Finite State Transducer mechanisms in speech recognition

Daniel Povey

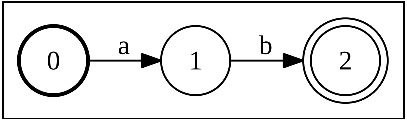
Microsoft Research

9/1/2011

Spoiler

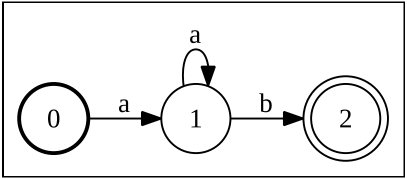
- "New" part of this talk is: a "perfect" lattice generation algorithm.
- Informally, when we generate a lattice for an utterance, we want it to:
  - Contain all sufficiently likely word sequences
  - Have the "correct" alignments and likelihoods for such sequences
  - Not be too large due to too-unlikely sequences or duplicate alignments
- Current lattice generation algorithms generally fall into two categories:
  - Store one traceback for each state $\rightarrow$ quick, but approximate.
  - Store N tracebacks $\rightarrow$ fairly exact, but slow.
- Our method stores one traceback, but is exact.

Finite State Acceptors (FSAs)



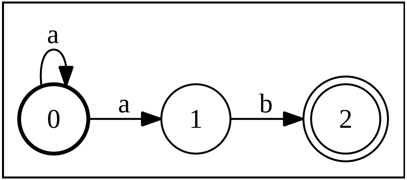
- An FSA "accepts" a set of strings
- (a string is a sequence of symbols).
- View FSA as a representation of a possibly infinite set of strings.
- This FSA accepts just the string ab, i.e. the set $\{ab\}$
- Numbers in circles are state labels (not really important).
- Labels are on arcs are the symbols.
- Start state(s) bold; final/accepting states have extra circle.
  - Note: it is sometimes assumed there is just one start state.

A less trivial FSA



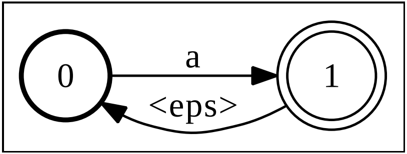
- The previous example doesn't show the power of FSAs because we could represent the set of strings finitely.
- This example represents the infinite set $\{ab, aab, aaab, \ldots\}$
- Note: a string is "accepted" (included in the set) if:
  - There is a path with that sequence of symbols on it.
  - That path is "successful' (starts at an initial state, ends at a final state).

Equivalence of FSAs



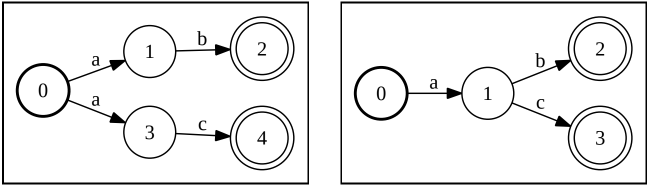
- This example represents the *same* infinite set as before $\{ab, aab, aaab, \ldots\}$
- ... but it looks different from the previous one.
- The FSAs are *equivalent* but not *equal*.

The ϵ (epsilon) symbol



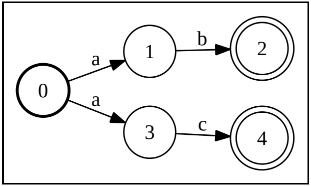
- The symbol ϵ has a special meaning in FSAs (and FSTs)
- It means "no symbol is there".
- This example represents the set of strings $\{a, aa, aaa, \ldots\}$
- If ϵ were treated as a normal symbol, this would be $\{a, a\epsilon a, a\epsilon a\epsilon a, \ldots\}$
- In text form, ϵ is sometimes written as <eps>
- Toolkits implementing FSAs/FSTs generally assume <eps> is the symbol numbered zero

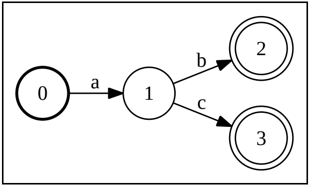
Deterministic FSAs



- The right hand FSA above is deterministic
- Means: no state has two outgoing arcs with the same label
- The classical definition of "deterministic" also forbids ϵ arcs.
- There is a definition that allows ϵ (Mohri/AT&T/OpenFst).
- Working out whether a given string (e.g. *ab*) is accepted is more efficient in deterministic FSAs.

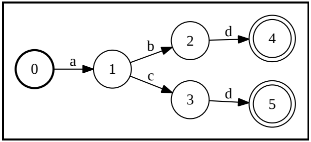
Determinizing FSAs

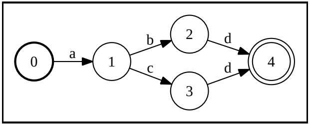




- There is a fairly easy algorithm to determinize FSAs
- Each state in the determinized FSA corresponds to a *set* of states in the original.
- The algorithm starts from the start state and uses a queue...
- In this example, states 1 and 3 on get combined as state 1.
- The classical approach has to treat ϵ specially.
- (Mohri's version simplifies things, making ϵ removal a separate stage).

Minimal deterministic FSAs





- Here, the left FSA is not minimal but the right one is.
- "Minimal" is normally only applied to *deterministic* FSAs.
- Think of it as suffix sharing, or combining redundant states.
- It's useful to save space (but not as crucial as determinization, for ASR).

Minimization of FSAs

- Common minimization algorithm partitions states into sets.
- Start out with a partition of just two sets (final/nonfinal).
- Keep splitting sets until we can't split any more.
- The sets are the states in the minimal FSA.
- Set splitting is based on yes/no criteria like "does a state have a transition with symbol a to a member of set s?"

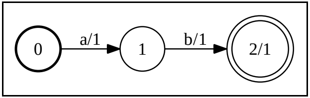
Other algorithms for FSAs

- Equality and equivalence testing
- Reversing (like reversing the arrows)
- Trimming (removing unreachable states)
- Union, difference (view these in terms of the "set of strings")
- Concatenation (of strings accepted by two FSAs)
- Closure (allowing arbitrary repetitions of the accepted strings)
- Epsilon removal

FSAs/FSTs and testability

- Many useful FSA/FST operations preserve equivalence (e.g. determinization, minimization)
- We can test equivalence
- Useful to test correctness of algorithms like determinization, minimization.
- E.g.: generate random FSA; determinize; check deterministic; check equivalent.
- This is one of the advantages of the FST framework
- You could hand-code algorithms to solve your specific problems (e.g. decoding), but could you test them?

Weighted Finite State Acceptors: normal case



- Like a normal FSA but with costs on the arcs and final-states
- Note: cost comes after "/". For final-state, "2/1" means final-cost 1 on state 2.
- View WFSA as a function from a string to a cost.
- In this view, unweighted FSA is f : string $\rightarrow \{0, \infty\}$.
- If multiple paths have the same string, take the one with the lowest cost.
- This example maps *ab* to $(3 = 1 + 1 + 1)$, all else to ∞.

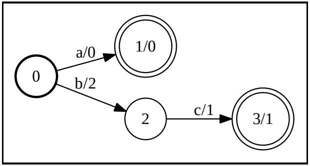
Semirings

- The semiring concept makes WFSAs more general.
- A semiring is
  - A set of elements (e.g. $\mathbb{R}$)
  - Two special elements $\bar{1}$ and $\bar{0}$ (the identity element and zero)
  - Two operations, $\oplus$ (plus) and $\times$ (times) satsifying certain axioms.
- Examples:
  - The reals, with $\oplus$ and $\otimes$ defined as the normal $+$ and $\times$, $\bar{1} = 1$, $\bar{0} = 0$.
  - The integers, as above.
  - The nonnegative reals, as above.
  - The nonnegative reals as above, except with $x \oplus y = \max(x, y)$.

Semirings and Weighted Finite State Acceptors (WFSAs)

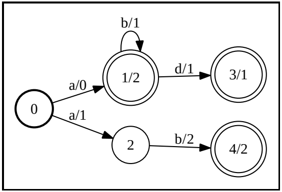
- The normal semirings used in ASR applications are
  - The tropical semiring, which is the simplest case ($\oplus$ means: take the minimum cost; $\times$ means, add the costs)
  - The log semiring, which is equivalent to the nonnegative reals, except representing them as negative log.
- There are also "fancier" semirings used (e.g.) in determinization, where the weight contains a string component.
- In WFSAs, weights are $\otimes$-multiplied along paths.
- Weights are $\oplus$-summed over paths with identical symbol-sequences.

Costs versus weights: terminology



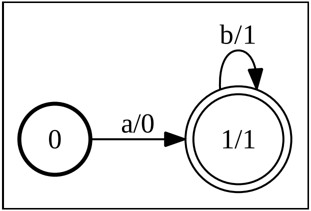
- Personally I use "cost" to refer to the numeric value, and "weight" when speaking abstractly, e.g.:
  - The acceptor above accepts *a* with unit weight.
  - It accepts *a* with zero cost.
  - It accepts *bc* with cost $4 = 2 + 1 + 1$
  - State 1 is final with unit weight.
  - The acceptor assigns zero weight to *xyz*.
  - It assigns infinite cost to *xyz*.

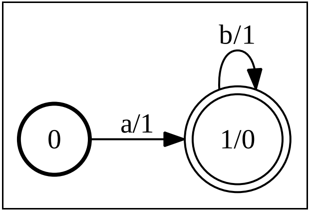
Function interpretation of WFSAs



- Consider the WFSA above, with the tropical ("Viterbi-like") semiring. Take the string *ab*.
- We "multiply" ($\otimes$) the weights along paths; this means adding the costs.
- Two paths for *ab*:
  - One goes through states (0, 1, 1); cost is $(0 + 1 + 2) = 3$
  - One goes through states (0, 2, 3); cost is $(1 + 2 + 2) = 5$
- We add weights across different paths; tropical $\oplus$ is "take min cost" $\rightarrow$ this WFSA maps *ab* to 3

Equivalence of WFSAs



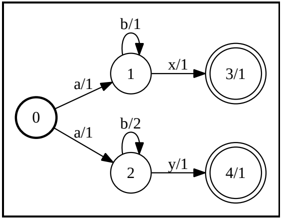


- Two WFSAs are equivalent if they represent the same function from (string) to (weight).
- For example, the above two WFSAs are equivalent (but not equal).
- Easy to test only if we can determinize.
- Otherwise there are randomized algorithms (pick random string from one; make sure it has same weight in both).

Determinization of Weighted Finite State Acceptors

- Determinization is also applicable to WFSAs.
- In algorithm, (subset of states) $\longrightarrow$ (*weighted* subset of states).
- Not all WFSAs are determinizable
- If you try to determinize a nondeterminizable WFSA:
  - There will be an infinite sequence of determinized-states with the same subset of states but different weights
  - Determinization will fail to terminate; eventually memory will be exhausted.

The twins property



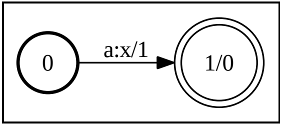
- A WFSA is determinizable if it has the "twins property"...
- (that "if" is "iff" subject to certain extra conditions).
- A WFSA (as above) *fails* to have the twins property[1] if:
  - There are two states s and t...
  - that are reachable from the start state with the same string...
  - and there are cycles at both s and t...
  - with the same string on them but different weights.

[1]We are glossing over some details here

Other algorithms on Weighted Finite State Acceptors

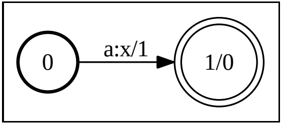
- Shortest-path and shortest-distance algorithms, e.g.:
  - Total weight of FSA, summed over all "successful" paths, i.e. from initial to final.
  - Total weight for all paths to final-state, starting at each state
  - Best path through FSA
  - N-best paths through FSA
- We can also apply most unweighted FSA algorithms to the weighted case.

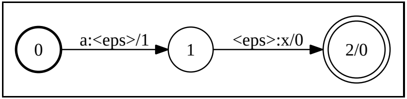
Weighted Finite State Transducers (WFSTs)



- Like a WFSA except with two labels on each arc.
- View it as a function from a (pair of strings) to a weight
- This one maps (a, x) to 1 and all else to ∞
- Note: view 1 and ∞ as costs. ∞ is $\bar{0}$ in semiring.
- Symbols on the left and right are termed "input" and "output" symbols.

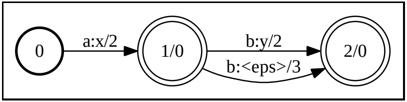
Equivalence of WFSTs





- WFSTs are equivalent if they represent same function from (string, string) to weight.
- Both these WFSTs map (a, x) to 1 and all else to ∞
- Therefore they are equivalent (although not equal)

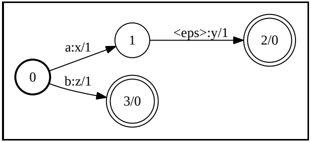
WFSTs versus matrices

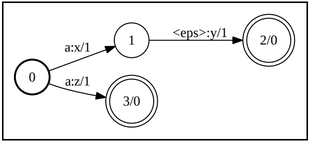


	x	xy
a	2	∞
ab	5	4

- WFSTs can be viewed as function from (input string, output string) to (weight).
- Matrices (over reals) can be viewed as function from (row index, column index) to (real).
- Note: the set of possible input and output strings is infinite.

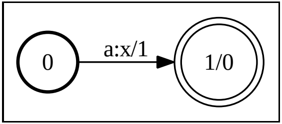
Functional WFSTs

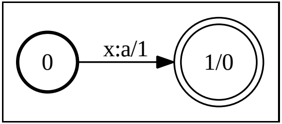




- A WFST is *functional* iff each input string has a non-$\bar{0}$ weight with at most one output string.
- Left transducer is functional: maps a to xy with cost 2, and b to z with cost 1.
- The right transducer is non-functional:
  - The string a has a nonzero weight with two strings.
  - I.e. (a, xy) and (a, z) both appear on successful paths.
- WFST can be interpreted as function from input to (output, weight) only if functional.

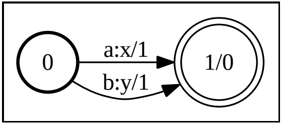
Inversion of WFSTs

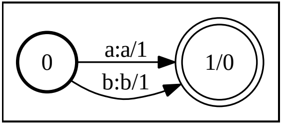




- Inversion just means swapping the input and output symbols.
- If A if WFST, we write its inverse as A^{-1}.
- Only vaguely analogous to function inversion ...
- In matrix analogy, it's transposition.

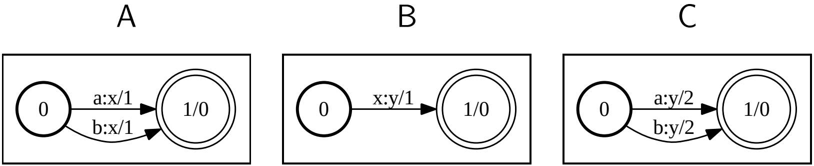
Projection of WFSTs





- Projection on the input means copying input symbols to output, so both are identical.
- Projection on the output means copying the output symbol to input.
- In FST toolkits like OpenFst, if input and output symbols are identical the WFST is termed an acceptor (WFSA) and can be treated as one.

Composition of WFSTs



- Notation: $C = A \circ B$ means, C is A composed with B.
- In special cases, composition is similar to function composition
- Composition algorithm "matches up" the "inner symbols"
  - i.e. those on the output (right) of A and input (left) of B

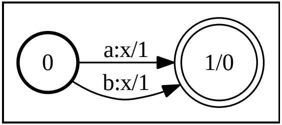
Composition of WFSTs vs matrix multiplication

- Composition of WFSTs corresponds exactly with matrix multiplication.
- Let s, t and u be strings, and w be a weight.
- Interpret A,B,C as functions from (string, string) to (weight).
- Then $C(s,u) = \oplus_t A(s,t) \otimes B(t,u)$ (sum over and discard central string t).
- Matrix multiplication follows the same pattern (sum over central index).

Composition of WFSTs (algorithm)

- Ignoring ϵ symbols, algorithm is quite simple.
- States in C correspond to tuples of (state in A, state in B).
  - But some of these may be inaccessible and pruned away.
- Maintain queue of pairs, initially the single pair $(0, 0)$ (start states).
- When processing a pair (s, t):
  - Consider each pair of (arc a from s), (arc b from t).
  - If these have matching symbols (output of a, input of b):
    - Create transition to state in C corresponding to (next-state of a, next-state of b)
    - If not seen before, add this pair to queue.
- With ϵ involved, need to be careful to avoid redundant paths...

Deterministic WFSTs



- Taken to mean "deterministic on the input symbol"
- I.e., no state can have >1 arc out of it with the same input symbol.
- Some interpretations (e.g. Mohri/AT&T/OpenFst) allow ϵ input symbols (i.e. being ϵ-free is a separate issue).
- I prefer a definition that disallows epsilons, except as necessary to encode a string of output symbols on an arc.
- Regardless of definition, not all WFSTs can be determinized.

Determinizing WFSTs

- Easiest to view the output symbol as part of the weight and determinize as WFSA.
- Encode the WFST as a WFSA in an appropriate semiring that contains the weight and string.
- Do WFSA determinization (and hope it succeeds).
- Two basic failure modes:
  - Fails because input FST was non-functional (multiple outputs for one input)
  - Fails because WFSA fails to have twins property ... i.e. has cycles with same input-symbol seq, different weights.
- Can also fail if there are paths with more output than input symbols and you don't want to introduce ϵ-input arcs (I allow these as a special case).

Speech recognition application of WFSTs

- $HCLG \equiv H \circ C \circ L \circ G$ is the recognition graph
- G is the grammar or LM (an acceptor)
- L is the lexicon
- C adds phonetic context-dependency
- H specifies the HMM structure of context-dependent phones

	Input	Output
H	p.d.f. labels	ctx-dep. phone
C	ctx-dep phone	phone
L	phone	word
G	word	word

Decoding graph construction (simple version)

- Create H, C, L, G separately
- Compose them together
- Determinize (like making a tree-structured lexicon)
- Minimize (like suffix sharing)

Decoding graph construction (complexities)

- Have to do things in a careful order or algorithms "blow up"
- Determinization for WFSTs can fail
  - Need to insert "disambiguation symbols" into the lexicon.
  - Need to "propagate these through" H and C.
- Need to make sure final $HCLG$ is stochastic, for optimal pruning performance.
  - I.e. sums to one, like a properly normalized HMM.
  - Standard algorithm to do this (weight-pushing) can fail
  - ... because FST representation of backoff LMs is non-stochastic
- Also we can't always recover the phone sequence from path through HCLG (but see later).

Decoding graph construction (our approach)

- Mostly followed quite closely the AT&T recipe.
- Added a disambiguation symbol for backoff arcs in the LM FST
  - Partly for reasons of taste; not comfortable with treating ϵ as “real symbol”.
  - It’s actually necessary because we use a determinization algorithm that does ϵ removal.
- Different approach to “weight pushing”– see next slide.
- Put special symbols on input of HCLG that encode more than the p.d.f.– see below.

Our approach to weight pushing

- Want to ensure that HCLG can be interpreted as properly normalized HMM ("stochastic").
- AT&T recipe does this as a final "weight pushing" step.
- This can fail for backoff LMs, or lead to weird results.
- Our approach is to ensure that *if G is stochastic*, each step of graph creation keeps it stochastic.
- This is done in such a way that where G is "nearly" stochastic, $HCLG$ will be "nearly" stochastic.
  - Requires doing some algorithms (e.g. determinization) in log semiring
  - Some algorithms (e.g. epsilon removal), need to "know about" two semirings at once.

Symbols on input of HCLG – the problem

- Normally, the symbols on the input of H (or $HCLG$) represent p.d.f.'s
- The symbol would actually be the integer identifier of the p.d.f., with range e.g. 5k if there are 5k p.d.f.'s.
- The decoder will give you an input-symbol string I, an output-symbol string O, and a weight w.
- We want I (input-symbol seq). to give enough information to
  - train transition models
  - work out the phone sequence
  - reconstruct the path through the HMM.
- We don't want to have to assume different phones have distinct p.d.f.'s.

Symbols on input of HCLG – our solution

- Make the symbols on *HCLG* a slightly a finer-grained identifier.
- We call this a “transition id”.
- From a transition id, we can work out
  - The p.d.f. index
  - The phone
  - The transition arc that was taken within the prototype HMM topology
- There would typically be about twice as many transition-ids as p.d.f. id’s.
- Doesn’t expand the graph size, in normal configurations.

Decoding characteristics

- For a simple, medium-vocab task (Resource Management), about 20 x faster than real time.
- We can decode Wall Street Journal read speech in about real time without significant loss in accuracy.
- This is with a pruned trigram language model– about 0.8M arcs.
- Final HCLG takes about 0.5G (gigabytes) in default OpenFst format.
- Can't go much larger or memory for graph compilation becomes a problem.
- Note: our HCLG currently has self-loops.

Ways to decode with bigger LMs

- Lattice rescoring
  - We can already do this (see lattice generation, below).
- Store HCLG in a more compact form
  - E.g. remove self-loops (decoder would add them)
  - Could then "factor" FST by compactly storing linear seqs of states.
  - Could decrease final graph size 10-fold but graph creation would still require a lot of memory.
- Online "correction" of LM scores in HCLG
  - Idea is to add the difference in LM score between (small, big) LM when we cross a word
  - Already done preliminary work (w/ Gilles Boulianne)
- OpenFst's "fast lookahead matcher"– need to investigate this.

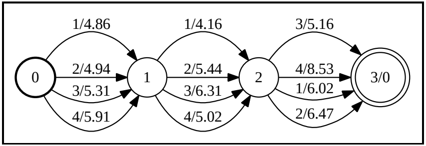
Lattice generation

- From now, will describe how lattice generation in Kaldi works.
- Up to now, we have described standard concepts (with a few minor twists)
- Here starts the "novel" part.
- We will begin with some preliminaries before describing the algorithm.

What is a lattice?

- The word “lattice” is used in the ASR literature as:
  - Some kind of compact representation of the alternate word hypotheses for an utterance.
  - Like an N-best list but with less redundancy.
  - Usually has time information, sometimes state or word alignment information.
  - Generally a directed acyclic graph with only one start node and only one end node.

WFST view of a speech recognition decoder



- First– a "WFST definition" of the decoding problem.
- Let U be an FST that encodes the acoustic scores of an utterance (as above).
- Let $S = U \circ HCLG$ be called the search graph for an utterance.
- Note: if U has N frames (3, above), then
  - #states in S is $\leq (N + 1)$ times #states in $HCLG$.
  - Like $N + 1$ copies of $HCLG$.

Pruning in a speech recognition decoder

- With beam pruning, we search a subgraph of S.
- The set of "active states" on all frames, with arcs linking them as appropriate, is a subgraph of S.
- Let this be called the *beam-pruned subgraph* of S; call it B.
- A standard speech recognition decoder finds the best path through B.
- In our case, the output of the decoder is a linear WFST that consists of this best path.
- This contains the following useful information:
  - The word sequence, as output symbols.
  - The state alignment, as input symbols.
  - The cost of the path, as the total weight.

A lattice – our format

- We let a lattice be an FST, similar to $HCLG$.
- The input symbols will be the p.d.f.'s (actually, transition-ids).
- The output symbols will be words.
- The weights will contain the acoustic costs plus the costs in HCLG
- ... well, actually we use a semiring that contains two floats:
  - Keeps track of (graph, acoustic) costs separately while behaving as if they were added together.
- We also have an "acceptor" form of the lattice. Here,
  - Words appear on both the input and output side.
  - The p.d.f.'s/transition-ids are encoded as part of the weight.
  - This format is more compact, and sometimes more convenient.

Our statement of the lattice generation problem

- Let the lattice beam be α (e.g. $\alpha = 8$).
- No missing paths:
  - Every word sequence whose best path in B is within α of the overall best path, should be present in L
- No extra paths:
  - No path should be in L if there is not an equivalent path in B.
- No duplicate paths:
  - No two paths in L should have the same word sequence.
- Accurate paths:
  - Each path through L should be equivalent to the best path through B with the same word sequence.

Our solution to the lattice generation problem

- During decoding, generate B (beam-pruned subgraph of S).
- Prune B with beam α.
  - I.e. prune all arcs not on path within α of best path.
  - Actually we do this in an online way to avoid memory bloat.
  - This is *not* the same as the beam pruning used in search.
- Let this pruned version of B be called P
- Do a special form of determinization on P (next slide)
- Prune the result with beam α

Lattice determinization

- Special determinization algorithm we call "lattice determinization"
- This is not really a determinization algorithm in a WFST sense, as it does not preserve equivalence.
- It keeps only the lowest-cost output-symbol sequence for each input-symbol sequence
  - Note: we'd apply it after inversion, so words are on input.
- Conceptual view (not how we really do it):
  - Convert WFST to acceptor in a special semiring (see below)
  - Remove epsilons from that acceptor
  - Determinize;
  - Convert back

Converting an FST to an acceptor

- Suppose we had an arc in B with "1520:HI/3.67" on it.
- I.e. input symbol is 1520, output is "HI", cost is 3.67
- First, invert → "HI:1520/3.67"
- Next, convert to acceptor → "HI:HI/(3.67,1520)"
- Remember– acceptors have same input/output symbols on arcs.
- The weight (after "/") is a tuple of (old-weight, string)
- Other valid weights: "(12.20,1520_1520_1629)", "(-2.64,)" (empty string)

Acceptor determinization – conventional semiring

- Multiplication ($\otimes$) is $\otimes$-multiplying the weights, concatenating the strings.
- Addition ($\oplus$) only defined if the string part is the same
  - Just corresponds to $\oplus$-adding the weights in their semiring.
- I.e. addition will crash if strings are different[2]
- ... this is not really a proper semiring
- ... designed to detect non-functional FSTs and to fail then.
- Common-divisor of (w, s) and (x, t) is $(w \oplus x, u)$
  - where u is the longest common prefix of s and t.
  - ... this is an operation needed for determinization (to normalize subsets)

[2] This is the way it's done in OpenFst, anyway.

Semiring for lattice-determinization

- Only defined if there is a total order over the weight part, and $\oplus$ on the weights corresponds to taking the max.
- Multiplication is as before.
- Addition means taking pair that has the largest weight part.
- We need to be a bit careful when the weights are the same.
  - This is for mathematical purity– it wouldn't make a practical difference.
  - It's a question of satisfying semiring axioms.
  - We take the shortest string (if lengths differ), then use lexicographical order.
  - Simple lexicographical order would have violated one of the axioms (distributivity of $\otimes$ over $\oplus$).
- Common-divisor is as before.

Lattice-determinization vs. conventional determinization

- Let's compare them where they are both defined (for tropical semiring, which is like Viterbi).
- When creating a weighted subset of states...
  - Suppose a particular state appears twice in the subset (i.e. we process > 1 links to that state)...
  - If the "weights" on that state have the same string part– both semirings behave the same.
  - Otherwise (different string part):
    - Lattice-determinization will continue happily, choosing the string with better weight.
    - Conventional determinization will say "I detected non-functional FST" and fail.

Lattice-determinization in practice

- The “OpenFst”/AT&T way to do it would be to:
  - Convert to our special semiring; remove epsilons; determinize; convert back.
- We don’t do it this way, because lattices would have a lot of input-epsilons
  - The remove-eps step would “blow up”.
- Instead we wrote a determinization algorithm that removes epsilons itself.
  - This is a bit more complex and harder to formally prove correctness (probably why Mohri doesn’t like it)
  - But easy to verify using random examples (can check equivalence and check if deterministic)
  - Optimized it by using efficient data-structures to store strings.

Lattice generation– simple version

- If we had unlimited memory, basic process would be:
- Generate beam-pruned subgraph B of search graph S
  - The states in B correspond to the active states on particular frames.
- Prune B with beam α to get pruned version P.
- Convert P to acceptor and lattice-determinize to get A (deterministic acceptor)
- Prune A with beam α to get final lattice L (in acceptor form).

Lattice generation– real version

- B would be too large to easily fit in memory– prune as we go, not at the end
  - We have a special algorithm for this that's efficient and exact.
  - I.e. it won't prune anything we wouldn't have pruned away at the end.
- Determinization of P can use a lot of memory for some utterances (if α is large, e.g. ≥ 10)
  - This is caused by links being generated that would have been pruned away later anyway.
  - We could create a determinization algorithm that would avoid this bloat
  - ... but it would probably be quite slow.
  - For now we just detect this, reduce the beam a bit and try again.

Lattice generation– why our algorithm is good

- We can prove that lattices generated in this way will satisfy the properties we stated[3]
- We can get “exact” lattices without approximation (e.g. the “word-pair approximation” of Ney), while only storing a single traceback.
- No extra time cost for generating lattices (for small-ish α)
- This is unlike the traditional “more exact” approaches, which have to store multiple tokens in each state.
- ... and those approaches are not even as exact as ours.
- We have verified that LM rescoring of these lattices gives the same results as decoding directly with the original LM.
- Will also verify using an N-best algorithm that the lattices have the properties we stated.

[3] We have to slightly weaken one of the properties

Lattice generation– experimental aspects

- We have verified that LM rescoring of these lattices gives the same results as decoding directly with the original LM (as beams get large enough)
- Will also verify using an N-best algorithm that the lattices have the properties we stated.
- Also measuring oracle WERs of lattices:
  - This is mainly because people expect it.
  - I don't acknowledge that oracle WER is a relevant metric (consider a single word-loop).
- Not presenting detailed results here (not ready yet/ not that useful anyway).
- Hard to compare experimentally with previous approaches:
  - These are often either not published, not completely described, or too tied to a particular type of decoder.

Lattice generation– summary

- I believe this is the "right" way to do lattice generation.
- No compromise between speed and exactness.
- Lattices have clearly defined properties that can be verified.
- Interesting algorithm:
  - A common objection to WFSTs is, "I could code this easier without WFSTs"
  - Many WFST algorithms have simple analogs, e.g. determinization = tree-structured lexicon; minimization = suffix sharing.
  - This algorithm is hard to explain without WFSTs, and shows the power of the framework.

The end

- Questions?