Single-Source Shortest Paths Algorithms

John McDonough

Language Technologies Institute,
Machine Learning for Signal Processing Group,
Carnegie Mellon University

March 28, 2012

Introduction

- In this lecture, we discuss algorithms for determing the *shortest* path through a weighted graph.
- We will learn that the algorithms for solving such problems are somewhat more complex than the BFS and DFS discussed in prior lectures.
- Such problems are still tractable, however, and, for a graph $G = (V, E)$, can be solved in $\mathcal{O}(V + E)$ time, provided there are no negative weights.
- An important technique for solving such problems is that of *relaxation*.
  Coverage: Cormen, Leiserson, and Rivest (1990), Chapter 24.

Definition: Shortest Path

- Consider a weighted, directed graph $G = (V, E)$ with a set of nodes or vertices V, and a set of edges E.
- There is also a weight function $w : E \to \mathbf{R}$ mapping edges to real-valued weights.
- The *weight* of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of the constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

- The *shortest-path weight* from u to v is defined as

$$\delta(u, v) \triangleq \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{if there is a path from } u \text{ to } v, \\ \infty, & \text{otherwise.} \end{cases}$$

- The *shortest-path* from vertex u to vertex v is then defined as any path with weight $w(p) = \delta(u, v)$.

Optimal Substructure of a Shortest Path

- Algorithms for determining the shortest path through a graph typically exploit the fact that a given shortest path must contain other shortest paths within it.
- This optimality is characterized more precisely in the following lemma.

Lemma: (Subpaths of shortest paths are shortest paths) Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbf{R}$, let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k, and for any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof of Optimal Substructure of a Shortest Path

- Decompose path p as $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, such that $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.
- Now assume that there exists a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$.
- Then $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from v_1 to v_k whose weight $w(p) = w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$.
- This contradicts the assumption that p is the shortest path from v_1 to v_k.

Cycles

- The graphs described in this lecture have real-valued weights on their edges.
- The shortest path between v_0 and v_k in a graph with only positive weights cannot contain any cycles.
- Let $p = \langle v_0, v_2, \ldots, v_k \rangle$ denote the shortest path between v_0 and v_k.
- Let $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ denote a cycle with positive weights such that $v_i = v_j$ and $w(c) > 0$.
- This implies the path $p' = \langle v_0, v_2, \ldots, v_i, v_{j+1}, \ldots, v_k \rangle$ has weight $w(p') = w(p) - w(c) < w(p)$, which contradicts the assumption that p is the shortest path from v_0 to v_k.

Representing Shortest Paths

- The representation for shortest paths is similar to that previously used for BFS trees.
- For a graph $G = (V, E)$, we store for each vertex $v \in V$ a *predecessor* $\pi[v]$ which is either another vertex or `NULL`.
- Given a vertex for which $\pi[v] \neq$ `NULL`, the procedure `Print-Path(`G, s, v`)` can be used to print the shortest path from s to v.
- Question: Is the path correctly printed?

```
00    def Print-Path(G, s, v):
01        print(v)
02        pred = π[v]
03        while not pred == NULL:
04            print(pred)
05            pred = π[pred]
```

Object of a Shortest Path Algorithm

- Upon termination, a shortest path algorithm will have set the predecessor $\pi[v]$ for each $v \in V$ such that it points towards the prior vertex on the shortest path from s to v.
- Note that $\pi[v]$ will *not* necessarily point to the predecessor of v on the shortest path from s to v while the algorithm is still running.
- Let us define the *predecessor subgraph* $G_\pi(V_\pi, E_\pi)$ as that graph induced by the back pointers π of each vertex.
- Let us define the set $V_\pi \triangleq \{v \in V : \pi[v] \neq \texttt{NULL}\} \cup \{s\}$.
- The directed edge set E_π is the set of edges induced by the π values for vertices in V_π:

$$E_\pi = \{(\pi[v], v) \in E : v \in V - \{s\}\}.$$

Shortest-Paths Tree

- A *shortest-paths tree* rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$ such that
  - V' is the set of vertices reachable from $s \in G$,
  - G' forms a rooted tree with root s, and
  - for all $v \in V$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Initialization

- During the execution of a shortest-paths algorithm, we maintain for each $v \in V$ an attribute $d[v]$ which is the current estimate of the shortest path distance.
- The attributes $\pi[v]$ and $d[v]$ are initialized as in the algorithm shown below.
- After initialization, $\pi[v] = \texttt{NUL}$ for all $v \in V$, $d[s] = 0$ and $d[v] = \infty$ for $v \in V - \{s\}$.

```
00    def Initialize-Single-Source(G, s):
01        for v ∈ G:
02            d[v] ← ∞
03            π[v] ← NULL
04        d[s] ← 0
```

Relaxation

- The process of *relaxing* an edge $u \rightarrow v$ means testing whether the distance from s to v can be reduced by traveling over u.
- This process is illustrated in the pseudocode given below.
- The relaxation procedure may decrease the value of the shortest path estimate $d[v]$ and update $\pi[v]$.
- The estimate $d[v]$ can never increase during relaxation, only remain the same or decrease.

```
00    def Relax(u, v, w):
01        if d[v] > d[u] + w(u, v):
02            d[v] ← d[u] + w(u, v)
03            π[v] ← u
```

Properties of Shortest Paths and Relaxation

1. **Triangle inequality:** For any edge $u \to v \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
2. **Upper bound property:** It holds that $d[v] \geq \delta(s, v)$ for all $v \in V$, and once $d[v] = \delta(s, v)$, the value of $d[v]$ is never again altered.
3. **No-path property:** If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$.
4. **Convergence property:** If $s \rightsquigarrow u \to v$ is the shortest path in G for some $u, v \in V$, and if $d[u] = \delta(s, u)$ at any time prior to relaxing $u \to v$, then $d[v] = \delta(s, v)$ at all times afterward.
5. **Path relaxation propery:** If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in the order $v_0 \to v_1, v_1 \to v_2, \ldots, v_{k-1} \to v_k \rangle$, then $d[v_k] = \delta(s, v_k)$.
6. **Predecessor-subgraph property:** Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Single-Source Shortest Paths in dags

1. Shortest distances are always well defined in dags (directed acylic graphs), as no negative weight cycles can exist even if there are negative weights on some edges.
2. For a dag $G = (V, E)$, the shortest paths to all nodes can be found in $\mathcal{O}(V + E)$ time.
3. First the vertices must be topologically sorted.
4. Thereafter the edges from each node can be relaxed, where the vertices are taken in topological order.

```
00    def DAG-Shortest-Paths(G, w, s):
01        sorted = Topo-Sort(G)
02        Initiliaze-Single-Source(G, s)
03        for u in sorted:
04            for v in Adj[u]:
05                Relax(u, v, w)
```

Run Time Analysis

- The topological sort of G can be performed in $\mathcal{O}(V + E)$ time.
- Thereafter, every vertex must be iterated over in the `for` loop of Line 03.
- The edges in the adjacency list of each vertex v are examined exactly once.
- Hence, the total time spent on the inner `for` loop of Lines 04-05 is $\mathcal{O}(V + E)$.

Dijkstra's Algorithm

- *Dijkstra's algorithm* solves the single-source shortest paths algorithm on a weighted, directed graph $G = (V, E)$, provided that $w(u, v) \geq 0$ for each edge $u \rightarrow v \in E$.
- The set S contains vertices whose shortest path distances have already been determined, and Q is a priority queue.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest path estimate, whose edges are then relaxed.

Pseudocode for Dijkstra's Algorithm

```
00    def Dijkstra(G, w, s):
01        Initialize-Single-Source(G, s)
02        S ← ∅
03        Q ← V[G]
04        while Q ≠ ∅:
05            u ← Extract-Min(Q)
06            S ← S ∪ {u}
07            for v ∈ Adj[u]:
08                Relax(u, v, w)
```

Correctness of Dijkstra's Algorithm

Theorem (correctness of Dijkstra's algorithm): Dijkstra's algorithm, when run on a weighted, directed graph $G = (V, E)$ with a non-negative weight function $w : e \in E \rightarrow \mathbf{R}$ and source s, terminates with $d[u] = \delta(s, u)$ for all $v \in V$.

Proof: We use the following loop invariant:

At the start of each iteration of the `while` loop of Lines 04–08, $d[v] = \delta(s, v)$ for each vertex $s \in S$.

Proof of Correctness of Dijkstra's Algorithm

- Assume that $u \in V$ is the first vertex added to S such that $d[u] \neq \delta(s, u)$.
- Let us examine the situation of the `while` loop when u is added to S.
- Prior to adding u to S, there is a path p connected a vertex in S, namely S, to a vertex in $V - S$, namely u.
- Let y be the first vertex on this path such that $y \in V - S$, and let x be the predecessor of y.
- The existence of such a $y \neq u$ implies $d[u] \leq d[y]$, as otherwise y would have been chosen for insertion into S ahead of u.

Proof (cont'd.)

- As shown in the figure, the path p can be decomposed as $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$.
- Either subpath p_1 or p_2 can have no edges.
- Firstly, $d[y] = \delta(s, y)$ when u is added to S.
- This follows from the fact that $d[x] = \delta(s, x)$ when u is added to S.
- As the edge $x \rightsquigarrow y$ was relaxed at the time that x was added to S, the claim follows from the convergence property.

Proof (cont'd.)

- Because y occurs before u on the shortest path from s to u, and all edges have nonnegative weights, we have $\delta(s, y) \leq \delta(s, u)$ and hence

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]. \tag{1}$$

- But both y and u were in $V - S$ when u was chosen for insertion in S, hence $d[u] \leq d[y]$.
- Hence, both inequalities in (1) are actually equalities, such that

$$d[y] = \delta(s, y) = \delta(s, u) = d[u]. \tag{2}$$

- Therefore, $d[u] = \delta(s, u)$, which contradicts our choice of u.
- We conclude, $d[u] = \delta(s, u)$ when u is inserted in S, and this equality was maintained at all later times.

Bellmann-Ford Algorithm

- The *Bellmann-Ford algorithm* determines the shortest path from the source s to each $v \in V$ for a graph $G = (V, E)$ with real-valued weights, which may be negative.
- The algorithm assigns each vertex $v \in V$ its correct shortest path weight, provided there are no cycles with negative weights.
- The algorithm then returns true iff there are no negative weight cycles.

```
00    def Bellmann-Ford(G, w, s):
01        for i ← 1 to |V[G]| − 1:
02            for u → v ∈ E[G]:
03                Relax(u, v, w)
04        for u → v ∈ E[G]:
05            if d[v] > d[u] + w(u, v):
06                return False
07        return True
```

Correctness of the Bellmann-Ford Algorithm

Lemma (correctness of the Bellmann-Ford algorithm): Let $G = (V, E)$ be a weighted, directed graph with a source s and weight function $w : E \rightarrow \mathbf{R}$, and assume that G contains no cycles with negative weights that are reachable from s. Then, after the $|V| - 1$ iterations of the `for` loop in Lines 01–03, it must hold that $d[v] = \delta(s, v)$ for all vertices $v \in V$ that are reachable from s.

Proof of Correctness of Bellmann-Ford Algorithm

- Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$, and $v_k = v$, be an acyclic shortest path from s to v.
- Path p has at most $|V| - 1$ edges.
- Each of the iterations of the `for` loop in Lines 01–03 relaxes all edges $e \in E$.
- Among the edges relaxed in the i-th iteration for all $i = 1, 2, \ldots, k$ is $v_{i-1} \rightarrow v_i$.
- Therefore, by the path-relaxation property it follows

$$d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v).$$

Corollary

Corollary: Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \rightarrow \mathbf{R}$. Then for each vertex $v \in V$, there is a path from s to v iff `Bellmann-Ford` terminates with $d[v] < \infty$ when it is run on G.

Examples of Semirings: Tropical Semiring

- In ASR we typically use one of two semirings, depending on the operation.
- The *tropical semiring* $(\mathbb{R}^+, \min, +, 0, 1)$, where $\mathbb{R}^+$ denotes the set of non-negative real numbers, is useful for finding the shortest path through a search graph.
- The set $\mathbb{R}^+$ is used in the tropical semiring because the hypothesis scores represent negative log-likelihoods.
- The two operations on weights correspond to the multiplication of two probabilities, which is equivalent to addition in the negative log-likelihood domain, and discarding all but the lowest weight, such as is done by the Viterbi algorithm.

Examples: Log-Probability Semiring

- The *log-probability semiring* $(\mathbb{R}^+, \oplus_{\log}, +, 0, 1)$ differs from the tropical semiring only inasmuch as the min operation has been replaced with the *log-add operation* $\oplus_{\log}$, which is defined as

$$a \oplus_{\log} b \triangleq -\log(e^{-a} + e^{-b}).$$

- The log-probability semiring is typically used for the weight pushing equivalence transformation discussed later.

Examples: String Semiring

- In addition to the tropical and log-probability semiring which clearly operate on real numbers, it is also possible to define the *string semiring* wherein the weights are in fact strings, and the operation $\oplus = \wedge$ corresponds to taking the longest common substring, while $\odot = \cdot$ corresponds to concatenation of two strings.
- Hence, the string semiring can be expressed as $K_{\text{string}} = (\Sigma^* \cup \infty, \wedge, \cdot, \infty, \epsilon)$.
- The string semiring will prove useful for weighted determinization.

Weighted Finite-State Acceptors

- We now define our first automaton, the *weighted finite-state acceptor:* A *weighted finite-state acceptor* (WFSA) $A = (\Sigma, Q, E, i, F, \lambda, \rho)$ on the semiring $K = (\Sigma, \oplus, \otimes, \bar{0}, \bar{1})$ consists of
  - an *alphabet* Σ,
  - a finite set of states Q,
  - a finite set of *transitions* $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Sigma \times Q$,
  - a *initial state* $i \in Q$ with weight λ,
  - a set of *end states* $F \subseteq Q$,
  - and a function ρ mapping from F to $\mathbb{R}^+$.

  A *transition* or *edge* $e = (p[e], l[e], w[e], n[e]) \in E$ consists of
  - a previous state $p[e]$,
  - a next state $n[e]$,
  - a label $l[e] \in \Sigma$, and
  - a weight $w[e] \in \Sigma$.

  A final state $n \in F$ may have an associated weight $\rho(n)$.

Diagram of Weighted Finite-State Acceptor

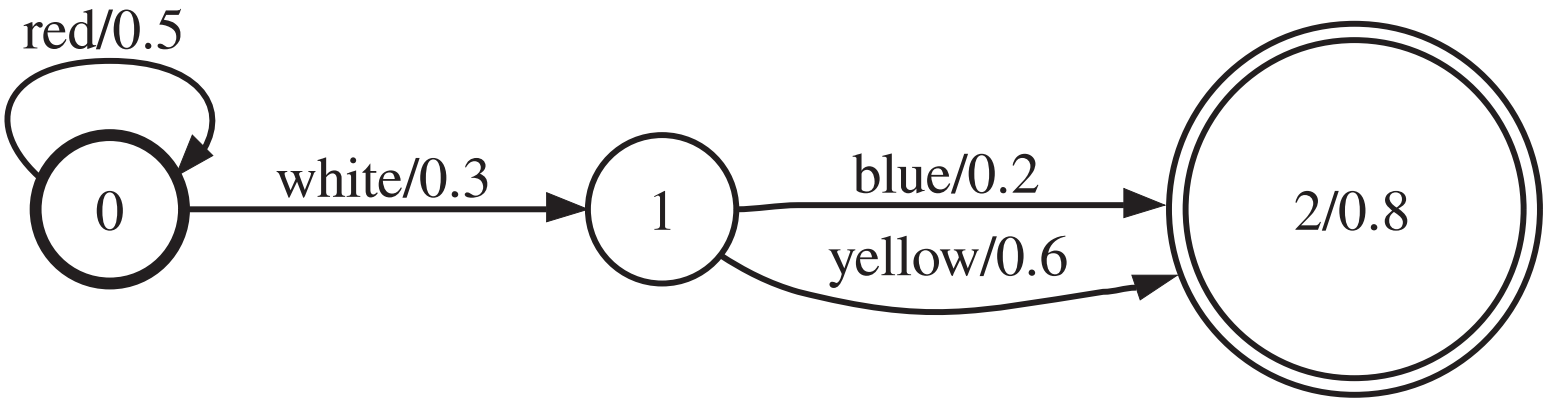


Figure: A simple weighted finite-state acceptor.

- A simple WFSA is shown in Figure 1.
- This acceptor would assign the input string “red white blue” a weight of $0.5 + 0.3 + 0.2 + 0.8 = 1.8$.

Successful Path

- As already explained, speech recognition will be posed as the problem of finding the shortest path through a WFSA, where the length of a path will be determined by a combined AM and LM score.
- Hence, we will require a formal definition of a *path*:
  A path π through an acceptor A is a sequence of transitions $e_1 \cdots e_K$, such that

$$n[e_k] = p[e_{k+1}] \;\; \forall \, k = 1, \ldots, K-1.$$

- A *successful path* $\pi = e_1 \cdots e_K$ is a path from the initial state i to an end state $f \in F$.

Weighted Finite-State Acceptor

- A *weighted finite-state acceptor* is so-named because it *accepts* strings from Σ^*, the *Kleene closure* of the alphabet Σ, and assigns a weight to each accepted string.
- A string s is accepted by A iff there is a successful path π labeled with s through A.
- The label $l[\pi]$ for an entire path $\pi = e_1 \cdots e_K$ can be formed through the concatenation of all labels on the individual transitions:

$$l[\pi] \triangleq l[e_1] \cdots l[e_K].$$

- The weight $w[\pi]$ of a path π can be represented as

$$w[\pi] \triangleq \lambda \otimes w[e_1] \otimes \cdots \otimes w[e_K] \otimes \rho(n[e_K]),$$

  where $\rho(n[e_K])$ is the final weight.
- Typically, Σ contains ϵ, which, as stated before, denotes the null symbol.

Weighted Finite-State Transducers

- We now generalize our notion of a WFSA in order to consider machines that translate one string of symbols into a second string of symbols from a different alphabet along with a weight.

  Weighted finite-state transducer: A WFST $T = (\Sigma, \Omega, Q, E, i, F, \lambda, \rho)$ on the semiring Σ consists

  - of an *input alphabet* Σ,
  - an *output alphabet* Ω,
  - a set of states Q,
  - a set of transitions $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times \Sigma \times Q$
  - a initial state $i \in Q$ with weight λ,
  - a set of final states $F \subseteq Q$,
  - and a function ρ mapping from F to $\mathbb{R}^+$.

Transitions

A transition $e = (p[e], l_i[e], l_o[e], w[e], n[e]) \in E$ consists of

- a previous state $p[e]$,
- a next state $n[e]$,
- an input symbol $l_i[e]$,
- an output symbol $l_o[e]$, and
- a weight $w[e]$.

Weighted Finite-State Transducers (cont'd.)

- A WFST, such as that shown in Figure 2 maps an input string to an output string and a weight.
- For example, such a transducer would map the input string "red white blue" to the output string "yellow blue red" with a weight of $0.5 + 0.3 + 0.2 + 0.8 = 1.8$.
- It differs from the WFSA only in that the edges of the WFST have *two* labels, an input and an output, rather than one.
- A string s is accepted by a WFST T iff there is a successful path π labeled with $l_i[\pi] = s$.
- The weight of this path is $w[\pi]$, and its output string is

$$l_o[\pi] \triangleq l_o[e_1] \cdots l_o[e_K].$$

- Any ϵ–symbols appearing in $l_o[\pi]$ can be ignored.

Diagram of a Weighted Finite-State Transducer

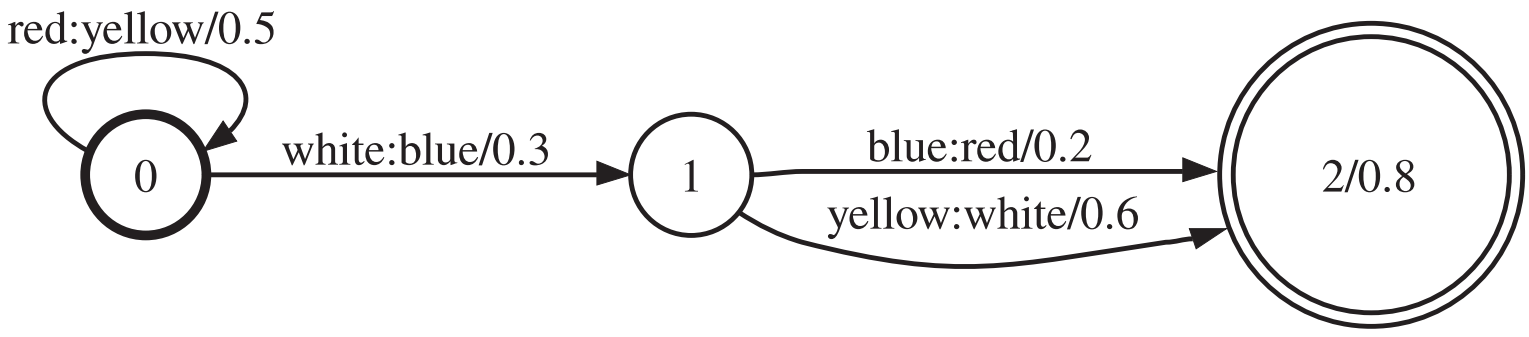


Figure: A simple weighted finite-state transducer.

Weighted Composition

- 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.

Composition without ϵ-Transitions

- In the absence then of ϵ–transitions, the construction of such a transducer R is straightforward.
- It entails simply 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 $n_T \in Q_T$, beginning with the initial nodes i_S and i_T.
- Each $n_R \in Q_R$ is uniquely determined by the pair (n_S, n_T).

Schematic of Weighted Composition

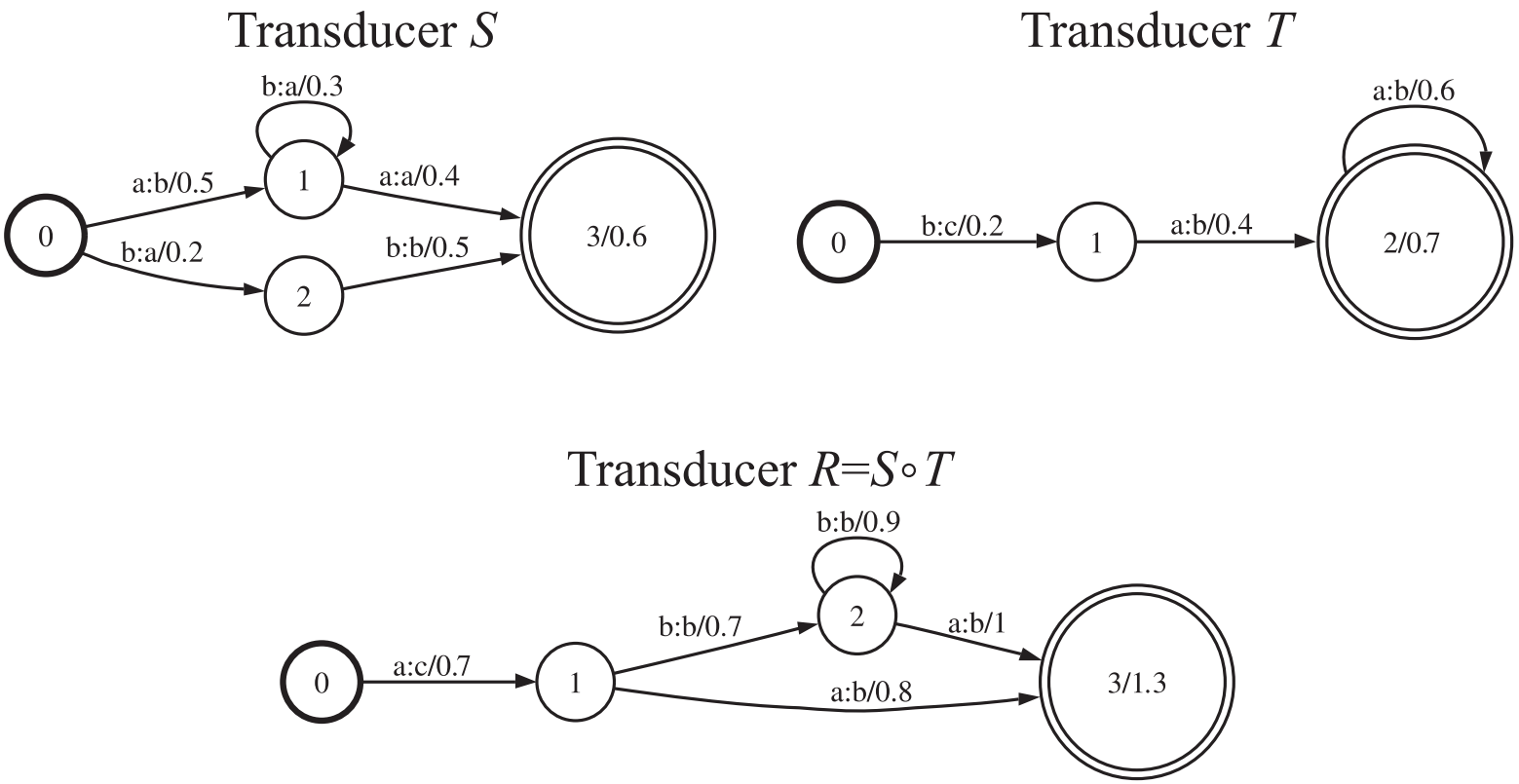


Figure: Weighted composition of two simple transducers.

- From the figure, it is clear that the transition from State 0 labeled with a:b/0.5 in S has been paired with the transition from State 0

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.

Composition Filter

- When ϵ–symbols are introduced, composition becomes more complicated, as it is necessary to specify when an ϵ–symbol on the output of a transition in n_S can be combined with an ϵ–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 ϵ-transition on either the input or output side is taken, placing the filter in State 1 or State 2 respectively, an ϵ-transition on the *other* side may not be taken until a non-ϵ match between input and output occurs.

Composition Filter Schematic

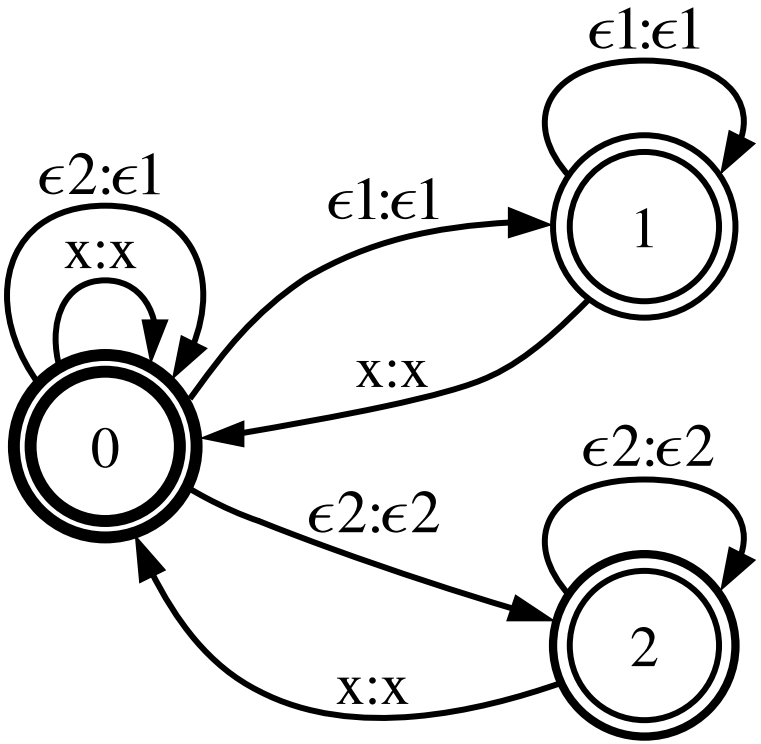


Figure: Filter used during composition with ϵ–symbols.

Weighted Determinization

- 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.

Advantages of Deterministic Transducers

- It is typically advantageous to work with deterministic WFSTs, because there is *at most* one path through the transducer labeled with a given input string.
- This implies that the effort required to learn if a given string is accepted by a transducer, and to calculate the associated weight and output string, is *linear* with the length of the string, and *does not* depend on the size of the transducer.
- More to the point, it implies that the acoustic likelihood that must be calculated when taking a transition during decoding need only be calculated *once*.
- This has a decisive impact on the efficiency of the search process inherent in ASR.

An Algorithm for Weighted Determinization

- Thus we are led to consider our first equivalence operation, *determinization*, which produces a deterministic transducer τ_2 that is equivalent to some given transducer τ_1.
  String-to-weight subsequential transducer: A *string-to-weight subsequential transducer* on the semiring Σ is an 8–tuple $\tau = (\Sigma, Q, i, F, \delta, \sigma, \lambda, \rho)$ consisting of
  - of an *input alphabet* Σ,
  - 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 δ mapping $Q \times \Sigma$ to Q,
  - a output function σ mapping $Q \times \Sigma$ to $\mathbb{R}^+$,
  - and a final weight function ρ mapping from F to $\mathbb{R}^+$.
- The determinization algorithm for weighted automata is similar to the classical powerset construction.

Pseudocode for Power Set Construction

The pseudocode for the power set construction is given below.

```
00  def powerSetConstruction(τ1, τ2):
01      F2 ← ∅
02      i2 ← i1
03      Q ← {i2}
04      while |Q| > 0:
05          pop q2 from Q
06          if ∃ q ∈ q2 such that q ∈ F1:
07              F2 ← F2 ∪ {q2}
08          for a such that δ(q2, a) ≠ ∅:
09              if δ2(q2, a) ∉ Q2:
10                  Q2 ← Q2 ∪ {δ2(q2, a)}
11                  push δ2(q2, a) on Q
```

Details of Weighted Determinization

- 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 τ_2 corresponds only to the initial state i_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 τ_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.

Details (cont'd.)

- Each time a new state in τ_2, consisting of a subset of the states of τ_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 τ_2 have been expanded and **Q** has been depleted, the algorithm terminates.

Weighted Determinization: Formal Description

- 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 Weighted Determinization

Pseudocode for the complete algorithm is provided in the listing below.

```
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) ← ⊕_{q∈F1,(q,x)∈q2} x ⊙ ρ1(q)
10      for a such that Γ(q2, a) ≠ ∅:
11        σ2(q2, a) ← ⊕_{(q,x)∈Γ(q2,a)} [ x ⊙ ⊕_{e=(q,a,σ1[e],n1[e])∈E1} σ1[e]]
12        δ2(q2, a) ← ∪_{q̂∈ν(q2,a)} { ( q̂, ⊕_{(q,x,t)∈γ(q2,a),n1[e]=q̂} [σ2(q2, a)]^{-1} ⊙ x ⊙ σ1[e] ) }
13        if δ2(q2, a) ∉ Q2 :
14          Q2 ← Q2 ∪ { δ2(q2, a) }
15          push δ2(q2, a) on Q
```

Pseudocode for 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 τ_2 to $\emptyset$ in Line 01, and equating the initial state and weight i_2 and λ_2 respectively to their counterparts in τ_1 in Lines 02–03.
- The initial state i_2 is pushed onto the queue **Q** (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

Pseudocode for Weighted Determinization

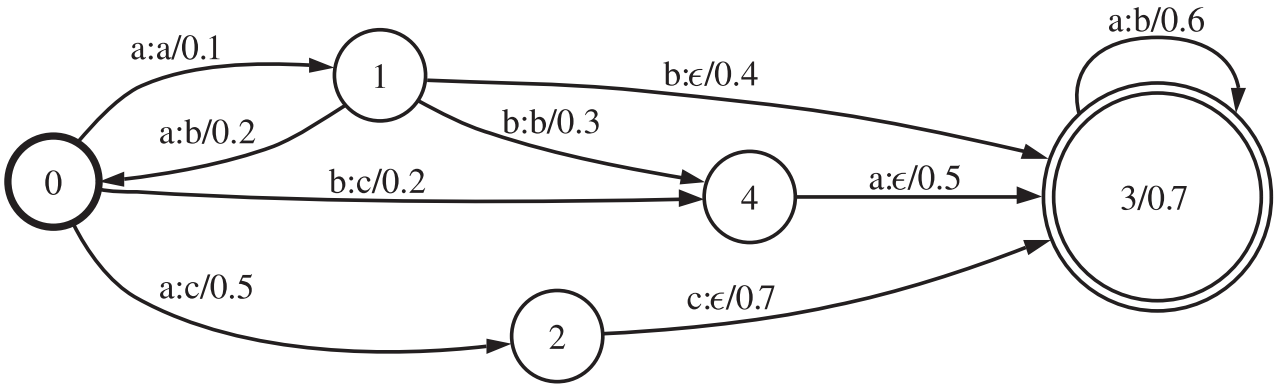
- The next step is to begin expanding the adjacency list of q_2 in Line 10, which specifies that the input symbols on the edges of the adjacency list of q_2 is obtained from the union of the input symbols on the adjacency lists of all q such that there exists $(q, x) \in q_2$.
- In Line 11, the weight 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 τ_2 in Line 14 and pushed onto the queue $\mathbf{Q}$.

Effects of Weighted Determinization

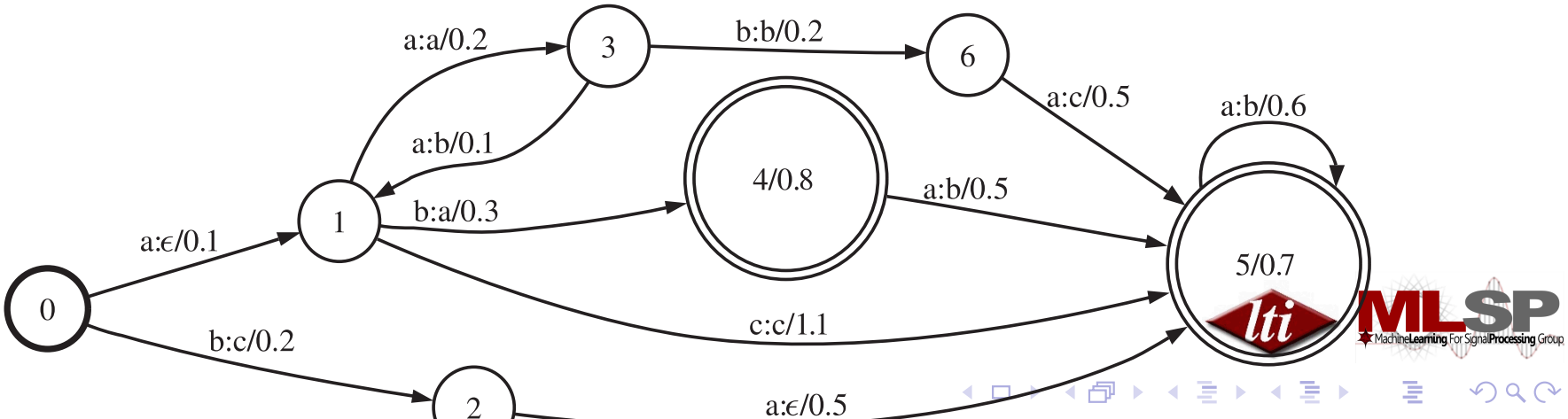
- A simple example of weighted determinization is shown in Figure 5.
- The two WFSTs in the figure are equivalent over the tropical semiring in that they both accept the same input strings, and for any given input string, produce the same output string and the same weight.
- For example, the original transducer will accept the input string *aba* along either of two successful paths, namely, using the state sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 3$ or the state sequence $0 \rightarrow 1 \rightarrow 4 \rightarrow 3$.
- 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.

Diagram of Weighted Determinization

Before Determinization



After Determinization



Summary

- In this lecture, we discussed algorithms for determing the *shortest* path through a weighted graph.
- Such problems are still tractable, however, and, for a graph $G = (V, E)$, can be solved in $\mathcal{O}(V, E)$ time.
- An important technique for solving such problems is that of *relaxation*, whereby the length of the shortest path is successively approximated.
- We considered two single-source shortest path algorithms:
  - Dijkstra's algorithm, which determines the shortest path from the source s to each $v \in V$ for a graph $G = (V, E)$ with positive-valued weights.
  - The Bellmann-Ford algorithm, which determines the shortest path from the source s to each $v \in V$ for a graph $G = (V, E)$ with real-valued weights, which may be negative.