Introduction to the use of WFSTs in Speech and Language processing

Paul Dixon and Sadaoki Furui

Department of Computer Science
Tokyo Institute of Technology

APSIPA Conference, 2009
Outline of Tutorial

- Introduction
  - Motivation
  - Description of some popular toolkits
- Introduction to WFSTs and algorithms
  - Covers WFSTs and the algorithms used frequently in speech recognition
- Sample applications using WFSTs
  - Edit distance
    - Implementing a counting WFST using a custom semiring
- Recognizing speech with WFSTs
  - Detailed walkthrough describing the construction of an integrated speech recognition WFST
  - Part-of-speech tagging using WFST framework
 Acknowledgements

- Tasuku Oonishi also at Titech working on the $T^3$ decoder
- WFST introduction section heavily influenced by the following tutorials and papers [15, 13, 4]
Introduction
Outline of Section

- Motivation
- Brief survey of popular WFST toolkits
Why use WFSTs?

- Economics, time to development, testing and so on
- Understanding, go inside look and understand the structures and operations
- Tools for rendering and investigating the internals of WFST - useful when performing analysis and postmortems
- Reduce dependency on domain specific toolkits
- Write and control all own tools based on WFSTs
Motivation for WFST in Speech

- Unified representation of knowledge source
- Pre/Post processing stage and even support tools can make use of WFST framework
- WFST Speech Decoders can be easier to design and implement
- WFST Speech Decoders can be faster [10]
- WFST Speech Decoders can be more flexible
Getting Started

- Get a WFST toolkit
- Many toolkits available, the following are widely used in the speech community
  - ATT FSM Library
  - OpenFst http://www.openfst.org/
  - FSA toolkit http://www-i6.informatik.rwth-aachen.de/~kanthak/fsa.html
- Start experimenting with tools
More Toolkit Details

- OpenFst - Open source toolkit - heavy c++ template code, library or shell commands
- MIT FST Toolkit - Open Source
- OpenFst and ATT use same text representation conversion to and from MIT trivial. Windows and Linux
ATT FSM toolkit

- Closed source
- Binaries for (Windows, *nix x86 and amd64 and some big endian)
- Shell commands free for non-commercial use
- Companion toolkits for language modeling (grm) and decoding tools (dcd) (both with more limited platform support)
- Perhaps the easiest way to get up a WFST speech system running by using the grm and dcd toolkits
- Hasn’t been updated for a long time and the Windows version is lagging a version
OpenFst

- Same authors as ATT toolkit
- Very Similar feature set to ATT tools
- Open source license (Apache 2)
- Shell commands or C++ interfaces
  - Heavy template code
  - Build times can be slow
- Allows for the creation of user types and semirings
  - For example ad custom arc with acoustic and language model scores
  - Add the real or expectation semirings
  - Plug in a custom fst type
- No *shorthand* command line options
- Well documented
MIT Fst

- Open source license (MIT)
- Windows and *nix platforms
- Has some extra interesting tools in addition to the standard WFST operations
- C++ or shell commands
- Source code is available (see usage restrictions)
- Part of the RWTH recognition system
Introduction to WFSTs and Algorithms
WFST Operations Covered

- Rational Operations
  - Sum (Union)
  - Product (Concatenation)
  - Closure (Kleene Closure)
- Unary Operations
  - Reversal
  - Inversion
  - Projection
- Binary Operations
  - Composition
- Optimization Algorithms
  - Determinization
  - Minimization
  - Epsilon Removal
  - Weight Pushing
- Other Commands and utilities
  - Compiling, Printing, Drawing and Map
  - ArcSort, Shortestpath, Shorterldistance (all by example)
WFST Operations not Covered

- Intersection
- Equivalence
- Label pushing
- Encode/Decode
- Difference
- Topological Sort
- Epsilon Normalization
- Equivalent
- Synchronization
- Prune
- A few others ....
Formally a weighted transducer $T$ of semiring $S$ is defined as the 8-tuple \[ (1) \]

\[
T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)
\]

Where:

- $\Sigma$ is a finite input alphabet
- $\Delta$ is a finite output alphabet
- $Q$ is a finite set of states
- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of final states
- $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\})$ is a finite set of transitions
- $\lambda$ is an initial weight function
- $\rho$ is the final weight function
A semiring is a system with two operators $\oplus$ and $\otimes$, the table below shows some example semiring that are often used in WFST tool-kits [13, 15]

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Set</th>
<th>$\oplus$</th>
<th>$\otimes$</th>
<th>$\bar{0}$</th>
<th>$\bar{1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>${0, 1}$</td>
<td>$\lor$</td>
<td>$\land$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Probability</td>
<td>$\mathbb{R}$</td>
<td>$+$</td>
<td>$\times$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Log</td>
<td>$\mathbb{R} \cup {-\infty, +\infty}$</td>
<td>$\oplus_{\log}$</td>
<td>$+$</td>
<td>$+\infty$</td>
<td>0</td>
</tr>
<tr>
<td>Tropical</td>
<td>$\mathbb{R} \cup {-\infty, +\infty}$</td>
<td>$\min$</td>
<td>$+$</td>
<td>$+\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

Where $\lor$ logical or and $\land$ logical and $\oplus_{\log}$ is

$x \oplus_{\log} y = -\log(e^{-x} + e^{-y})$

Only the operations needed to:
- Compute the weight of a path using the $\otimes$ operator
- Compute the cost of sequence by summing over all the paths using the $\oplus$ operator
Semirings

- **Boolean semiring**
  - Un-weighted machines are equivalent to weighted machines over the Boolean semiring

- **Real (Probability) semiring**
  - The weights represent real numbers (or probabilities)

- **Log semiring**
  - Weights are negative logs of the probabilities

- **Tropical semiring**
  - Same as log semiring with the $\oplus$ replaced with the Viterbi approximation.

- Log and tropical semirings have the conventions $\exp(-\infty) = 0$ and $-\log(0) = \infty$ [2]
• $\Sigma^*$ (Sigma star) – is the Kleene closure of an alphabet $\Sigma$ and represents the set of all finite strings over the alphabet $\Sigma$

• **Simple path** – is a path with no cycle

• A semiring is **idempotent** if $x \oplus x = x$ for all $x \in S$ (Boolean and tropical semirings) [13]

• **Commutative** – ”A commutative semiring has a commutative multiplicative operator $\otimes$” [13]

• **Idempotent** – A semiring is idempotent if $x \oplus x = x$ for all $x \in S$ (quote) [13]

• **Mirror Image** – ”The mirror image of a string $x = x_1, x_2, \ldots, x_n$ is the string $x^R = x_n, x_{n-1}, \ldots, x_1$” (quote) [13]
• **Deterministic** – Has one initial state and at any state no more than one outgoing arc with any input label

• **Sequential** – Are deterministic on one side

• **Sub-sequential** – Have an additional string at the final state

• **p-Sub-sequential** – Have an \( p \) additional strings at the final state

• **Stochastic** – The sum of arc weights leaving a state is one

• **Trim** – Trim Transducers have no useless states
Commutative – \( a \otimes (b \oplus c) = a \otimes b + a \otimes c = (b \oplus c) \otimes a \) [18]

Regulated – A sum does not depend on the order of the terms

Unambiguous – For any string \( x \in \Sigma^* \) it has one accepting/successful path at most

Cycle – A cycle is a path where the initial and states are the same

\( \epsilon \)-cycle – Is a cycle where the input and output labels are epsilons

Successful – Path is a path from the initial to final states

non-accessible – Not reachable from an initial state

non-co-accessible – Not reachable from a final state
A transition $e$ from the set transitions $E$ is described using the following [2]

$p[e]$ denotes the origin state ($0$ in this example)
$n[e]$ denotes the next state ($1$ in this example)
$i[e]$ denotes the input label ($ip$ in this example)
$o[e]$ denotes the output label ($op$ in this example)
$w[e]$ denotes the weight ($0.5$ in this example)
A path $\pi$ is constructed from a sequence of transitions through the machine according to $\pi = e_1, \ldots, e_k$

- A path $\pi$ is a member of $E^*$ (The Kleene closure of $E$) [13]
- $p[\pi]$ denotes the origin state [13]
- $n[\pi]$ denotes the next state [13]
- A cycle is a path where $p[\pi] = n[\pi]$ [13]
- A path weight in the Log or Tropical semirings is identical because the multiplication operation is identical [2]
- $w[\pi]$ is the weight of the path obtained by using the $\otimes$ to the consecutive path weights [13]
- $P(Q_1, Q_2)$ denotes all the paths from state $Q_1$ to state $Q_2$ [13]
Weighted Finite State Transducers

Weighted Acceptors

- In an *Acceptor A* each transition has an input label
  - In the *weighted* case the transitions can be augmented with an optional weight
- \( P(Q_1, x, Q_2) \) denotes all the paths from state \( Q_1 \) to state \( Q_2 \) that accept the string \( x \) [13]
- \( [A](x) \) denotes the sum of all paths with input sequence \( x \) [13]
- \( [A](x) = \bigoplus_{\pi \in P(I, x, F)} \lambda(p[\pi]) \otimes w[\pi] \otimes \rho(n[\pi]) \) [13]
- Used to recognize string
In the real semiring

\[ [A] (a, b) = (1 \times 1 \times 1) + (3 \times 3 \times 1) = 10 \]
In weighted transducers the arcs have *input* and *output* labels with optional weight [13]

\[ P(Q_1, x, y Q_2) \] denotes all the paths from state \( Q_1 \) to state \( Q_2 \) that transduces the string \( x \) to the string \( y \) [13]

\[ \mathbb{K}(x, y) \] denotes the sum of all paths with input sequence \( x \) [13]

\[
\mathbb{K}(x, y) = \bigoplus_{\pi \in P(I, x, y, F)} \lambda(p[\pi]) \otimes w[\pi] \otimes \rho(n[\pi]) \ [4, 13]
\]

Used to perform a mapping between strings with a weight [13]
In the real semiring

\[
\text{\begin{bmatrix} A \end{bmatrix}} (bb, XZ) = 4 \times 3 \times 1 = 10
\]
When using the tool-kits several levels of representation

- Text level - text files describing the WFST topology and symbols
- Binary level - compiled binary format from the compilation tool
- Graph level - graphical representation can be one of the many image formats generated from graphviz

Introduce the representations

- Text level WFSTs need to be compiled before they can be used
- Print - to recover the text level
- Drawing - to visualize the WFST
For OpenFst and ATT FSM Toolkit text format is 3, 4 or 5 columns:
- Acceptors have 3 columns optional 4th column the weights
- Transducers have 4 columns with optional 5th column representing the weights

ATT fsmcompile defaults to acceptor format – use -t to specify transducer

OpenFst fstcompile defaults to transducer format – use -acceptor=true to specify acceptor

Can use a transducer to emulate an acceptor by setting the input and output labels the same for each arc

Final states single entry on own line with an optional final cost
Simple Acceptor Example

**Un-weighted acceptor definition**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>Tokyo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Kyoto</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Yokohama</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Nagoya</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Osaka</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Corresponding symbols**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokyo</td>
<td>1</td>
</tr>
<tr>
<td>Kyoto</td>
<td>2</td>
</tr>
<tr>
<td>Yokohama</td>
<td>3</td>
</tr>
<tr>
<td>Nagoya</td>
<td>4</td>
</tr>
<tr>
<td>Osaka</td>
<td>5</td>
</tr>
</tbody>
</table>
**Simple Transducer Example**

Weighted transducer definition:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>a</td>
<td>Z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>b</td>
<td>X</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a</td>
<td>X</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>b</td>
<td>Z</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>b</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>b</td>
<td>Z</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>b</td>
<td>Z</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>b</td>
<td>X</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Corresponding input symbols:

- 0
- a 1
- b 2

Corresponding output symbols:

- 0
- X 1
- Z 2
Printing and Drawing WFSTs

- Printing WFST
  - Can recover the text format from a binary format
- Drawing WFSTs
  - Very useful for understanding
  - Debugging and performing postmortems
  - Use Graphviz
  - Viewing large graphs is a problem
Graphical Representation

- States are circles
- Arcs are lines
- Initial state thick line
- Final state two lines on state
- Each arc label is represented as in_label:out_label/weight
  - in_label
  - out_label
  - weight when no weight is shown or given defaults to 1
A Few Additional Points

- A final state can have outgoing arcs
- An initial state can also be a final state
  - Double bold circle
- Not possible with ATT/OpenFst but allowable in general
  - Non final states can have a cost
  - Machines can have more than one initial state
  - However, both of these are easy to simulate
Graphviz (dot) had some font hard wiring for postscript
Latest version works with the pdf backend

```
fstdraw -acceptor=true -isymbols=jpn.syms -portrait=true jpn.ofsa | dot -Tpdf > jpn.pdf
```
Weighted Finite State Transducers

Null Transitions

- Null or epsilon (\(\epsilon\)) transitions
- Consume no input symbols when traversing
- Generate no output symbols when traversing
- Make certain algorithms more complicated
- Why they occur:
  - Inserted by certain algorithm
  - Needed in transducers to perform one-to-many mapping, cycles...
- Can always be removed from an WFA
- Impossible to remove from certain WFST, one-to-many mapping, cycles...
- Introduce non-determinism
Rational Operations

For the presented algorithms:

- Useful for combining many simpler machines to create more complex ones
- Algorithms are efficient and straightforward to implement
- Admit *on-the-fly* construction
- Introduce epsilons
Weighted Finite State Transducers

Sum (Union)

\[
[T_1 \oplus T_2](x, y) = [T_1](x_1, y_1) \oplus [T_2](x_2, y_2)
\]

fsmunion sapporo.fsm sendai.fsm kumamoto.fsm > cities.fsm
fstop sapporo.fsm sendai.fsm | fsunion - kumamoto.fsm >
Weighted Finite State Transducers

Product (Concatenation)

\[
\left[ T_1 \otimes T_2 \right] (x, y) = \bigoplus \left[ T_1 \right] (x_1, y_1) \otimes \left[ T_2 \right] (x_2, y_2)
\]

\[
x = x_1 x_2 \\
y = y_1 y_2
\]

```
fsmconcat grammar.fsm stations.fsm > gs.fsm
fstconcat grammar.fsm stations.fsm > gs.fsm
```
Closure

$$\left[ T^* \right](x, y) = \bigoplus_{n=0}^{\infty} \left[ T^n \right](x, y)$$ [4]

fsmclosure cities.fsm > citiesclosed.fsm
fstclosure cities.fsm > citiesclosed.fsm
Weighted Finite State Transducers

Reversal

- Input is a single weighted transducer $T$
- Output is a single reversed weighted transducer $\tilde{T}$
- $[\tilde{T}] (x, y) = [T] (\tilde{x}, \tilde{y})$ [4]
\( T^{-1}(x, y)) = T(y, x) \) \cite{13}

- Input is a single weighted transducer \( T \)
- Output is a single weighted transducer \( T^{-1} \) with the input and output labels reversed

Example based on \cite{13}
Input is a single weighted transducer $T$

Output a single acceptor with either:

- $\Pi_1 T$ – The output labels removed
- $\Pi_2 T$ – The input labels removed
Composition

- Very powerful operation
- Allows multiple levels of information to be combined
- Formal definition

\[ [T_1 \circ T_2] ((x, y)) = \bigoplus_z [T_1] (x, z) \otimes [T_2] (z, y) \]
Epsilon Free Composition

- Given a state $q_1$ in $T_1$ and a state $q_2$ in $T_2$
- Cycle through the arc combinations
- If an output symbol matches an input symbol
- Create a new arc

(a) $T_1$

(b) $T_2$

(c) $T_1 \circ T_2$
The presence of epsilons makes composition more difficult

Canonical example from Mohri [14]
Canonical Example

- Create $\tilde{T}_1$ by replacing $\epsilon$ outputs with $\epsilon_2$ adding $\epsilon : \epsilon_1$ self loops to every state
- Create $\tilde{T}_2$ by replacing $\epsilon$ inputs with $\epsilon_1$ adding $\epsilon_2 : \epsilon$ self loops to every state
- Compose use standard procedure and allow the match $\epsilon_2 \epsilon_1$

(a) $\tilde{T}_1$

(b) $\tilde{T}_2$
Path Multiplicity Problem

- A problem is the composed machine has multiple paths, when only one is needed.
- Therefore costs assigned to paths are wrong in certain semiring (non-idempotent (e.g. probability, log)).
We need to keep just one of the paths
Filter out unwanted paths
The solutions is to use a composition *filter*. This will restrict the paths that can be made in the composed machine. This can be represented as an WFST, where $x$ represents any symbol.

![Diagram of WFST](image-url)
It is now possible to perform composition of all 3 together using the standard algorithm

\[ \tilde{T}_1 \circ F \circ \tilde{T}_2 \]
The following factors can be used to make composition more efficient.

- Sorting or index the output of T1 and the input of T2
  - ATT – indexed with `fsmconvert`
  - OpenFst – sort with `fstarcosrt`

- Favour placing outputs labels early to avoid matching delays and the generation of dead-end states

- ATT FSM toolkit has indexed types that faster provide faster composition, for OpenFst use the arc sorting features
Determinization

- An automaton is deterministic if it meets the following conditions:
  - A single initial state
  - No two arcs leaving a state with the same input label
  - OpenFst and ATT FSM library treat $\epsilon$ the same as a normal symbol
- Determinization takes a WFST and attempts to construct a deterministic equivalent
- Along with composition and shortest path algorithm extremely useful
- Huge impact on search network efficiency
- Matching is much more efficient
- Caveats:
  - Doesn’t terminate in some cases or in a reasonable amount of time in other cases
  - Not all automata can be determinized
- Toolkits are extended to deal with transducers
Example taken verbatim from Mohri [13]
- Generalized versions of prefix tree
- Consider the toy lexicon
After applying determinization
Minimization

- Minimization takes a deterministic WFA and returns an equivalent WFA with the minimal amount of states.
- First pushes the costs in the machine.
- Encodes as a costless acceptor.
- Performs the minimization.
  - The OpenFst minimize command can accept transducers but under the hood will convert to an acceptor.
  - The ATT tools require that WFSTs are encoded as acceptors.
- Seems to have little improvement on speech recognition speed.
- Makes a small size reduction on speech recognition transducers.
- Choice of semiring can negatively affect performance.[15]
Example taken verbatim from Mohri [13]
Minimization will share the suffixes in the toy lexicon
Weighted Finite State Transducers

Weight Pushing

- Weights can either be pushed towards the initial or final states
- Tropical semiring example taken verbatim from [13]

![Diagram](a) Original

![Diagram](b) Pushed to initial

Tropical semiring example taken verbatim from [13]
Consider the previous example repeated in the real semiring.

In the pushed machine:
- The weights along the paths are normalized.
- The machine is stochastic – the weight leaving a state all sum to one.
- The path scores are still the same as the un-pushed machine.

(a) Original

(b) Pushed to initial
Epsilon Removal

- Epsilon transitions can harm WFST efficiency
- The epsilon removal algorithm attempts to remove these null transitions
- Use with care as it can lead to explosion in network size
- Not all epsilons can be removed
Openfst tool to convert between FST types
Currently no support for converting semirings
Types not well documented so here is a list dumped out from the FstRegister. All should be available in both Log and Tropical semirings

vector
compact16_acceptor
compact16_unweighted_acceptor
compact64_string
compact64_weighted_string
compact8_unweighted
compact_acceptor
compact_unweighted_acceptor
const16

const
compact16_string
compact16_weighted_string
compact64_unweighted
compact8_acceptor
compact8_unweighted_acceptor
compact_string
compact_weighted_string
compact64
compact16_unweighted
compact64_acceptor
compact64_unweighted_acceptor
compact8_string
compact8_weighted_string
compact_unweighted
const8
OpenFst `fstmap` applies transformations to the arc and final weights as illustrated.

(a) Original

(b) Identity
Map continued

(c) Invert

(d) Plus (2)
Weighted Finite State Transducers

Map continued

(e) Original

(f) Rmeight
Map continued

(g) Times (2)

(h) Superfinal
Section Summary

- Introduced WFSTs
- Covered the algorithms frequently used in speech recognition network construction
Questions?
Simple applications using WFSTs
This section will introduce two simple examples of using WFSTs:

- Basic edit distance computation using WFSTs
- More complicated counting example with a custom semiring
In this example use the shell level OpenFst tools to calculate the edit distance between two strings. The example demonstrates composition and shortest path operations[11, 5].

(a) WFST for Tokyo

(b) WFST for Kyoto
Edit Distance - Filter

- a
- b

0
fstcompose tokyo.ofst filter.ofst | fstcompose kyoto.ofst > tfk.ofst
Examples

Edit Distance - Best Path

fstshortestpath tfk.ofst > best.ofst
Counting Example

- In this example
- Use C++ and OpenFst to build a *grmcount*[2] clone
- Add a custom semiring to OpenFst
  - Create a user defined arc type
  - Build a shared objects that is loaded by OpenFst tools
- Count n–grams in text corpus
- Makes use of the following operations:
  - *Composition* between the corpus and counting machine
  - *Projection*, *Epsilon Removal*, *Determinization* and *Minimization* to generate a compact set of counts
  - *Composition* and *ShortestDistance* to extract the count for a specific n–gram
Counting Example

- From [2] calculate the expected count using
  \[ c(x) = \sum_{u \in \Sigma} |u|_{x} [A](x) \]
- Where \([A](x)\) is the probability of \(x\) in \(A\)
- The below WFST implements the counting where \(X : X\) denotes an arbitrary transducer [12]

The loops at state 0 consume unwanted input mapping them to epsilons
- The machine \(X\) will match the desired input(s) patterns
- The end loops will then map the unwanted trailing input to epsilons
Counting Example

- Project outputs and remove epsilons will give a path for each match
- Optionally determinize and minimize for a compact representation
- Can be applied to lattices that cannot be enumerated [2]
- Extended to class models and used for training WFSTs by Murat and Roark[19]
- To compute $c(x)$ for a single n-gram compose with a linear chain automata representing the input $x$ and use the shortest distance algorithm to obtain the sum over all paths
- Or compose with linear chain transducer with epsilons outputs and remove epsilons to give a single state and weight
First Add Real Semiring

- In practice can just use the log semiring.
  - For numerical stability
  - Already registered with all the tools (printing, drawing ...)

- Serves as example to illustrate OpenFst extensibility

- Therefore the algorithms and tools should work with the custom semiring

- Real semiring does have practical application, for example Shu and Hethering[20] used the real semiring for EM training of FSTs
Implement Real Weight

- Real (probability) semiring $+, \times, 0, 1$
- OpenFst weight requirements are on the site here [21] and here[3]
- The `floatweight` type can be used to do most of the boiler plate work
- Essential to implement the following static methods `Plus`, `Times`, `Zero`, `One` and `Type`
- Implemented in `real-weight.h`
Register Custom Arc Type

Now add support for the real semiring in the shell tools

```
#include <fst/arc.h>
#include <fst/vector-fst.h>
#include <fst/const-fst.h>
#include <fst/mains.h>
#include "real-weight.h"

using namespace fst;
extern "C"
{
    void real_arc_init()
    {
        typedef ArcTpl<RealWeight> RealArc;
        REGISTER_FST(VectorFst, RealArc);
        REGISTER_FST(ConstFst, RealArc);
        REGISTER_FST_MAINS(RealArc);
    }
}
```

g++ -02 -fPIC real-arc.cpp -o real-arc.o -c
-I/path/to/openfst-1.1/src/bin/
g++ -shared -o real-arc.so real-arc.o

Final step add shared object path to LD_LIBRARY_PATH
The Counting Utility

- `acount`
- Input a text file
- Output the compact `count` machine and
- All the intermediate steps and the unigram counts
- Usage: `acount input.txt output.dir order.int`
Sample Text From APSIPA Site

at the centre of sapporo the city hall and the sapporo station are located at the end of an alluvial fan at 20m above sea level while the salmon museum in makomanai is at an altitude of 80m the distance between the two places is about 8 kilometers between the the total superficies of the alluvial fan is approximately 20 square kilometers the central part of sapporo was formed 6000 years before by the deposit of earth carried by the toyohira river from jozankei and was frequently flooded in the 19th century when the river banks were not built yet there is abundant ground water away from the riverbed due to the river underflows since there is no need to dig deep wells to draw an high quality water life is easy on this fertile land which was used for agriculture including the culture of fruit trees

Illustrate the process counting bigram and unigrams on a single sentence (modified)

the distance between the two places is about 8 kilometers between the

/acount sample.sentance sentance.output 2
Real Semiring Arc Registration

Run `fstinfo corpus.ofst`, and if something went wrong

ERROR: FstMainRegister::GetMain: real-arc.so: cannot open shared object file: No such file or directory
ERROR: fstprint: Bad or unknown arc type "real" for this operation (PrintMain)

Otherwise:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fst type</td>
<td>vector</td>
</tr>
<tr>
<td>arc type</td>
<td>real</td>
</tr>
<tr>
<td>input symbol table</td>
<td>none</td>
</tr>
<tr>
<td>output symbol table</td>
<td>none</td>
</tr>
<tr>
<td># of states</td>
<td>13</td>
</tr>
<tr>
<td># of arcs</td>
<td>12</td>
</tr>
<tr>
<td>initial state</td>
<td>0</td>
</tr>
<tr>
<td># of final states</td>
<td>1</td>
</tr>
<tr>
<td># of input/output epsilons</td>
<td>0</td>
</tr>
<tr>
<td># of input epsilons</td>
<td>0</td>
</tr>
<tr>
<td># of output epsilons</td>
<td>0</td>
</tr>
<tr>
<td># of accessible states</td>
<td>13</td>
</tr>
<tr>
<td># of coaccessible states</td>
<td>13</td>
</tr>
<tr>
<td># of connected states</td>
<td>13</td>
</tr>
<tr>
<td># of strongly conn components</td>
<td>13</td>
</tr>
<tr>
<td>expanded</td>
<td>y</td>
</tr>
<tr>
<td>mutable</td>
<td>y</td>
</tr>
<tr>
<td>acceptor</td>
<td>y</td>
</tr>
<tr>
<td>input deterministic</td>
<td>y</td>
</tr>
<tr>
<td>output deterministic</td>
<td>y</td>
</tr>
<tr>
<td>input/output epsilons</td>
<td>n</td>
</tr>
<tr>
<td>input epsilons</td>
<td>n</td>
</tr>
<tr>
<td>output epsilons</td>
<td>n</td>
</tr>
<tr>
<td>input label sorted</td>
<td>y</td>
</tr>
<tr>
<td>output label sorted</td>
<td>y</td>
</tr>
<tr>
<td>weighted</td>
<td>n</td>
</tr>
<tr>
<td>cyclic</td>
<td>n</td>
</tr>
<tr>
<td>cyclic at initial state</td>
<td>n</td>
</tr>
</tbody>
</table>
The distance between the two places is about 8 kilometers.
Projected Machine
Epsilon Removed Machine

**Examples**

0

- distance
- the
- between
- distance
- the
- between
- two
- the
- places
- two
- places
- is
- about
- is
- about
- kilometers
- the

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

About 8 kilometers between the places is about 8 kilometers. Two places is about 8 kilometers. Two places is about 8 kilometers. Two places is about 8 kilometers.
Optimized Machines

Determinized

Minimized
the distance between the two places is about 8 kilometers between the

the/4
distance/2
two/2
places/2
is/2
about/2
8/2
8/5
kilometers/0.5

the/4
distance/2
two/2
places/2
is/2
about/2
8/2
8/5
kilometers/0.5

the/2
distance/1
between/1
two/1
places/1
is/1
about/1
about/1
8/1
8/1
kilometers/1

the/2
distance/1
between/1
two/1
places/1
is/1
about/1
about/1
8/1
8/1
kilometers/1
Extracting Counts

- **Method 1 - ShortestDistance**
  
  between \(0\) the \(2\)
  
  
  \[
  \text{fstcompile} \ --\text{arc\_type}=\text{real} \ \text{--isymbols}=\text{words} \ . \ \text{syms} \ \text{--osymbols}=\text{words} \ . \ \text{syms} \ \text{betweenthe} \ . \ \text{wfst} \ |
  \]
  
  \[
  \text{fstcompose} \ \text{min} \ . \ \text{ofst} - \ | \ \text{fstshortestdistance} \ \text{--reverse}=\text{true}
  \]
  
  0 2
  
  1 0.5
  
  2 1

- **Method 2 - Projection and Epsilon Removal**

  between:- \(0\) the:- \(2\)

  
  
  \[
  \text{fstcompile} \ --\text{arc\_type}=\text{real} \ \text{--isymbols}=\text{words} \ . \ \text{syms} \ \text{--osymbols}=\text{words} \ . \ \text{syms} \ \text{the} \ . \ \text{wfst} \ |
  \]
  
  \[
  \text{fstcompose} \ \text{min} \ . \ \text{ofst} - \ | \ \text{fstproject} \ \text{--project\_output}=\text{true} \ \text{|fstrmepsilon|} \ \text{fstprint}
  \]

  0 2

  Don’t need to perform the previous optimizations to obtain counts with this method
Examples

Compare to a counting utility

Finally, compare account to the counting tool in the MitLM[9] toolkit

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>2</td>
<td>alluvial</td>
<td>2</td>
</tr>
<tr>
<td>alluvial</td>
<td>2</td>
<td>and</td>
<td>2</td>
</tr>
<tr>
<td>fan</td>
<td>2</td>
<td>by</td>
<td>2</td>
</tr>
<tr>
<td>in</td>
<td>2</td>
<td>fan</td>
<td>2</td>
</tr>
<tr>
<td>kilometers</td>
<td>2</td>
<td>from</td>
<td>2</td>
</tr>
<tr>
<td>by</td>
<td>2</td>
<td>in</td>
<td>2</td>
</tr>
<tr>
<td>there</td>
<td>2</td>
<td>kilometers</td>
<td>2</td>
</tr>
<tr>
<td>from</td>
<td>2</td>
<td>there</td>
<td>2</td>
</tr>
<tr>
<td>water</td>
<td>2</td>
<td>water</td>
<td>2</td>
</tr>
<tr>
<td>an</td>
<td>3</td>
<td>an</td>
<td>3</td>
</tr>
<tr>
<td>sapporo</td>
<td>3</td>
<td>river</td>
<td>3</td>
</tr>
<tr>
<td>was</td>
<td>3</td>
<td>sapporo</td>
<td>3</td>
</tr>
<tr>
<td>river</td>
<td>3</td>
<td>to</td>
<td>3</td>
</tr>
<tr>
<td>to</td>
<td>3</td>
<td>was</td>
<td>3</td>
</tr>
<tr>
<td>at</td>
<td>4</td>
<td>at</td>
<td>4</td>
</tr>
<tr>
<td>is</td>
<td>6</td>
<td>is</td>
<td>6</td>
</tr>
<tr>
<td>of</td>
<td>7</td>
<td>of</td>
<td>7</td>
</tr>
<tr>
<td>the</td>
<td>17</td>
<td>the</td>
<td>17</td>
</tr>
</tbody>
</table>
Section Summary

- Basic edit distance computation using WFSTs
- More complicated counting example with a custom semiring
Questions?
Speech Recognition with WFSTs
Overview of Section

- Detailed walkthrough on the construction of real world large vocabulary integrated WFST speech recognition system
- Much more practical than other tutorials
- Based on the procedure from Mohri [15]
- Rules of thumb and things to avoid
  - Illustrate what happens when these guidelines are and are not used
- The final recipe is very small
- It should provide a process that builds a good integrated recognition WFST
Overview of Task

- Use Corpus of Spontaneous Japanese to illustrate construction methods and toolkits
- Standard large vocabulary Japanese task
  - 233 hour of acoustic training data
  - 116 minutes of acoustic testing data
- Acoustic Models have 3000 states and 128 Gaussian per state
- Vocabulary 65k words where a word is a base form – POS Tag – Reading
- Evaluate word accuracy after stripping POS tags and reading
- Use $T^3$ decoder with GPU acoustic scoring [6–8]
- Decode on a set of beam widths and see how the construction influences the RTF/WER relationship
WFSTs In Speech Recognition

- Represent each knowledge source as a WFST, typically
- Language Model – G
- Lexicon – L
- Context Dependency – C
- Acoustic Models – H
- Combine using composition – $H \circ C \circ L \circ G$
- Optimize final and intermediate WFSTs
- Use in decoder
Use the process described in the paper [15]
The cascade will be $\pi \det(C \circ \det(L \circ G))$
Mostly use the ATT tools for the walkthrough
Will also include a comparison to OpenFst
Should be straightforward to implement using a scripting language
WFSTs allow for highly efficient networks to be constructed
Use as many optimizations as possible
  The decoder should go really fast?
  The search network should get really small?
The construction process can be very fragile
Certain transducers need to be constructed in specific ways
Applying too many optimizations
  Often doesn’t improve performance or network size
  Can make performance worse
  Can make it impossible to construct the final network
Language Model

- Language model is converted to WFST
- Finite state grammar is trivial
- Also possible to efficiently represent an n–gram as WFST\[1\]
- Use epsilons transitions to \textit{back-off} states to reduce the network size \[1, 15\]
Trigram Example

- $P()$ is the n-gram probability, $B()$ is the back-off weight
- The states labels encodes the history
- Example from [5]
Random strings generated (by an FSA) with an alphabet of two words

b a a a a a a b a a a
b b
b b
b b
a b a b b
a
a
a a b a a
b b
a a
ARPA format LM trained with mitlm using estimate-ngram -o2 -t ab.train -wl ab.lm

\data\ngram 1=4
ngram 2=8

\1-grams:
-0.477121 </s>
-99 </s>-0.176091
-0.477121 b -0.124939
-0.477121 a -0.324511

\2-grams:
-0.477121 <s> b
-0.352183 <s> a
-0.477121 b </s>
-0.477121 b b
-0.477121 b a
-0.579784 a </s>
-0.676694 a b
-0.278754 a a

\end\
• Converted to an WFST
Other Points for ARPA n–gram Models

- Use the 99 value to set the start state
- Use concatenation if you want a sentence begin tag
- For end states:
  - Can’t assume no back-off implies the final state. When using a restricted vocabulary list or SRILM some back-off values might be missing
  - Instead detect sentence end tags \(</s>\) and set the corresponding state as final
  - Also remember the higher n–grams \(a \ b \ </s>\) will have a final state labelled \(b, <s>\)
  - Direct them to a unique final state
- Remember to convert log base
- Trigram built from CSJ using ATT GRM Tools
- Training set of approx 400,000 sentences
- Katz smoothing and shrinking factor of 1

<table>
<thead>
<tr>
<th>WFST</th>
<th>States</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>299634</td>
<td>1422541</td>
</tr>
</tbody>
</table>
Lexicon

- Built from the pronunciation dictionary
- Maps sequences of phonemes to words
- Create a chain transducer for each word in the lexicon
- Take the union
- Perform closure
  - So we can looping through the machine
Illustrate the procedure on a random subset from the CSJ dictionary

共存 ky o: s o N
あの an o
額絵 g a k u e
サードネス sa: d o n e s u
均一 ki N i ts u
履歴 rer i ki
見回し m i m a: sh i
郎 ro:
危険 ki ke N
ゆうこ y u u k o
For each of the entries construct a linear chain transducer:
Lexicon Example

Take the union of all words to build lexicon WFST

Diagram:

- ky: 共存
- a: あの
- g: 額絵
- s: サードネス
- k: 均一
- r: 履歴
- m: 見回し
- r: 郎
- k: 危険
- y: ゆうこ

Diagram nodes and transitions:

1. お
2. す
3. お
4. N
5. 0
6. な
7. お
8. 9
9. a
10. き
11. う
12. え
13. 14
14. a
15. で
16. お
17. に
18. e
19. s
20. u
21. 22. い
23. N
24. い
25. つ
26. u
27. 28. え
29. r
30. い
31. き
32. い
33. 34. い
35. m
36. a
37. sh
38. i
39. 40. お
41. 42. い
43. き
44. え
45. N
46. 47. う
48. う
49. き
50. o
51. 52. 6
53. あ
54. の
55. 額
56. 絵
57. 58. 59. 26
Perform Closure

<table>
<thead>
<tr>
<th>Node</th>
<th>Label</th>
<th>Next Node</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>1</td>
<td>s:サードネス</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>ky:共存</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>a:あの</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>g:額絵</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>k:危険</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>r:郎</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>r:履歴</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>y:ゆうこ</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>m:見回し</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>d:</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>a::-</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>d:-</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>o:-</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>k:-</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>i:-</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>n:-</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>e:-</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>s:-</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
</tr>
<tr>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>52</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: The table shows the nodes and their connections, along with labels for each node. The labels include characters in Japanese and English, indicating different types of data or entities.
- Easier just to perform the previous step as a single script
  - Simple to do in a single pass
  - Probably not a good idea to create 100k small files
  - Easier to add the auxiliary symbols

- Place output labels early in the path
  - The epsilons will delays in the matching and generate many dead end states
  - Homophones prevent determinization – add auxiliary symbols
  - Avoid optimizing the lexicon - because this will push the output labels further back
  - Sorting (or indexing) arcs is crucial in large system
Create two lexicons:
- A lexicon with outputs labels at the start of the chains $L^s$
- A lexicon with outputs labels at the end of the chains $L^e$

Both $L^s$ and $L^e$ have the same amount of states and arcs

Compose $L^s$ or $L^e$ with $G$

The labels are indexed on the input of $G$ and the output of $L$

$L^s \circ G$ took 4.5 seconds

$L^e \circ G$ took...
Good and bad label placement

- Occurs, regardless of filter, back-off epsilon treatment or toolkit
- Repeat again on a 3000 word lexicon
- Compose again without connection `fsmcompose -d ls.fsm g.fsm`

<table>
<thead>
<tr>
<th></th>
<th>$L^s \circ G$</th>
<th>trim $L^s \circ G$</th>
<th>$L^e \circ G$</th>
<th>trim $L^e \circ G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>46258</td>
<td>30710</td>
<td>66520834</td>
<td>66848</td>
</tr>
<tr>
<td>Arcs</td>
<td>58507</td>
<td>42932</td>
<td>68004350</td>
<td>819747</td>
</tr>
<tr>
<td>Accessible States</td>
<td>46258</td>
<td>30710</td>
<td>67992311</td>
<td>66848</td>
</tr>
<tr>
<td>Co-accessible States</td>
<td>30710</td>
<td>30710</td>
<td>66848</td>
<td>66848</td>
</tr>
</tbody>
</table>
Sorting or Indexing of arcs is also important

Informal comparison of composition with indexed, un-indexed machines

Un-indexed composition took 4.5 minutes

Indexed composition took approx. 5 seconds

OpenFst will/should complain if arcs aren’t sorted
The lexicon may not be determinizable because of homophones
Verified on the walkthrough lexicon anyway
Determinization just runs out of memory
Solution is to add auxiliary symbols to disambiguate the homophones
Consider the toy lexicon with an additional homophone:

共存 ky o: s o N
あの a n o
額絵 g a k u e
サードネス s a: d o n e s u
均一 k i N i t s u
履歴 r e r i k i
見回し m i m a: sh i
郎 r o:
危険 k i k e N
ゆうこ y u u k o
棄権 k i k e N

Add an auxiliary phone $\#_n$ to each pronunciation and alter $n$ to disambiguate to the homophones.

It is now possible to determinize the lexicon.
Lexicon Augmented With Auxiliary Symbols
The ATT include a special determinization algorithm to atomically add disambiguation symbols

```
fsmdeterminize -l 1000 in.fsm > out.fsm
```

Auxiliary symbols begin from numbers labeled 1000

Print out the wfst to get the symbols for later removal

Even using this algorithm – avoid determinizing the lexicon before composition
Lexicon Example

Determinize to generate prefix tree
Minimize to share common suffixes
Optimizing $L$ makes it smaller
But determinization pushes the labels back
We have seen this can severely damage the efficiency of composition
Verified the composition of $L$ or $L_{opt}$ with $G$
The composition of any $L_{opt}$ permutation fails
Confirms it is best not to optimize $L$ at this stage
The labels placements are optimized

The auxiliary symbols are in place

However using the *Epsilon Matching* filter and the epsilons in the language model will cause problems

One solution is to replace epsilon transitions in $G$

  - Replace epsilon back-offs with an alternative symbol (usually $\phi$)
  - Add a $\phi$ loop to the lexicon

Easier solution is to use a *Sequence* filter

  - ATT fsmcompose with –s option
  - OpenFst defaults to sequence filter
## Walkthrough LG

<table>
<thead>
<tr>
<th>WFST</th>
<th>States</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>299634</td>
<td>1422541</td>
</tr>
<tr>
<td>L</td>
<td>401704</td>
<td>466703</td>
</tr>
<tr>
<td>LG</td>
<td>1182138</td>
<td>2383541</td>
</tr>
<tr>
<td>det(LG)</td>
<td>1410016</td>
<td>2613069</td>
</tr>
</tbody>
</table>
Context-Dependency

- Map from context dependent sequences to context independent sequence
- Often the acoustic models contain only subset of tri-phones
- First construct a network that can handle all possible tri-phones
- Then replace with actual tri-phones
- Add arcs according to the following rules:
  - Use a delayed (and inverted) construction to ensure determinzability [15]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Source</th>
<th>Destination</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tri-phone</td>
<td>l-c+r</td>
<td>l,c</td>
<td>c,r</td>
<td>r</td>
<td>l-c+r</td>
</tr>
<tr>
<td>Right Bi-phone</td>
<td>c+r</td>
<td>-,c</td>
<td>c,r</td>
<td>r</td>
<td>c+r</td>
</tr>
<tr>
<td>Left Bi-phone</td>
<td>l-c</td>
<td>l,c</td>
<td>c,- (Final)</td>
<td>-</td>
<td>l-c</td>
</tr>
<tr>
<td>Mono-phone</td>
<td>c</td>
<td>– (Start)</td>
<td>–,c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>
Construct a chain automata from the string ababb
Compose with CD transducer
The resulting WFST maps from CI to CD phones

Additionally check a monophone a
Need to deal with the additional auxiliary phones
At each state in CD network add self loops that pass the auxiliary symbols through to the next level
Minimal example on the next slide
After performing the final set of optimizations replace the auxiliary symbols with epsilon symbols
Deterministic Context Dependency Construction with Auxiliary Symbols
## Walkthrough CLG

<table>
<thead>
<tr>
<th>WFST</th>
<th>States</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>299634</td>
<td>1422541</td>
</tr>
<tr>
<td>L</td>
<td>401704</td>
<td>466703</td>
</tr>
<tr>
<td>LG</td>
<td>1182138</td>
<td>2383541</td>
</tr>
<tr>
<td>det(LG)</td>
<td>1410016</td>
<td>2613069</td>
</tr>
<tr>
<td>C det(LG)</td>
<td>1466022</td>
<td>2738532</td>
</tr>
<tr>
<td>det ( C det (LG))</td>
<td>1466022</td>
<td>2738532</td>
</tr>
</tbody>
</table>
Final Steps

- Final processing step before recognition
- Remove auxiliary symbols
- Substitute logical Tri-phones with physical Tri-phones
## Walkthrough $\pi CLG$

<table>
<thead>
<tr>
<th>WFST</th>
<th>States</th>
<th>Arcs</th>
<th>Input Eps</th>
<th>Output Eps</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>299634</td>
<td>1422541</td>
<td>199631</td>
<td>199631</td>
</tr>
<tr>
<td>$L$</td>
<td>401704</td>
<td>466703</td>
<td>65001</td>
<td>401703</td>
</tr>
<tr>
<td>$LG$</td>
<td>1182138</td>
<td>2383541</td>
<td>404840</td>
<td>1182135</td>
</tr>
<tr>
<td>det($LG$)</td>
<td>1410016</td>
<td>2613069</td>
<td>434721</td>
<td>1418229</td>
</tr>
<tr>
<td>C det($LG$)</td>
<td>1466022</td>
<td>2738532</td>
<td>444095</td>
<td>1511787</td>
</tr>
<tr>
<td>det ( C det (LG))</td>
<td>1466022</td>
<td>2738532</td>
<td>444095</td>
<td>1511787</td>
</tr>
<tr>
<td>$\pi$ det ( C det (LG))</td>
<td>1466022</td>
<td>2738532</td>
<td>692946</td>
<td>1511787</td>
</tr>
</tbody>
</table>
Toy Language model $G$
$L \circ G$
$det(L \circ G)$
$C \circ \text{det}(L \circ G)$
Recognition Performance

![Graph showing Recognition Performance](image)

- Accuracy vs. RTF (Relative Time Factor)
- det(Cdet(LG)) as the red line
## Comparison of a few Construction Techniques

<table>
<thead>
<tr>
<th>WFST</th>
<th>States</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi \det(\mathbf{C} \det(\mathbf{LG}))$</td>
<td>1466022</td>
<td>2738532</td>
</tr>
<tr>
<td>Tropical $\pi \det(\mathbf{C} \det(\mathbf{LG}))$</td>
<td>1466022</td>
<td>2738532</td>
</tr>
<tr>
<td>$\pi \min(\det(\mathbf{C} \det(\mathbf{LG})))$</td>
<td>1397962</td>
<td>2643926</td>
</tr>
<tr>
<td>CLG</td>
<td>2036680</td>
<td>30169707</td>
</tr>
<tr>
<td>dmake</td>
<td>1205482</td>
<td>2614254</td>
</tr>
</tbody>
</table>
Performance with no Optimizations

![Graph showing the accuracy of speech recognition with different RTF values. The graph compares det(Cdet(LG)) and CLG. The accuracy increases as the RTF increases.]
Mohri[15] doesn’t mention the construction semiring

Allauzen[1] used log semiring for composition and determinization

The example presented here used the log semiring throughout for composition and determinization

Tropical semiring can be slightly faster but accuracy drops for narrow beams
Construction in the Log or Tropical Semirings

![Graph](image)

- det(C_{det(LG)}) - log
- det(C_{det(LG)}) - tropical

Accuracy vs. RTF
Comparison with OpenFst

- Using the tropical semiring
- The network size is identical on either toolkit
Construction Summary

- Presented a construction walkthrough
  - Simple to implement
  - Obtains good performance on a real LVCSR setup
- Many many other permutations and combinations
Extensions

- **On-the-fly**
  - The component WFSTs are composed dynamically as needed by the decoder during search
  - More sophisticated technique exists to make this faster [16, 17]

- Compose in the acoustic models

- In this case add in the $H$ machines

- Apply factorization to further reduce WFST size

- Take the composition a step further, compose with the acoustic observation $O$ to get $O \circ H \circ C \circ L \circ G$ and use shortest-path
  - This would not be practical for large systems
  - However the combination of *Composition* and *Shortestpath* can be used to build many useful applications for speech and language processing
POS Tagger Using Component Networks and On-the-fly Composition

- Just like in the speech recognition walk through
- Train an n-gram model
- Build a lexicon that convert words to tag–words pairs
- Compose and determinize
- Construct and input transducer from the input
- Compose and run to shortest path
- Possible to do the composition of the component networks on-the-fly
- Runtime system will use three WFSTs:
  - The input sequence
  - Optimized lexicon – Closed and determinized
  - Language model
- Extend to *On-the-fly* and only expand the parts of the composed machine as needed by the shortest-path algorithm
This is our input sequence
Compose with looped *lexicon*

This gives a lattice of possible constructions of the input sequence
- Score all the paths in the lattice by composing with the language model
- Easy to see many different types of language model can be used
- Multiple paths due to epsilon back-offs
Further Processing

- The optimizations have removed the synchronization between input and outputs
- Just want the output sequences
- Project output and remove epsilons
- Determinize and use shortestpath to obtain n–best lists
To convert to an automata representing word sequences
To give a compact automata
To share common prefixes and ensure n−best lists are unique
To get n–best paths run shortestpath
There are \( n \) paths in the final machine
Advantages

- We have demonstrated how useful application can be rapidly developed using WFSTs
- Much easier to extend – for example changing the language model in Chasen would be difficult but trivial in this previous example
- One framework and toolkit for nearly everything – not a mix of domain specific tools
Summary of Section

- Detailed description of the construction of an integrated speech recognition WFST
  - Conversion of the knowledge sources to WFSTs
  - Rules of thumb
  - Optimization
  - Evaluation on a large vocabulary task
- Applying a similar WFST framework to POS tagging
Thank You For Attending

- Questions?


