Composition, Determinization, and Weight Pushing

Dr. John McDonough

Spoken Language Systems
Saarland University

June 22, 2009
Introduction

- Last lecture, we considered how such a search graph could be constructed for use in automatic speech recognition.
- We described the important knowledge sources required to construct such a graph: The grammar $G$, the pronunciation lexicon $L$, and the hidden Markov model $H$.
- In order to combine these knowledge sources and optimize the final graph, three all-import WFST operations are required:
  1. Weighted composition, which is a variant of FSA intersection;
  2. Weighted determinization, which is a variant of FSA power set construction;
  3. Weight pushing, which is defined only for weighted graphs.
- Here, we give detailed descriptions of these algorithms.

Let $A = (Q_A, \Sigma, \delta_A, i_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, i_B, F_B)$ be two FSAs accepting the languages $L_A = L(A)$ and $L_B = L(B)$ respectively. We wish to construct a transducer $C(Q_C, \Sigma, \delta_C, i_C, F_C)$ accepting the language $L_C = L(C) = L_A \cap L_B$.

$C$ can be constructed through the intersection of $A$ and $B$.

For any node $n$ in $Q_A$ or $Q_B$, define the set of symbols

$$\text{symbol}(n) = \{ a | \delta(n, a) \neq \emptyset \}.$$ 

Moreover, define any node $n_C \in Q_C$ as $n_C = [n_A, n_B]$ for $n_A \in Q_A$ and $n_B \in Q_B$.

Let $E[n, a]$ denote the edge leaving $n$ labeled with $a$. 
Pseudocode for FSA Intersection

```python
00 def fsaIntersection((A, B, C):
01     iC ← [iA, iB]
02     QC ← iC
03     FC ← ∅
04     push iC on Q
05     while |Q| > 0:
06         pop nC from Q
07         for a ∈ symbol(nA[nC])
08             and a ∈ symbol(nB[nC]):
09                 qA ← next[E[nA[nC], a]]
10                 qB ← next[E[nB[nC], a]]
11                 qC ← [qA, qB]
12                 δC(nC, a) ← qC
13                 if qC ∉ QC:
14                     QC ← QC ∪ {qC}
15                     push qC on Q
16                 if qA ∈ FA and qB ∈ FB:
17                     FC ← FC ∪ {qC}
```
Consider a transducer $S$ which maps an input string $u$ to an output string $v$ with a weight of $w_1$, and a transducer $T$ which maps input string $v$ to output string $y$ with weight $w_2$.

The composition

$$R = S \circ T$$

of $S$ and $T$ maps string $u$ directly to $y$ with weight

$$w = w_1 \otimes w_2.$$

We will adopt the convention that the components of particular transducer are denoted by subscripts; e.g., $Q_R$ denotes the set of states of the transducer $R$.

In the absence then of $\epsilon$–transitions, the construction of such a transducer $R$ is straightforward.

It entails pairing the output symbols on the transitions of a node $n_S \in Q_S$ with the input symbols on the transitions of a node $n_T \in Q_T$. 
The transition from State 0 labeled with a:b/0.5 in $S$ has been paired with the transition from State 0 labeled with b:c/0.2 in $T$, resulting in the transition labeled a:c/0.7 in $R$.

After each successful pairing, the new node $n_R = (n_S, n_T)$ is placed on a queue to eventually have its adjacency list expanded.
Composition is not Local

- The pairing of the transitions of $n_S$ with those of $n_T$ is *local*, inasmuch as it only entails the consideration of the adjacency lists of two nodes at a time.
- This fact provides for the so-called lazy implementation of weighted composition.
- As $R$ is constructed it can so happen that nodes are created that do not lie on a successful path; i.e., from such a node, there is no path to an end state.
- Such nodes are typically removed or *purged* from the graph as a final step.
- It is worth noting, however, that this purge step is *not* a local operation as it is necessary to consider the entire transducer $R$ in order to determine if any given node is on a successful path.
When $\epsilon$-symbols are introduced, composition becomes more complicated, as it is necessary to specify when an $\epsilon$-symbol on the output of a transition in $n_S$ can be combined with an $\epsilon$-symbol on the input of $n_T$.

In order to avoid the creation of redundant paths through $R$, it is necessary to replace the composition $S \circ T$ with $S \circ V \circ T$, where $V$ is a filter.

Hence, a node $n_R \in Q_R$ is specified by a triple $(n_S, n_T, f)$, where $f \in \{0, 1, 2\}$ is an index indicating the state of $V$.

In effect, the filter specifies that after a lone $\epsilon$-transition on either the input or output side is taken, placing the filter in State 1 or State 2 respectively, an $\epsilon$-transition on the other side may not be taken until a non-$\epsilon$ match between input and output occurs, thereby returning the filter to State 0 over one of the edges labeled with $x:x$. 
Figure: Filter used during composition with $\epsilon$—symbols.
We now present the first of a series of equivalence transformations.

We begin with a pair of definitions.

*Equivalent:* Two WFSAs are *equivalent* if for any accepted input sequence, they produce the same weight. Two WFSTs are equivalent if for any accepted input sequence they produce the same output sequence and the same weight.

*Deterministic:* A WFST is *deterministic* if at most one transition from any node is labeled with any given input symbol.
Let $M_1 = (Q_1, \Sigma, \delta_1, i_1, F_1)$ denote a NFSA accepting language $L$.

Define a DFSA $M_2 = (Q_2, \Sigma, \delta_2, i_2, F_2)$ as follows:

- The states of $M_2$ are all subsets of the states of $M_1$, that is $Q_2 = 2^{Q_1}$.
- $M_2$ keeps track in its states the subset of states that $M_1$ could in at any time.
- $F_2$ is the subset of states in $Q_2$ which contain a state $f \in F_1$.
- An element of $m \in Q_2$ will be denoted as $m = [m_1, m_2, \ldots, m_N]$, where each $m_n \in Q_1$.
- Finally, $i_2 = [i_1]$. 
Definition of $\delta_2([m_1, m_2, \ldots, m_N], a)$

- By definition,
  
  $$\delta_2([m_1, m_2, \ldots, m_N], a) = [p_1, p_2, \ldots, p_N]$$

  if and only if

  $$\delta_1(\{m_1, m_2, \ldots, m_N\}, a) = \{p_1, p_2, \ldots, p_N\}$$

- In other words, $\delta_2([m_1, m_2, \ldots, m_N], a)$ is computed for $[m_1, m_2, \ldots, m_N] \in Q_2$ by applying $\delta_1$ to each $m_n \in Q_1$. 
Implementing the Power Set Construction

- The power set $2^Q$ of $Q$ contains $2^{|Q|}$ subsets.
- This implies that the power set construction requires exponential running time in the worst case; i.e., it is intractable.
- Fortunately, for the FSAs used for speech recognition and many other applications, the vast majority of subsets in the power set are never constructed.
- The key to successfully implementing the power set construction is to *not* construct a priori all subsets in the power set.
- Rather, only those subsets are constructed which are actually required.
- This subset is comprised of those subsets which are *accessible* or *reachable* from the initial node.
Pseudocode for Power Set Construction

```python
00 def powerSetConstruction(τ1, τ2):
01     F_2 ← ∅
02     i_2 ← i_1
03     Q ← { i_2 }
04     push i_2 on Q
05     while |Q| > 0:
06         pop q_2 from Q
07         if ∃ q ∈ q_2 such that q ∈ F_1:
08             F_2 ← F_2 ∪ {q_2}
09             for a such that δ_2(q_2, a) ≠ ∅:
10                 if δ_2(q_2, a) ∉ Q_2:
11                     Q_2 ← Q_2 ∪ { δ_2(q_2, a) }
12                     push δ_2(q_2, a) on Q
```

**Figure:** Pseudocode for power set construction.
Diagram of Power Set Construction
Thus we are led to consider our first equivalence operation, \textit{determinization}, which produces a deterministic transducer $\tau_2$ that is equivalent to some given transducer $\tau_1$. 

\textbf{String-to-weight subsequential transducer:} A \textit{string-to-weight subsequential transducer} on the semiring $\mathbb{K}$ is an 8–tuple $\tau = (\Sigma, Q, i, F, \delta, \sigma, \lambda, \rho)$ consisting of:

- of an \textit{input alphabet} $\Sigma$,
- a set of states $Q$,
- an initial state $i \in Q$ with weight $\lambda \in \mathbb{R}^*$,
- a set of final states $F \subseteq Q$,
- a transition function $\delta$ mapping $Q \times \Sigma$ to $Q$,
- a output function $\sigma$ mapping $Q \times \Sigma$ to $\mathbb{R}^+$,
- and a final weight function $\rho$ mapping from $F$ to $\mathbb{R}^+$.

As mention previously, weighted determinization is similar to classical power set construction.
The states in the determinized transducer correspond to *subsets* of states in the original transducer, together with a residual weight.

The initial state $i_2$ in $\tau_2$ corresponds only to the state $i_1$ of $\tau_1$.

The subset of states together with their residual weights that can be reached from $i_1$ through a transition with the input label $a$ then form a state in $\tau_2$. 
As there may be several transitions with input label $a$ having different weights, the output of the transition from $i_2$ labeled with $a$ can only have the minimum weight of all transitions from $i_1$ labeled with $a$.

The residual weight above this minimum must then be carried along in the definition of the subset to be applied later.

Each time a new state in $\tau_2$, consisting of a subset of the states of $\tau_1$ together with their residual weights, is defined, it is added to a queue $Q$, so that it will eventually have its adjacency list expanded.

When the adjacency lists of all states in $\tau_2$ have been expanded and $Q$ has been depleted, the algorithm terminates.
In order to clearly describe such an algorithm, let us define the following sets:

- \( \Gamma(q_2, a) = \{(q, x) \in q_2 : \exists e = (q, a, \sigma[e], n_1[e]) \in E_1 \} \) denotes the set of pairs \((q, x)\) which are elements of \(q_2\) where \(q\) has at least one edge labeled with \(a\);
- \( \gamma(q_2, a) = \{(q, x, e) \in q_2 \times E_1 : e = (q, a, \sigma_1[e], n_1[e]) \in E_1 \} \) denotes the set of triples \((q, x, e)\) where \((q, x)\) is a pair in \(q_2\) such that \(q\) admits a transition with input label \(a\);
- \( \nu(q_2, a) = \{q' \in Q_1 : \exists (q, x) \in q_2, \exists e = (q, a, \sigma_1[e], q') \in E_1 \} \) is the set of states \(q'\) in \(Q_1\) that can be reached by transitions labeled with \(a\) from the states of subset \(q_2\).
Pseudocode for the complete algorithm is provided in the listing below.

```python
00 def determinize(τ₁, τ₂):
01     F₂ ← ∅
02     i₂ ← i₁
03     λ₂ ← λ₁
04     Q ← { i₂ }
05     while |Q| > 0:
06         pop q₂ from Q
07         if ∃ (q, x) ∈ q₂ such that q ∈ F₁:
08             F₂ ← F₂ ∪ {q₂}
09             ρ₂(q₂) ← ⊕ x ⊗ ρ₁(q)
10                q∈F₁,(q,x)∈ q₂
11         for a such that Γ(q₂, a) ≠ ∅:
12             σ₂(q₂, a) ← ⊕(q,x)∈Γ(q₂,a) [ x ⊗ ⊕ e=(q,a,σ₁[e],n₁[e])∈ E₁ σ₁[e]]
13             δ₂(q₂, a) ← ∪ { ( ḗ, ⊕ [σ₂(q₂, a)]⁻¹ ⊗ x ⊗ σ₁[e] ) } ḗ∈ ν(q₂,a)
14                (q,x,t) ∈ γ(q₂,a),n₁[e]=ḡ
15         if δ₂(q₂, a) ∉ Q₂:
16             Q₂ ← Q₂ ∪ { δ₂(q₂, a) }
17             push δ₂(q₂, a) on Q
```

Details of Weighted Determinization

- The weighted determinization algorithm is perhaps most easily understood by specializing all operations for the tropical semiring.
- This implies $\oplus$ is replaced by min and $\odot$ is replaced by $+$. 
- The algorithm begins by initializing the set $F_2$ of final states of $\tau_2$ to $\emptyset$ in Line 01, and equating the initial state and weight $i_2$ and $\lambda_2$ respectively to their counterparts in $\tau_1$ in Lines 02–03.
Details (cont’d.)

- The initial state $i_2$ is then pushed onto the queue $Q$ in Line 04.
- In Line 05, the next subset $q_2$ to have its adjacency list expanded is popped from $Q$.
- If $q_2$ contains one or more pairs $(q, x)$ comprised of a state $q \in Q_1$ and residual weight $x$ whereby $q \in F_1$, then $q_2$ is added to the set of final states $F_2$ in Line 08 and assigned a final weight $\rho_2(q_2)$ equivalent to the minimum of all $x \odot \rho_1(q)$ where $(q, x) \in q_2$ and $q \in F_1$ in Line 09.
The next step is to begin expanding the adjacency list of $q_2$ in Line 10, which specifies that the input symbols on the edges of the adjacency list of $q_2$ is obtained from the union of the input symbols on the adjacency lists of all $q$ such that there exists $(q, x) \in q_2$.

In Line 11, the weight assigned the edge labeled with $a$ on the adjacency list of $q_2$ is obtained by considering each $(q, x) \in \Gamma(q_2, a)$ and finding the edge with the minimum weight on the adjacency list of $q$ that is labeled with $a$ and multiplying this minimum weight with the residual weight $x$.

Thereafter, the minimum of all the weights $x$ is taken for all pairs $(q, x)$ in $\Gamma(q_2, a)$. 
In Line 12, the identity of the new subset of \((q, x) \in Q_2\) is determined and assigned to \(\delta(q_2, a)\).

If this new subset is previously unseen, it is added to the set \(Q_2\) of states of \(\tau_2\) in Line 14 and pushed onto the queue \(Q\) in Line 15 to have its adjacency list expanded in due course.
Effects of Weighted Determinization

- A simple example of weighted determinization is shown in the figure.
- The two WFSTs in the figure are equivalent over the tropical semiring in that they both accept the same input strings, and for any given input string, produce the same output string and the same weight.
- For example, the original transducer will accept the input string \textit{aba} along either of two successful paths, namely, using the state sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 3$ or the state sequence $0 \rightarrow 1 \rightarrow 4 \rightarrow 3$. 
Effects (cont’d.)

Both sequences produce the string $ab$ as output, but the former yields a weight of $0.1 + 0.4 + 0.6 = 1.1$, while the latter assigns a weight of $0.1 + 0.3 + 0.5 = 0.9$.

Hence, given that these WFSTs are defined over the tropical semiring, the final weight assigned to the input $aba$ is 0.9, the minimum of the weights along the two successful paths.
The second transducer also accepts the input string *aba*.
There is but a single sequence labeled with this input, namely, 0 → 1 → 4 → 5, which produces a weight of $0.1 + 0.3 + 0.5 = 0.9$. 
Search in ASR

- For any given transducer, there are many equivalent transducers that differ only in the distribution of weights along their edges.
- An ASR system typically uses a beam search to find the most likely word sequence.
- The efficiency of the beam search depends very strongly on eliminating unlikely hypotheses as early as possible from the beam.
- This implies that the weights should be *pushed* as far toward the initial node as possible to achieve the most efficient search.
- Here we discuss an algorithm for achieving this optimal distribution of weights; see the figure.
Diagram of Weight Pushing

Before Weight Pushing

After Weight Pushing

Figure: Weight pushing over the tropical semiring for a simple transducer.
Potential Function

The weight pushing algorithm proposed begins with the definition of a potential function $V : Q \rightarrow \mathbb{K} - \{\bar{0}\}$.

The weights of the transducer are then reassigned according to

$$
\lambda \leftarrow \lambda \otimes V(i),
$$

$$
\forall e \in E, w[e] \leftarrow [V(p[e])]^{-1} \otimes (w[e] \otimes V(n[e])),
$$

$$
\forall f \in F, \rho(f) \leftarrow [V(f)]^{-1} \otimes \rho[f].
$$

This reassignment has no effect on the weight assigned to any accepted string, as each weight from $V$ is added and subtracted once.
For optimal weight pushing, we assign a potential to a state $q$ to be equal to the weight of the shortest path from $q$ to the set of final states $F$, such that

$$V(q) = \bigoplus_{\pi \in P(q)} w[\pi],$$

where $P(q)$ denotes the set of all paths from $q$ to $F$.

The general all pairs shortest path algorithm is too inefficient for weight pushing on very large transducers.

Instead an *approximate* shortest path algorithm is used.
Psuedocode for Calculating the Potential Function

```python
def shortestDistance():
    for j in 1 to |Q|:
        d[j] ← r[j] ← 0
    Q ← {i}
    while |Q| > 0:
        pop q from Q
        R ← r[q]
        r[q] ← 0
        for e ∈ E[q]:
            if d[n[e]] ≠ d[n[e]] ⊕ (R ⊗ w[e]):
                d[n[e]] ← d[n[e]] ⊕ (R ⊗ w[e])
                r[n[e]] ← r[n[e]] ⊕ (R ⊗ w[e])
                if n[e] ∉ Q:
                    push n[e] on Q
        d[i] ← 1
```
Psuedocode (cont’d.)

- The algorithm functions by first assigning all states $q$ a potential of $\bar{0}$ in Lines 01–02, and placing the initial state $i$ on a queue $Q$ of states that are to be relaxed in Line 03.
- For each node $q$, the current potential $d[q]$ as well as the amount of weight $r[q]$ that has been added since the last relaxation step are maintained.
- When $q$ is popped from $Q$, all nodes $n[e]$ that can be reached from the adjacency list $E[q]$ are tested in Line 09 to determine whether they should be relaxed.
Psuedocode (cont’d.)

- The relaxation itself occurs in Lines 10 and 11. Thereafter the relaxed node \( n[e] \) is placed on \( Q \) if not already there in Lines 12 and 13.

- The algorithm terminates when \( Q \) is depleted. The approximation in this algorithm involves the test in Line 09, which, strictly speaking, must always be true implying, that the algorithm will never terminate.

- In practice, however, a small threshold on the deviation from equality can be set so that the algorithm terminates after a finite number of relaxations.
Before calculating the potential of each node, it is necessary to first reverse the graph.

This implies that for every edge \( e = (p, l_i, l_o, w, n) \) in the original graph \( R \) there will be an edge \( e_{\text{reverse}} = (n, l_i, l_o, w, p) \) in \( R_{\text{reverse}} \).

More formally, given a graph \( G = (V, E) \) with weight function \( w : E \rightarrow \mathbb{R} \), and a set of final states \( F \subset V \), consider a directed, weighted graph \( G' = (V', E') \) with initial state \( i \), and

\[
\begin{align*}
V' &\triangleq V \cup \{i\}, \\
F' &\triangleq \{s\}, \\
E' &\triangleq \{v \rightarrow u : u, v \in V \text{ and } u \rightarrow v \in E\} \cup \{i \rightarrow f : f \in F\}.
\end{align*}
\]
As coarticulation effects are prevalent in all human speech, a phone must be modeled in its left and right context to achieve optimal recognition performance.

A triphone model uses one phone to the left and one to the right as the context of a given phone. Similarly, a pentaphone model models considers two phones to the left and two to the right; a septaphone model models considers three phones to the left and three to the right.

Using even a triphone model, however, requires the contexts to be clustered.

This follows from the fact that if 45 phones are needed to phonetically transcribe all the words of a language, and if the HMM representing each context has three states, then there will be a total of $3 \times 45^3 = 273,375$ GMMs in the complete AM, all of which need to be trained.
Such training could not be robustly accomplished with any reasonable amount of training data.

Moreover, many of these contexts will never occur in any given training set for two reasons:

1. It is common to use different pronunciation lexicons during training and test, primarily because the vocabularies required to cover the training and test sets are often different.

2. State-of-the-art ASR systems typically use *crossword* contexts to model coarticulation effects between words.

From the latter point it is clear that even if the training and test vocabularies are exactly the same, new contexts can be introduced during test if the same words appear in a *different order*.
Figure: A decision tree for modeling context dependency
A popular solution to these problems is to use triphone, pentaphone, or even septaphone contexts, but to use such context together with context or state clustering.

With this technique, sets of contexts are grouped or clustered together, and all contexts in a given cluster share the same GMM parameters.

The relevant context clusters are most often chosen with a decision tree such as that depicted in the next figure.

As shown in the figure, each node in the decision tree is associated with a question about the phonetic context.
The question “Left-Nasal” at the root node of the tree is to be interpreted as, “Is the left phone a nasal?”

Those phonetic contexts for which this is true are sent to the left, and those for which it is false to the right.

This process of posing questions and partitioning contexts based on the answer continues until a leaf node is reached, whereupon all contexts clustered to a given leaf are assigned the same set of GMM parameters.
State Clustering (cont’d.)

- Usually the clusters are estimated with the same training set as that used for HMM parameter estimation.
- In addition to ensuring that each state cluster has sufficient training data for reliable parameter estimation, this decision tree technique provides a convenient way of assigning contexts not seen in the training data to an appropriate cluster.
- In order to model coarticulation effects during training and test, the context-independent transducer $H$ is replaced with the context-dependent transducer $HC$ shown in the following figure.
- The edges of $HC$ are labeled on the input side with the GMM names (e.g., “AH-b(82)”, “AH-m(32)”, and “AH-e(43)”) associated with the leaf nodes of a decision tree, such as that depicted in the figure.
Context-Dependency Transducer $HC$
In this next lecture, we considered how such a search graph can be constructed using the theory of weighted finite-state transducers.

This yields a very efficient, but can be costly in terms of main memory (RAM).

Recently, hybrid decoding techniques have been developed that yield nearly the efficiency of WFST search without the tremendous RAM requirements.