Finite-state transducers

- $a : 0$ is a notation for a mapping between two alphabets $a \in \Sigma_1$ and $0 \in \Sigma_2$
- Finite-state transducers (FSTs) accept pairs of strings
- Finite-state automata equate to regular languages and FSTs equate to regular relations
- e.g. $L = \{ (x^n, y^n) : n > 0, x \in \Sigma_1$ and $y \in \Sigma_2 \}$ is a regular relation accepted by some FST. It maps a string of $x$’s into an equal length string of $y$’s
Finite-state transducers

\[ R(T_1) = R(T_2) = \{ (aa, 10), (ab, 1) \} \]
Finite-state transducers

Regular relations

• A generalization of regular languages
• The set of regular relations is:
  – The empty set and \((x,y)\) for all \(x, y \in \Sigma_1 \times \Sigma_2\) is a regular relation
  – If \(R_1, R_2\) and \(R\) are regular relations then:
    \[
    R_1 \cdot R_2 = \{(x_1x_2, y_1y_2) \mid (x_1, y_1) \in R_1, (x_2, y_2) \in R_2\}
    \]
    \[
    R_1 \cup R_2
    \]
    \[
    R^* = \bigcup_{i=0}^{\infty} R_i
    \]
  – There are no other regular relations
Finite-state transducers

• Formal definition:
  – \( Q \): finite set of states, \( q_0, q_1, \ldots, q_n \)
  – \( \Sigma \): alphabet composed of input/output pairs \( i:o \)
    where \( i \in \Sigma_1 \) and \( o \in \Sigma_2 \) and so \( \Sigma \subseteq \Sigma_1 \times \Sigma_2 \)
  – \( q_0 \): start state
  – \( F \): set of final states
  – \( \delta(q, i:o) \) is the transition function which returns
    a set of states

Finite-state transducers: Examples

• \((a^n, b^n)\): map \( n \) a’s into \( n \) b’s
• rot13 encryption (the Caesar cipher): assuming 26 letters
each letter is mapped to the letter 13 steps ahead (mod 26),
e.g. \( \text{cipher} \rightarrow \text{pvcure} \)
• reversal of a fixed set of words
• reversal of all strings upto fixed length \( k \)
• input: binary number \( n \), and output: binary number \( n+1 \)
• upcase or lowercase a string of any length
• *Pig latin: \( \text{pig latin is goofy} \rightarrow \text{igpay atinlay is oofygay} \)
• *convert numbers into pronunciations,
  e.g. \( 230.34 \) two hundred and thirty point three four
Finite-state transducers

• Following relations are cannot be expressed as a FST
  – \((a^n b^n, c^n)\): because \(a^n b^n\) is not regular
  – reversal of strings of any length
  – \(a^i b^j \rightarrow b^j a^i\) for any \(i, j\)

• Unlike regular languages, regular relations are not closed under intersection
  – \((a^n b^*, c^n) \cap (a^* b^n, c^n)\) produces \((a^n b^n, c^n)\)
  – However, regular relations with input and output of equal lengths are closed under intersection

Regular Relations Closure Properties

• Regular relations (rr) are closed under some operations
• For example, if \(R_1, R_2\) are regular relns:
  – union \((R_1 \cup R_2\) results in \(R_3\) which is a rr)
  – concatenation
  – iteration \((R_1^+ = \text{one or more repeats of } R_1)\)
  – Kleene closure \((R_1^* = \text{zero or more repeats of } R_1)\)
• However, unlike regular languages, regular relns are not closed under:
  – intersection (possible for equal length regular relns)
  – complement
Regular Relations Closure Properties

• New operations for regular relations:
  – composition
  – project input (or output) language to regular language; for FST \( t \), input language = \( \pi_1(t) \), output = \( \pi_2(t) \)
  – take a regular language and create the identity regular relation; for FSM \( f \), let FST for identity relation be \( \text{Id}(f) \)
  – take two regular languages and create the cross product relation; for FSMs \( f \& g \), FST for cross product is \( f \times g \)
  – take two regular languages, and mark each time the first language matches any string in the second language

Regular Relation/FST
Kleene Closure

![Diagram of a regular relation/FST](image)

1/23/08
Regular Expressions for FSTs

\[(a:c) (b:d)^*\]
\[ g:i \varepsilon:j (h:k)^* \]

\[ ((a:0 \| a:1) (b:0 \| b:1))^* \]
Subsequential FSTs

Sequential transducer = transducer with deterministic input

- Consider an FST in which for every symbol scanned from the input we can deterministically choose a path and produce an output.
- Such an FST is analogous to a deterministic FSM. It is called a subsequential FST.
- Subsequential transducers with \( p \) outputs on the final state is called a \( p \)-subsequential FST.
- \( p \)-subsequential FSTs can produce ambiguous outputs for a given input string.

\[
\begin{align*}
\text{input: abbaa} & \quad \text{output: bbab} \\
\text{ambiguos output:} & \quad \{\text{aaa, aab}\}
\end{align*}
\]
FST that is not subsequential

Input: $x^n$
Output: $a^n$ if $n$ is even, else $b^n$

FST Algorithms

- **Compose**: Given two FSTs $f$ and $g$ defining regular relations $R_1$ and $R_2$, create the FST $f \circ g$ that computes the composition: $R_1 \circ R_2$
- **Recognition**: Is a given pair of strings accepted by FST $t$?
- **Transduce**: Given an input string, provide the output string(s) as defined by the regular relation provided by an FST
Composing FSTs

What is $T_1$ composed with $T_2$, aka $T_1 \circ T_2$?

Composing FSTs
Composing FSTs

Start with pair of final states

\[(0,0) (1,1) a : a \quad (0,0) (2,1) b : a \quad (0,1) (0,1) a : d \quad (1,1) (3,1) b : d \]
\[(0,1) (1,2) a : a \quad (0,1) (2,2) b : a \quad (0,1) (0,2) a : c \quad (1,1) (3,2) b : c \]
\[(2,0) (3,1) b : a \quad (2,1) (3,2) b : a \]
Composing FSTs

0 1 \ a:b
0 2 \ b:b
2 3 \ b:b

(0,0) (1,1) \ a:a
(0,1) (1,2) \ a:a
(2,0) (3,1) \ b:b

0 1 \ b:a
0 2 \ b:b
1 2 \ b:a

(0,0) (2,1) \ b:a
(0,1) (2,2) \ b:a
(2,1) (3,2) \ b:a

0 0 \ a:a
1 1 \ a:d
1 2 \ a:c

(0,1) (1,1) \ a:d
(0,1) (1,3) \ b:d
(0,1) (0,2) \ a:c
(1,1) (3,2) \ b:c

1/23/08
Composing FSTs

\[ T_1 \circ T_2: \]

\[
\begin{array}{c}
0,0 \quad 1,1 \quad 2,1 \quad 3,2 \\
\quad \quad \quad \quad \quad \quad \\
\end{array}
\]

\[
\begin{array}{c}
a:a \\
b:a \\
\end{array}
\]

\[
\begin{array}{c}
b:c \\
\quad \\
\end{array}
\]

\[
\begin{array}{c}
ab := ac \\
bb := aa
\end{array}
\]

Composing FSTs

\[ ( a:c (b:d)^* ) \mid ( (e:g)^* f:h ) \]

g:i \varepsilon:j (h:k)^* 

e:i \varepsilon:j f:k \]
FST Composition

- Input: transducer S and T
- Transducer composition results in a new transducer with states and transitions defined by matching compatible input-output pairs:
  \[
  \text{match}(s,t) = \\
  \{ (s,t) \rightarrow^{s',t'} : s \rightarrow^{s',y} s' \in S.\text{edges} \text{ and } t \rightarrow^{y,z} t' \in T.\text{edges} \} \cup \\
  \{ (s,t) \rightarrow^{s',z} (s',t) : s \rightarrow^{s',x} s' \in S.\text{edges} \} \cup \\
  \{ (s,t) \rightarrow^{x,z} (s,t') : t \rightarrow^{x,z} t' \in T.\text{edges} \}
  \]
- Correctness: any path in composed transducer mapping u to w arises from a path mapping u to v in S and path mapping v to w in T, for some v

Complex FSTs with composition

- Take, for example, the task of constructing an FST for the Soundex algorithm
- Soundex is useful to map spelling variants of proper names to a single code (hashing names)
- It depends on a mapping from letters to codes
Soundex

- Mapping from letters to numbers:
  \[ b, f, p, v \rightarrow 1 \]
  \[ c, g, j, k, q, s, x, z \rightarrow 2 \]
  \[ d, t \rightarrow 3 \]
  \[ l \rightarrow 4 \]
  \[ m, n \rightarrow 5 \]
  \[ r \rightarrow 6 \]

Soundex

- The Soundex algorithm:
  - If two or more letters with the same number are adjacent in the input, or adjacent with intervening h’s or w’s, omit all but the first.
  - Retain the first letter and delete all occurrences of a, e, h, i, o, u, w, y.
  - Except for the first letter, change all letters into numbers.
  - Convert result into LNNN (letter and 3 numbers), either truncate or add 0s.
Soundex

- Example:
  - Losh-shkan, Los-qam
  - Loshhkan, Losqam
  - Lskn, Lsqm
  - L225, L225

- Other examples:
  - Euler (E460), Gauss (G200), Hilbert (H416), Knuth (K530), Lloyd (L300), Lukasiewicz (L222), and Wachs (W200)

Soundex

- How can we implement Soundex as a FST?
- For each step in Soundex, the FST is quite simple to write
- Writing a single FST from scratch that implements Soundex is quite challenging
- A simpler solution is to build small FSTs, one for each step, and then use FST composition to build the FST for Soundex
FST that is not subsequential

Input: $x^n$
Output: $a^n$ if $n$ is even, else $b^n$

Conversion to subsequential FST

Input: $x^n$
- Step1 output: $(x1lx2)*x2$ if $n$ is even, else $(x1lx2)*x1$
- Step2 output: reversal of Step1 output
- Step3 output: $a^n$ if $n$ is even, else $b^n$

Interesting fact: this can be done for any non-subsequential FST to convert it into a subsequential FST
Recognition of string pairs

function FSTRecognize (input[], output[], q):
    Agenda = { (start-state, 0, 0) }
    Current = (state, i, o) = pop(Agenda) // i :- inputIndex, o :- outputIndex
    while (true) {
        if (Current is an accept item) return accept
        else Agenda = Agenda ∪ GenStates(q, state, input, output, i, o)
        if (Agenda is empty) return reject
        else Current = (state, i, o) = pop(Agenda)
    }

generation function GenStates (q, state, input[], output[], i, o):
    return
    { (q', i, o) : for all q' = q(state, ε:ε) } ∪
    { (q', i, o+1) : for all q' = q(state, ε:output[o+1]) } ∪
    { (q', i+1, o) : for all q' = q(state, input[i+1]:ε) } ∪
    { (q', i+1, o+1) : for all q' = q(state, input[i+1], output[i+1]) }

Transduction: input → output

• The transduce operation for a FST \( t \) can be simulated efficiently using the following steps:
  1. Convert the input string into a FSM \( f \) (the machine only accepts the input string, nothing else).
  2. Convert \( f \) into a FST by taking \( \text{Id}(f) \) and compose with \( t \) to give a new FST \( g = \text{Id}(f) \circ t \). (note that \( g \) only contains those paths compatible with input \( f \))
  3. Finally project the output language of \( g \) to give a FSM for the output of transduce: \( \pi_2(g) \)
  4. Optionally, eliminate any transitions that only derive the empty string from the \( \pi_2(g) \) FST.

• What follows is an alternate version that attempts to produce all output strings
Transduction: input → output

function FSTtransduce (input[], q):
    Agenda = { (start-state, 0, []) } // each item contains list of partial outputs
    Current = (state, i, out) = pop(Agenda) // i :- inputIndex, out :- output-list
    output = ()
    while (true) {
        if (Current is an accept item) output ⊕ out
        else Agenda = Agenda ∪ GenStates(q, state, input, out, i)
        if (Agenda is empty) return output
        else Current = (state, i, o) = pop(Agenda)
    }
Transduction: input $\rightarrow$ output

function FSTTransduce (input[], q):
    Agenda = { (start-state, 0, []) } // each item contains list of partial outputs
    Current = (state, i, out) = pop(Agenda) // i :: inputIndex, out :: output-list
    output = ()
    while (true) {
        if (Current is an accept item) output $\oplus$ out
        else Agenda = Agenda $\cup$ GenStates(q, state, input, out, i)
        if (Agenda is empty) return output
        else Current = (state, i, o) = pop(Agenda)
    }

function GenStates (q, state, input[], out, i):
    return { (q’, i, out) : for all q’ = q(state, $\varepsilon$:$\varepsilon$) } $\cup$
    { (q’, i, out $\oplus$ newOut) : for all q’ = q(state, $\varepsilon$:newOut) } $\cup$
    { (q’, i+1, out) : for all q’ = q(state, input[i+1]:$\varepsilon$) } $\cup$
    { (q’, i+1, out $\oplus$ newOut) : for all q’ = q(state, input[i+1], newOut) }
Transduction: input $\rightarrow$ output

function FSTTransduce (input[], q):
  Agenda = { (start-state, 0, [ ]) } // each item contains list of partial outputs
  Current = (state, i, out) = pop(Agenda) // i :- inputIndex, out :- output-list
  output = ()
  while (true) {
    if (Current is an accept item) output $\oplus$ out
    else Agenda = Agenda $\cup$ GenStates(q, state, input, out, i)
    if (Agenda is empty) return output
    else Current = (state, i, o) = pop(Agenda)
  }

function GenStates (q, state, input[], out, i):
  return { (q', i, out) : for all q' = q(state, $\epsilon$:$\epsilon$) } $\cup$
  { (q', i, out $\oplus$ newOut) : for all q' = q(state, $\epsilon$:newOut) } $\cup$
  { (q', i+1, out) : for all q' = q(state, input[i+1]:$\epsilon$) } $\cup$
  { (q', i+1, out $\oplus$ newOut) : for all q' = q(state, input[i+1], newOut) }

$\oplus$ concatenates new output string to each item in out (the output list for each item)

Cross-product FST

• For regular languages $L_1$ and $L_2$, we have two FSAs, $M_1$ and $M_2$
  \[ M_1 = (\Sigma, Q_1, q_1, F_1, \delta_1) \]
  \[ M_2 = (\Sigma, Q_2, q_2, F_2, \delta_2) \]
• Then a transducer accepting $L_1 \times L_2$ is defined as:
  \[ T = (\Sigma, Q_1 \times Q_2, \langle q_1, q_2 \rangle, F_1 \times F_2, \delta) \]
  \[ \delta(\langle s_1, s_2 \rangle, a, b) = \delta_1(s_1, a) \times \delta_2(s_2, b) \]
  for any $s_1 \in Q_1, s_2 \in Q_2$ and $a, b \in \Sigma \cup \{\epsilon\}$
Summary

• Finite state transducers specify regular relations
  – Encoding problems as finite-state transducers
• Extension of regular expressions to the case of regular relations/FSTs
• FST closure properties: union, concatenation, composition
• FST special operations:
  – creating regular relations from regular languages (Id, cross-product);
  – creating regular languages from regular relations (projection)
• FST algorithms
  – Recognition, Transduction
  – Determinization, Minimization? (not all FSTs can be determined)