` the command line arguments and an opened stream to an `Fst` (opened from the first argument or standard input if no arguments). `CALL_FST_MAIN` does the type dispatch by examining the Fst's header and then passing on the (partially-read) input stream, which can be used by `InfoMain` to read in the actual `Fst`. This dispatch works only with arc and main function pairs that have been *main-registered*. Each OpenFst distribution binary registers its templated main function with the arc types marked registered here. An arc type `A` and main
function \( M \) pair can be registered with the following call:

\[
\text{REGISTER\_FST\_MAIN}(M, A);
\]

To avoid code bloat in a given program, registering arc types should be used sparingly.

To use main registration in your own program, you need to include additionally `<fst/main.h>` and link additionally to `libfstmain.so`.

**FST Dynamic Shared Objects**

The examples above show how users can modify programs to be able to read new arc and Fst types. However, it would not be ideal to have to do so for all the distribution binaries or other existing programs. Instead, this can be done more easily with dynamic shared objects (DSOs).

To add a new Fst type, `MyFst` with `MyFst::Type()` = "my_fst", use the code:

```c
extern "C" {
void my_fst_init() {
    // Register some arc types with this Fst type
    REGISTER_FST(MyFst, StdArc);
    REGISTER_FST(MyFst, LogArc);
}
}
```

compiled into a dynamic shared object `my_fst.so`. If `my_fst.so` can be found in the `LD_LIBRARY_PATH` (or equivalent), you should be able to read the new Fst type with existing programs.

To add a new arc type, `MyArc` with `MyArc::Type()` = "my_arc", use the code:

```c
extern "C" {
void my_arc_init() {
    // Register some Fst types with this arc type
    REGISTER_FST(VectorFst, MyArc);
    REGISTER_FST(ConstFst, MyArc);
    // Register the OpenFst binaries with this arc type
    REGISTER_FST_MAINS(MyArc);
    // Register some other main with this arc type
    REGISTER_FST_MAIN(SomeMain, MyArc);
}
}
```

compiled into a dynamic shared object `my_arc.so`. If can be found in `LD_LIBRARY_PATH` (or equivalent), you should be able to read the new arc type with existing programs.

**Mappers**

Mappers are function objects used by the Map operation to transform arcs and/or final states. A mapper has the form:

```c
// This determines how final weights are mapped.
enum MapFinalAction {
```
// A final weight is mapped into a final weight. An error
// is raised if this is not possible.
MAP_NO_SUPERFINAL,

// A final weight is mapped to an arc to the superfinal state
// when the result cannot be represented as a final weight.
// The superfinal state will be added only if it is needed.
MAP_ALLOW_SUPERFINAL,

// A final weight is mapped to an arc to the superfinal state
// unless the result can be represented as a final weight of weight
// Zero(). The superfinal state is always added (if the input is
// not the empty Fst).
MAP_REQUIRE_SUPERFINAL
};

// This determines how symbol tables are mapped.
enum MapSymbolsAction {
    // Symbols should be cleared in the result by the map.
    MAP_CLEAR_SYMBOLS,
    // Symbols should be copied from the input FST by the map.
    MAP_COPY_SYMBOLS,
    // Symbols should not be modified in the result by the map itself.
    // (They may set by the mapper).
    MAP_NOOP_SYMBOLS
};

class SomeMapper {
public:
    // Maps an arc type A to arc type B.
    B operator()(const A &arc);
    // Specifies final action the mapper requires (see above).
    // The mapper will be passed final weights as arcs of the
    // form A(0, 0, weight, kNoStateId).
    MapFinalAction FinalAction() const;
    // Specifies input symbol table action the mapper requires (see above).
    MapSymbolsAction InputSymbolsAction() const;
    // Specifies output symbol table action the mapper requires (see above).
    MapSymbolsAction OutputSymbolsAction() const;
    // This specifies the known properties of an Fst mapped by this
    // mapper. It takes as argument the input Fst's known properties
    uint64 Properties(uint64 props) const;
};

The following mappers are defined in the OpenFst library:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FromGallicMapper</td>
<td>Extracts output label from gallic weight</td>
</tr>
<tr>
<td>IdentityMapper</td>
<td>Maps to self</td>
</tr>
<tr>
<td>InvertWeightMapper</td>
<td>Reciprocate all non-0 weights</td>
</tr>
<tr>
<td>PlusMapper</td>
<td>Adds (⊕) a constant to all weights</td>
</tr>
<tr>
<td>QuantizeMapper</td>
<td>Quantize all weights</td>
</tr>
<tr>
<td>ReverseWeightMapper</td>
<td>Reverse all weights</td>
</tr>
<tr>
<td>RmWeightMapper</td>
<td>Map all non-0 weights to 1</td>
</tr>
<tr>
<td>SuperFinalMapper</td>
<td>Redirects final states to new superfinal state</td>
</tr>
<tr>
<td>TimesMapper</td>
<td>(Right) multiplies (⊗) a constant to all weights</td>
</tr>
</tbody>
</table>
Matchers

Matchers can find and iterate through requested labels at FST states; their principal use is in composition matching. In the simplest form, these are just a search or hash keyed on labels. More generally, they may implement matching special symbols that represent sets of labels such as \( \rho \) (rest), \( \sigma \) (all) or \( \phi \) (fail), which can be used for more compact automata representations and faster matching.

The Matcher interface is:

```cpp
enum MatchType {
    MATCH_INPUT, // Match input label.
    MATCH_OUTPUT, // Match output label.
    MATCH_NONE, // Match nothing.
    MATCH_UNKNOWN, // Match type unknown.
};

template <class F>
class SomeMatcher {
public:
    typedef F FST;
    typedef F::Arc Arc;
    typedef typename Arc::StateId StateId;
    typedef typename Arc::Label Label;
    typedef typename Arc::Weight Weight;

    // Required constructors.
    SomeMatcher(const F &fst, MatchType type);
    SomeMatcher(const SomeMatcher &matcher);

    // Returns the match type that can be provided (depending on
    // compatibility of the input FST). It is either
    // the requested match type, MATCH_NONE, or MATCH_UNKNOWN.
    // If 'test' is false, a constant time test is performed, but
    // MATCH_UNKNOWN may be returned. If 'test' is true,
    // a definite answer is returned, but may involve more costly
    // computation (e.g., visiting the Fst).
    MatchType Type(bool test) const;

    // Specifies the current state.
    void SetState(StateId s);

    // This finds matches to a label at the current state.
    bool Find(Label label);

    // These iterate through any matches found:
    // No more matches.
    bool Done() const;
    // Current arc (when !Done)
};
```
The following matchers are defined in the OpenFst library:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortedMatcher</td>
<td>Binary search on sorted input</td>
</tr>
<tr>
<td>RhoMatcher&lt;M&gt;</td>
<td>$\rho$ symbol handling; templated on underlying matcher</td>
</tr>
<tr>
<td>SigmaMatcher&lt;M&gt;</td>
<td>$\sigma$ symbol handling; templated on underlying matcher</td>
</tr>
<tr>
<td>PhiMatcher&lt;M&gt;</td>
<td>$\phi$ symbol handling; templated on underlying matcher</td>
</tr>
</tbody>
</table>

SortedMatcher requires the underlying Fst be sorted on the appropriate side. \texttt{Find(0)} matches any epsilons on the underlying Fst explicitly (as if they were any other symbol) but also returns an implicit self-loop (namely \texttt{Arc(kNoLabel, 0, Weight::One(), current_state)} if the \texttt{match_type} is \texttt{MATCH_INPUT} and \texttt{Arc(0, kNoLabel, Weight::One(), current_state)} if the \texttt{match_type} is \texttt{MATCH_OUTPUT}). This loop implements a 'non-consuming' match in composition (with \texttt{kNoLabel} informing composition that this is the case). A composition filter determines which of these epsilon transitions are ultimately accepted.

The special symbols referenced above behave as described in this table:

<table>
<thead>
<tr>
<th>Matches all</th>
<th>Consumes no symbol</th>
<th>Consumes symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matches rest</td>
<td>$\epsilon$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td></td>
<td>$\phi$</td>
<td>$\rho$</td>
</tr>
</tbody>
</table>

The $\epsilon$ symbol is assigned label 0 by convention. The numeric label of the other special symbols is determined by a constructor argument to their respective matchers.

The $\rho$, $\sigma$ and $\phi$ matchers augment the functionality of their underlying template argument matcher. In this way, matchers can be cascaded (with special symbol precedence determined by the order).

A design choice for these matchers is whether to \textit{remove} the special symbol in the result (used for the $\rho$, $\sigma$, and $\phi$ matchers) or \textit{return} it (used for epsilon-handling). The first case is equivalent to (but more efficient than) applying special-symbol removal prior to composition (c.f., epsilon removal). This case requires that only one of the FSTs in composition contain such symbols. The second case requires well-defined semantics and that composition proper identify and handle any non-consuming symbols on each FST. (The result of \texttt{Find(kNoLabel)} identifies on one FST, while the matcher's returning a \texttt{kNoLabel} loop handles the other, both described above.)

The template \texttt{Matcher<F>} selects the pre-designated matcher for Fst type \texttt{F}; it is typically \texttt{SortedMatcher}. Composition uses this matcher by default. It can be changed by using the version of \texttt{ComposeFst} that accepts \texttt{ComposeFstOptions}. Note two matchers (of the same C++ type but different \texttt{MatchType}) are used in composition -- one for each FST. Whether actual match queries are performed on one or both FSTs depends on the matcher constructor arguments, the matcher capabilities (queried by \texttt{Type()}) and composition itself.

An example use of a matcher is here and an example use of a $\rho$ matcher in composition is here.
Mutable Fsts

A **MutableFst** is an ExpandedFst that has additional methods that specify how to set the start state, final weights, properties and the input and output symbols, how to add and delete states and arcs, as well as methods to copy and read the mutable FST. In particular, a **MutableFst** class has the interface:

```cpp
template <class A>
class MutableFst : public ExpandedFst<class A> {
  public:
    typedef A Arc;
    typedef typename A::StateId StateId;
    typedef typename A::Weight Weight;

    // Set the initial state
    virtual void SetStart(StateId) = 0;
    // Set the initial state
    virtual void SetFinal(StateId, Weight) = 0;
    // Set property bits wrt mask
    virtual void SetProperties(uint64 props, uint64 mask) = 0;
    // Add a state, return its ID
    virtual StateId AddState() = 0;
    // Add an arc to state
    virtual void AddArc(StateId, const A &arc) = 0;
    // Delete some states
    virtual void DeleteStates(const vector<StateId> &) = 0;
    // Delete all states
    virtual void DeleteStates() = 0;
    // Delete some arcs at state
    virtual void DeleteArcs(StateId, size_t n) = 0;
    // Delete all arcs at state
    virtual void DeleteArcs(StateId) = 0;
    // Get a copy of this MutableFst
    virtual MutableFst<A> *Copy() const = 0;
    // Read an MutableFst from an input stream; returns NULL on error
    static MutableFst<A> *Read(istream &strm, const FstReadOptions &opts);
    // Read an MutableFst from a file; return NULL on error
    // Empty filename reads from standard input
    static MutableFst<A> *Read(const string &filename);
    // Set input label symbol table; NULL signifies not unspecified
    virtual void SetInputSymbols(const SymbolTable* isyms) = 0;
    // Set output label symbol table; NULL signifies not unspecified
    virtual void SetOutputSymbols(const SymbolTable* osyms) = 0;
};
```

**MutableFst** is an abstract class (note the pure virtual methods). An example is **VectorFst**.

The companion mutable arc iterator class provides access to and modification of the transitions of the FST.

**Natural Orders**

The natural order $\leq$ associated with a semiring is defined as $a \leq b$ iff $a \oplus b = a$. In the OpenFst library, we define the strict version of this order as:

```cpp
template <class W>
NaturalLess() {
  bool operator()(const W &w1, const W &w2) const {
```
An order is left monotonic w.r.t a semiring iff \( a \leq b \Rightarrow \forall c, c \oplus a \leq c \oplus b \) and \( c \otimes a \leq c \otimes b \); right monotonic is defined similarly. An order is negative iff \( 1 \leq 0 \).

The natural order is a left (right) monotonic and negative partial order iff the semiring is idempotent and left (right) distributive. It is a total order iff the semiring has the path property. See Mohri, "Semiring Framework and Algorithms for Shortest-Distance Problems", Journal of Automata, Languages and Combinatorics 7(3):321-350, 2002.

This is the default total order (under the requirements above) that we use for the shortest path and pruning algorithms. This order is the natural one to use given that it generally needs to be total, monotonic and negative: total so that all weights can be compared, monotonic so there is a practical algorithm, and negative so that the "free" weight 1 is preferred to the "disallowed" weight 0.

**Operation Options**

Many FST operations have versions that accept options, especially option structures, that have not been documented in this Wiki for brevity other than to mention some of the parameters that can be changed. For example, most of the delayed Fsts have constructors that accept options that control caching behavior.

Here is an example that selects minimal caching and the rho matcher (for fst2 \( \rho \)'s) in composition::

```c
typedef RhoMatcher< SortedMatcher<StdFst> > RM;

ComposeFstOptions<StdArc, RM> opts;
opts.gc_limit = 0;
opts.matcher1 = new RM(fst1, MATCH_NONE, kNoLabel);
opts.matcher2 = new RM(fst2, MATCH_INPUT, SomeRhoLabel);
StdComposeFst cfs(fst1, fst2, opts);
```

Follow the links to the code under each operation's documentation for the specific details.

**Properties**

Each Fst has associated with it a set of stored properties that assert facts about it. These are queried in an FST with the `Properties()` method and set in a MutableFst with the `SetProperties()` method. OpenFst library operations use these properties to optimize their performance. OpenFst library operations and mutable FSTs attempt to preserve as much property information in their results as possible without significant added computation.

Some properties are binary - they are either true or false. For each such property, there is a single stored bit that is set if true and not set if false. The binary Fst properties are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kExpanded</td>
<td>Is an ExpandedFst</td>
</tr>
<tr>
<td>kMutable</td>
<td>Is a MutableFst</td>
</tr>
</tbody>
</table>
Other properties are trinary - they are either true, false or unknown. For each such property, there are two stored bits; one is set if true, the other is set if false and neither is set if unknown.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptor</td>
<td>kAcceptor</td>
<td>Input and output label are equal for each arc</td>
</tr>
<tr>
<td></td>
<td>kNotAcceptor</td>
<td>Input and output label are not equal for some arc</td>
</tr>
<tr>
<td>Accessible</td>
<td>kAccessible</td>
<td>All states reachable from the initial state</td>
</tr>
<tr>
<td></td>
<td>kNotAccessible</td>
<td>Not all states reachable from the initial state</td>
</tr>
<tr>
<td></td>
<td>kCoAccessible</td>
<td>All states can reach a final state</td>
</tr>
<tr>
<td></td>
<td>kNotCoAccessible</td>
<td>Not all states can reach a final state</td>
</tr>
<tr>
<td>Cyclic</td>
<td>kCyclic</td>
<td>Has cycles</td>
</tr>
<tr>
<td></td>
<td>kAcyclic</td>
<td>Has no cycles</td>
</tr>
<tr>
<td></td>
<td>kInitialCyclic</td>
<td>Has cycles containing the initial state</td>
</tr>
<tr>
<td></td>
<td>kInitialAcyclic</td>
<td>Has no cycles containing the initial state</td>
</tr>
<tr>
<td>Deterministic</td>
<td>kIDeterministic</td>
<td>Input labels are unique leaving each state</td>
</tr>
<tr>
<td></td>
<td>kNonIDeterministic</td>
<td>Input labels are not unique leaving some state</td>
</tr>
<tr>
<td></td>
<td>kODeterministic</td>
<td>Output labels are unique leaving each state</td>
</tr>
<tr>
<td></td>
<td>kNonODeterministic</td>
<td>Output labels are not unique leaving some state</td>
</tr>
<tr>
<td>Epsilons</td>
<td>kEpsilons</td>
<td>Has input/output epsilons</td>
</tr>
<tr>
<td></td>
<td>kNoEpsilons</td>
<td>Has no input/output epsilons</td>
</tr>
<tr>
<td></td>
<td>kIEpsilons</td>
<td>Has input epsilons</td>
</tr>
<tr>
<td></td>
<td>kNoIEpsilons</td>
<td>Has no input epsilons</td>
</tr>
<tr>
<td></td>
<td>kOEpsilons</td>
<td>Has output epsilons</td>
</tr>
<tr>
<td></td>
<td>kNoOEpsilons</td>
<td>Has no output epsilons</td>
</tr>
<tr>
<td>Sorted</td>
<td>kILabelSorted</td>
<td>Input labels sorted for each state</td>
</tr>
<tr>
<td></td>
<td>kNotILabelSorted</td>
<td>Input labels not sorted for each state</td>
</tr>
<tr>
<td></td>
<td>kOLabelSorted</td>
<td>Output labels sorted for each state</td>
</tr>
<tr>
<td></td>
<td>kNotOLabelSorted</td>
<td>Output labels not sorted for each state</td>
</tr>
<tr>
<td></td>
<td>kTopSorted</td>
<td>States topologically sorted</td>
</tr>
<tr>
<td></td>
<td>kNotTopSorted</td>
<td>States not topologically sorted</td>
</tr>
<tr>
<td>Weighted</td>
<td>kWeighted</td>
<td>Non-trivial arc or final weights</td>
</tr>
<tr>
<td></td>
<td>kNotWeighted</td>
<td>Only trivial arc and final weights</td>
</tr>
</tbody>
</table>

The call `fst.Properties(mask, false)` returns the stored property bits set in the mask bits; some properties may be unknown. It is a constant-time operation. The call `fst.Properties(mask, true)` returns the stored property bits set in the mask bits after computing and updating any of those set in the mask that are unknown. It is a linear-time \(O(V + E)\) operation if any of the requested bits were unknown.

### State Iterators

A state iterator is used to access the states of an FST. It has the form:

```cpp
template <class F>
class StateIterator {
    typedef typename F::Arc Arc;
    typedef typename Arc::StateId StateId;

    public:
        StateIterator(const &F fst);
        // End of iterator?
}
```
bool Done() const;
// Current state ID (when !Done)
StateId Value() const;
// Advance to next state (when !Done)
void Next();
// Return to initial position
void Reset();
};

It is templated on the Fst class \( F \) to allow efficient specializations but defaults to a generic version on the abstract base Fst class.

See here for conventions that state iterator use must respect.

An example use of a state iterator is shown here.

## State Queues

State queues are used by, among others, the shortest path and shortest distance algorithms and by the Visit operation. A state queue has the form:

template <class StateId>
class SomeQueue {
public:
    // Ctr: may need args (e.g., Fst, comparator) for some queues
    SomeQueue(...);
    // Returns the head of the queue
    StateId Head() const;
    // Inserts a state
    void Enqueue(StateId s);
    // Removes the head of the queue
    void Dequeue();
    // Updates ordering of state s when weight changes, if necessary
    void Update(StateId s);
    // Does the queue contain no elements?
    bool Empty() const;
    // Remove all states from queue
    void Clear();
};

Pre-defined state queues include:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoQueue</td>
<td>Automatically-selected from Fst properties</td>
</tr>
<tr>
<td>FifoQueue</td>
<td>First-In, first-Out</td>
</tr>
<tr>
<td>LifoQueue</td>
<td>Last-In, first-Out</td>
</tr>
<tr>
<td>SccQueue</td>
<td>Component graph top-ordered meta-queue</td>
</tr>
<tr>
<td>ShortestFirstQueue</td>
<td>Priority (least weight)</td>
</tr>
<tr>
<td>StateOrderQueue</td>
<td>State-ID ordered</td>
</tr>
<tr>
<td>TopOrderQueue</td>
<td>Topologically ordered</td>
</tr>
</tbody>
</table>

Some queues accept arc filters to control which transitions are explored.
State Tables

State tables determine the bijective mapping between state tuples (e.g. in composition triples of two FST states and a composition filter state) and their corresponding state IDs. They are classes, templated on state tuples, of the form:

template <class T>
class SomeStateTable {
    typedef typename T StateTuple;

    // Required constructors.
    SomeStateTable();
    // Lookup state ID by tuple. If it doesn't exist, then add it.
    StateId FindState(const StateTuple &);
    // Lookup state tuple by state ID.
    const StateTuple<StateId> &Tuple(StateId const);
};

A state tuple has the form:

template <class S>
struct SomeStateTuple {
    typedef typename S StateId;

    // Required constructor.
    SomeStateTuple();
    // Data
    ...
};

A specific state tuple is a ComposeStateTuple that has data members StateId state_id1, StateId state_id2, and FilterState filter_state.

The following state tables are defined in the OpenFst library:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashStateTable</td>
<td>Hash map implementation</td>
</tr>
<tr>
<td>CompactHashStateTable</td>
<td>Hash set implementation</td>
</tr>
<tr>
<td>VectorStateTable</td>
<td>Vector implementation</td>
</tr>
<tr>
<td>VectorHashStateTable</td>
<td>Vector and hash set implementation</td>
</tr>
<tr>
<td>ErasableStateTable</td>
<td>Deque implementation - permits erasures</td>
</tr>
</tbody>
</table>

Different state tables provide different time and space tradeoffs for applications.

Composition state tables are defined using state tables with ComposeStateTuple. They are the principal data structure used by composition other than the result cache.

The following composition state tables are defined in the OpenFst library:

<table>
<thead>
<tr>
<th>Name</th>
<th>State Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenericComposeStateTable</td>
<td>CompactHashStateTable</td>
<td>General-purpose choice</td>
</tr>
<tr>
<td>ProductComposeStateTable</td>
<td>VectorStateTable</td>
<td>Efficient when the composition state space is densely populated</td>
</tr>
<tr>
<td>StringDetComposeStateTable</td>
<td>VectorStateTable</td>
<td></td>
</tr>
</tbody>
</table>

State Tables 86
OpenFst Library

<table>
<thead>
<tr>
<th>Efficient when FST1 is a string and FST2 is deterministic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DetStringComposeStateTable VectorStateTable</td>
</tr>
<tr>
<td>Efficient when FST1 is deterministic and FST2 is a string</td>
</tr>
<tr>
<td>EraseableComposeStateTable ErasableStateTable</td>
</tr>
<tr>
<td>Allows composition state tuple erasure</td>
</tr>
</tbody>
</table>

GenericComposeStateTable is the default composition state table. It can be changed by using the version of ComposeFst that accepts ComposeFstOptions.

Symbol Tables

Symbol tables store the bijective mapping between textual labels, used in reading and printing an FST textual file, and their integer assignment, used in the FST’s internal representation. Symbol tables are usually read in with fstcompile, can be stored by the FST, and used to print out the FST with fstprint. Here are some examples of manipulating symbol tables directly:

```c++
// Various ways to reading symbol tables
StdFst *fst = StdFst::Read("some.fst");
SymbolTable *isyms = fst->InputSymbolTable();
SymbolTable *osyms = fst->OutputSymbolTable();
SymbolTable *syms = SymbolTable::ReadText("some.syms");

// Adding and accessing symbols and keys
syms->AddSymbol("kumquat", 7);
int64 key = syms->Find("kumquat");
string symbol = syms->Find(7);

// Various ways of writing symbol tables
fst->SetInputSymbols(isyms);
fst->SetOutputSymbols(osyms);
fst->Write("some.fst");
syms->WriteText("some.syms");
```

User-defined Fst Arcs and Weights

A user may define his own weight type so long as it meets the necessary requirements.

A user may define his own arc type so long as has the right form. Some Fst I/O with new arc types requires registration.

User-defined Fst Classes

TBA
Visitors

The simplest way to traverse an FST is in state order using a state iterator.

A very general traversal method is to use:

```
Visit(fst, visitor, queue);
```

where the `visitor` object specifies the actions taken in the traversal while the state queue object specifies the traversal order. A `visitor` has the form:

```
// Visitor Interface - class determines actions taken during a visit.
// If any of the boolean member functions return false, the visit is aborted by first calling FinishState() on all unfinished (grey) states and then calling FinishVisit().

template <class Arc>
class SomeVisitor {
public:
  typedef typename Arc::StateId StateId;

  SomeVisitor(T *return_data);
  // Invoked before visit
  void InitVisit(const Fst &fst);
  // Invoked when state discovered (2nd arg is visitation root)
  bool InitState(StateId s, StateId root);
  // Invoked when arc to white/undiscovered state examined
  bool WhiteArc(StateId s, const Arc &a);
  // Invoked when arc to grey/unfinished state examined
  bool GreyArc(StateId s, const Arc &a);
  // Invoked when arc to black/finished state examined
  bool BlackArc(StateId s, const Arc &a);
  // Invoked when state finished
  void FinishState(StateId s);
  // Invoked after visit
  void FinishVisit();
};
```

While a depth-first search can be implemented using `Visit()` with the `LifoQueue()`, it is often better to use the more specialized `DFSVisit()` in `<fst/dfs-visit.h>` since it is somewhat more space-efficient and the specialized visitor interface described there has additional functionality for a DFS.

Pre-defined FST visitors include:

<table>
<thead>
<tr>
<th>Visitor</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyVisitor</td>
<td>Visit</td>
<td>Copies in a queue-specified order</td>
</tr>
<tr>
<td>SccVisitor</td>
<td>DfsVisit</td>
<td>Finds strongly-connected components, accessibility and coaccessibility</td>
</tr>
<tr>
<td>TopOrderVisitor</td>
<td>DfsVisit</td>
<td>Finds topological order</td>
</tr>
</tbody>
</table>

The visit operations optionally accept arc filters to control which transitions are explored.

Weights

A `Weight` is a type that is used to represent the cost of taking transitions in an FST.
The following basic weight templates are defined in the OpenFst library:

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Name</th>
<th>Set</th>
<th>(\oplus) (Plus)</th>
<th>(\otimes) (Times)</th>
<th>0 (Zero)</th>
<th>1 (One)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexicographic</td>
<td>LexicographicWeight(W_1, W_2)</td>
<td>(W_1 \times W_2)</td>
<td>min</td>
<td>(\otimes_{W_1} X \otimes_{W_2})</td>
<td>((0_{W_1,0_{W_2}})</td>
<td>((1_{W_1,1_{W_2}})</td>
<td>min: lexicographic order w.r.t. (W_1) and (W_2) natural orders</td>
</tr>
<tr>
<td>Log</td>
<td>LogWeightTpl&lt;T&gt;</td>
<td>([-\infty, \infty])</td>
<td>-log(e^x)</td>
<td>+</td>
<td>(\infty)</td>
<td>0</td>
<td>T: floating point</td>
</tr>
<tr>
<td>MinMax</td>
<td>MinMaxWeightTpl&lt;T&gt;</td>
<td>([-\infty, \infty])</td>
<td>min</td>
<td>max</td>
<td>(\infty)</td>
<td>-(\infty)</td>
<td>T: floating point</td>
</tr>
<tr>
<td>Product</td>
<td>ProductWeight(W_1, W_2)</td>
<td>(W_1 \times W_2)</td>
<td>(\oplus_{W_1} X \otimes_{W_2})</td>
<td>(\oplus_{W_1} X \otimes_{W_2})</td>
<td>((0_{W_1,0_{W_2}})</td>
<td>((1_{W_1,1_{W_2}})</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>StringWeight(L, {}) StringWeight(L, {})</td>
<td>(L^* \cup {\infty})</td>
<td>longest com. prefix</td>
<td>(\cdot)</td>
<td>(\infty)</td>
<td>(\varepsilon)</td>
<td>L: signed integral</td>
</tr>
<tr>
<td>Tropical</td>
<td>TropicalWeightTpl&lt;T&gt;</td>
<td>([-\infty, \infty])</td>
<td>min</td>
<td>+</td>
<td>(\infty)</td>
<td>0</td>
<td>T: floating point</td>
</tr>
</tbody>
</table>

The following weight types have been defined in the OpenFst library in terms of the above:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GallicWeight(L, W, S)</td>
<td>ProductWeight(StringWeight(L, S), W)</td>
</tr>
<tr>
<td>LogWeight</td>
<td>LogWeightTpl&lt;float&gt;</td>
</tr>
<tr>
<td>MinMaxWeight</td>
<td>MinMaxWeightTpl&lt;float&gt;</td>
</tr>
<tr>
<td>TropicalWeight</td>
<td>TropicalWeightTpl&lt;float&gt;</td>
</tr>
</tbody>
</table>

Weight pairs, such as ProductWeight and LexicographicWeight, can use command line flags to control their textual formatting. FLAGS_fst_pair_weight_separator is printed between the weights (default: ",,"). FLAGS_fst_pair_parentheses (default: "") brackets the pair; if you create complex nested pairs (i.e., tuples), they may need to be printed with non-empty brackets (e.g. " ()") to ensure correct parsing if read back in. These affect only textual (not binary) I/O.

Additional weight information:

- Corresponding arc types
- Elementary weight information
- User-defined weights
- Weight type requirements
OpenFst Glossary

**acceptor**
An acceptor is a finite automaton where each transition has a label and possibly a weight. In this library, an acceptor is represented as a transducer with the input and output label of each transition being equal.

**accessible**
A state \( q \) is accessible iff there is a path from the initial state to \( q \).

**co-accessible**
A state \( q \) is co-accessible iff there is a path from \( q \) to a final state.

**delayed**
An FST is delayed (or lazy, on-the-fly, or dynamic) if the computation of its states and transitions occur only when requested. The complexity of a delayed FSTs constructor is constant-time, while the complexity of its traversal is a function of only those states and transitions that are visited.

**deterministic**
An acceptor is deterministic iff for each state \( q \) there is at most one transition labeled with a given label. A transducer can be input deterministic (or subsequential) or output deterministic.

**epsilon, \( \varepsilon \)**
An input label that consumes no input or an output label emits no output.

**equivalent**
Two transducers are equivalent if for each input string, they produce the same output strings with the same weight. The output strings, however, may differ in the number of paths, in the location of epsilon transitions, or in the distribution of the weights along the paths.

**functional**
A transducer is functional if each input string is transduced to a unique output string. There may be multiple paths, however, that contain this input and output string pair.

**multiplicity**
The maximum number of times a label is repeated at a state - a measure of non-determinism.

**out-degree**
The number of transitions exiting a state.
**semiring**

A semiring is a set of elements (*weights*) together with two binary operations $\oplus$ and $\otimes$ and two designated elements 0 and 1 with the following properties:

- $\oplus$: associative, commutative, and has 0 as its identity.
- $\otimes$: associative and has identity 1, distributes w.r.t. $\oplus$, and has 0 as an annihilator: $0 \otimes a = a \otimes 0 = 0$.

**successful path**

A path from the initial state to some final state.

**transducer**

A transducer is a finite automaton where each transition has an input label, an output label, and possibly a weight.

**trim**

An automaton is trim (or connected) iff it contains no inaccessible or no-coaccessible states.

**zero-sum-free**

A semiring is zero-sum-free if $\forall a, b: a \oplus b = 0 \Rightarrow a = b = 0$. 