Finite-State Methods
Finite state acceptors (FSAs)

- Things you may know about FSAs:
  - Equivalence to regexps
  - Union, Kleene *, concat, intersect, complement, reversal
  - Determinization, minimization
  - Pumping, Myhill-Nerode

Defines the language \( a? c^* \)

\[
= \{a, ac, acc, accc, ..., \varepsilon, c, cc, ccc, ...\}
\]
n-gram models not good enough

- Want to model grammaticality
- A “training” sentence known to be grammatical:

```
BOS mouse traps catch mouse traps EOS
```

- Resulting trigram model has to overgeneralize:
  - allows sentences with 0 verbs
    ```
    BOS mouse traps EOS
    ```
  - allows sentences with 2 or more verbs
    ```
    BOS mouse traps catch mouse traps catch mouse traps EOS
    ```
  - Can’t remember whether it’s in subject or object (i.e., whether it’s gotten to the verb yet)
Want to model grammaticality

\texttt{BOS mouse traps catch mouse traps EOS}

Finite-state can capture the generalization here:

\textbf{Noun+ Verb Noun+}

- **preverbal states** (still need a verb to reach final state)
- **postverbal states** (verbs no longer allowed)

Allows arbitrarily long NPs (just keep looping around for another Noun modifier).

Still, never forgets whether it’s preverbal or postverbal!

(Unlike 50-gram model)
How powerful are regexps / FSAs?

- More powerful than n-gram models
  - The hidden state may “remember” arbitrary past context
  - With k states, can remember which of k “types” of context it’s in

- Equivalent to HMMs
  - In both cases, you observe a sequence and it is “explained” by a hidden path of states. The FSA states are like HMM tags.

- Appropriate for phonology and morphology

  Word = Syllable+
  = (Onset Nucleus Coda?)+
  = (C+ V+ C*)+
  = ( (b|d|f|...)+ (a|e|i|o|u)+ (b|d|f|...)* )+
How powerful are regexps / FSAs?

- But less powerful than CFGs / pushdown automata
  - Can’t do recursive center-embedding
  - Hmm, humans have trouble processing those constructions too ...

- This is the rat that ate the malt.
- This is the malt that the rat ate.

- This is the cat that bit the rat that ate the malt.
- This is the malt that the rat that the cat bit ate.

- This is the dog that chased the cat that bit the rat that ate the malt.
- This is the malt that [the rat that [the cat that [the dog chased] bit] ate].

finite-state can handle this pattern (can you write the regexp?)

but not this pattern, which requires a CFG
How powerful are regexps / FSAs?

- But less powerful than CFGs / pushdown automata
- More important: Less explanatory than CFGs
  - An CFG \textit{without} recursive center-embedding can be converted into an equivalent FSA – but the FSA will usually be far larger
  - Because FSAs can’t reuse the same phrase type in different places

\[ S = \]

\[ \text{duplicated structure} \]

\[ \text{duplicated structure} \]

\begin{center}
\textit{more elegant} – using nonterminals like this is equivalent to a CFG
\end{center}
We’ve already used FSAs this way ...

- CFG with regular expression on the right-hand side:
  \[ X \rightarrow (A \mid B) \ G \ H \ (P \mid Q) \]
  \[ NP \rightarrow (\text{Det} \mid \epsilon) \ Adj^* \ N \]

- So each nonterminal has a finite-state automaton, giving a “recursive transition network (RTN)”

\[ X \rightarrow \]

Automaton state replaces dotted rule \((X \rightarrow A \ G \ . \ H \ P)\)
We’ve already used FSAs once ..

NP → rules from the WSJ grammar become a single DFA

NP → ADJP ADJP JJ JJ NN NNS
   | ADJP DT NN
   | ADJP JJ NN
   | ADJP JJ NN NNS
   | ADJP JJ NNS
   | ADJP NN
   | ADJP NN NN
   | ADJP NN NNS
   | ADJP NNS
   | ADJP NPR
   | ADJP NPRS
   | DT
   | DT ADJP
   | DT ADJP , JJ NN
   | DT ADJP ADJP NN
   | DT ADJP JJ JJ NN
   | DT ADJP JJ NN
   | DT ADJP JJ NN NN

etc.

regular expression

DFA

etc.
But where can we put our weights?

- CFG / RTN

- bigram model of words or tags (first-order Markov Model)

- Hidden Markov Model of words and tags together??
Another useful FSA ...

Wordlist

/net/dict/words

25K words
206K chars

Network

FSM
17728 states,
37100 arcs

slide courtesy of L. Karttunen (modified)
Weights are useful here too!

Wordlist

clear 0
clever 1
ear 2
ever 3
fat 4
father 5

Network

c/0
l/0
e/0
a/0
r/0

compile

f/4
a/0
t/0
h/1
v/1
e/0

Computes a perfect hash!
Sum the weights along a word’s accepting path.
Example: Weighted acceptor

- Compute **number of paths** from each state *(Q: how?)*
- Successor states partition the path set
- Use offsets of successor states as arc weights
- **Q**: Would this work for an arbitrary numbering of the words?
Example: Unweighted transducer

V[head=vouloir, ...
  tense=Present,
  num=SG, person=P3]
  |
veut

the problem of morphology ("word shape") - an area of linguistics
**Example: Unweighted transducer**

vouloir +Pres +Sing + P3

Finite-state transducer

veut

VP [head=vouloir, ...]

V [head=vouloir, ...]
tense=Present,
num=SG, person=P3]

veut

canonical form

inflection codes

v o u l o i r

+Pres
+Sing
+P3

the relevant path

inflected form
Example: Unweighted transducer

- Bidirectional: generation or analysis
- Compact and fast
- Xerox sells for about 20 languages including English, German, Dutch, French, Italian, Spanish, Portuguese, Finnish, Russian, Turkish, Japanese, ...
- Research systems for many other languages, including Arabic, Malay

---

vouloir +Pres +Sing + P3

Finite-state transducer

veut

canonical form

inflection codes

the relevant path

inflected form
What is a function?

- Formally, a set of \(<\text{input}, \text{output}>\) pairs where each \text{input} \in \text{domain}, \text{output} \in \text{co-domain}.

- Moreover, \(\forall x \in \text{domain}, \exists \text{exactly one} \ y \text{ such that} \ <x,y> \in \text{the function.}

\[ \text{domain(square)} = \{0, 1, 2, 3, \ldots\} \]
\[ \text{range(square)} = \{0, 1, 4, 9, \ldots\} \]
What is a relation?

- **square:** \( \text{int} \rightarrow \text{int} \)
  
  \[
  = \{ \langle 0,0 \rangle, \langle 1,1 \rangle, \langle -1,1 \rangle, \langle 2,4 \rangle, \langle -2,4 \rangle, \\
  \langle 3,9 \rangle, \langle -3,9 \rangle, \ldots \} 
  \]

- **inverse(square):** \( \text{int} \rightarrow \text{int} \)
  
  \[
  = \{ \langle 0,0 \rangle, \langle 1,1 \rangle, \langle 1,-1 \rangle, \langle 4,2 \rangle, \langle 4,-2 \rangle, \\
  \langle 9,3 \rangle, \langle 9,-3 \rangle, \ldots \} 
  \]

- Is \( \text{inverse(square)} \) a function?
  
  \[
  \begin{align*}
  0 & \mapsto \{0\} & 9 & \mapsto \{3,-3\} & 7 & \mapsto \{\} & -1 & \mapsto \{\}
  \end{align*}
  \]

- No - we need a more general notion!
  
  - A **relation** is any set of \( \langle \text{input}, \text{output} \rangle \) pairs
Regular Relation (of strings)

- **Relation**: like a function, but multiple outputs ok
- **Regular**: finite-state
- **Transducer**: automaton w/ outputs

- $b \rightarrow \{b\}$  $a \rightarrow \{\}$
- $\text{aaaaa} \rightarrow \{\text{ac, aca, acab, acabc}\}$

- Invertible?
- Closed under composition?
Can weight the arcs: \( \rightarrow \) vs. \( \rightarrow \)

- \( b \rightarrow \{b\} \quad a \rightarrow \{\} \)
- \( \text{aaaaa} \rightarrow \{ac, aca, acab, acabc\} \)

How to find best outputs?
- For \( \text{aaaaa} \)?
- For all inputs at once?
Function from strings to ...

Acceptors (FSAs)

Unweighted

<table>
<thead>
<tr>
<th>Transition</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.5</td>
</tr>
<tr>
<td>ε</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Weighted

<table>
<thead>
<tr>
<th>Transition</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.5</td>
</tr>
<tr>
<td>ε</td>
<td>0.5</td>
</tr>
<tr>
<td>c</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Transducers (FSTs)

Strings

<table>
<thead>
<tr>
<th>Transition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x</td>
</tr>
<tr>
<td>ε</td>
<td>y</td>
</tr>
<tr>
<td>c</td>
<td>z</td>
</tr>
</tbody>
</table>

Numbers

<table>
<thead>
<tr>
<th>Transition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.5</td>
</tr>
<tr>
<td>ε</td>
<td>0.5</td>
</tr>
<tr>
<td>c</td>
<td>0.7</td>
</tr>
</tbody>
</table>

(String, num) pairs

<table>
<thead>
<tr>
<th>Transition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x/0.5</td>
</tr>
<tr>
<td>ε</td>
<td>y/0.5</td>
</tr>
<tr>
<td>c</td>
<td>z/0.7</td>
</tr>
</tbody>
</table>

{false, true}
Sample functions

Acceptors (FSAs)

Unweighted

{false, true}
Grammatical?

Weighted

numbers
How grammatical? Better, how likely?

Transducers (FSTs)

Weighted

strings
Markup Correction Translation

Weighted

(string, num) pairs
Good markups Good corrections Good translations

Unweighted

Good markups Good corrections Good translations
Terminology (acceptors)

Regular language
- defines
- compiles into
- implements
- matches
- matches
- accepts (or generates)

Regexp

String

FSA

recognizes
Terminology (transducers)

- defines
- compiles into
- implements
- matches
- matches
- accepts (or generates)
- recognizes

Regular relation

Regexp

FST

String pair
Perspectives on a Transducer

- Remember these CFG perspectives:

<table>
<thead>
<tr>
<th>3 views of a context-free rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>generation (production):</td>
</tr>
<tr>
<td>parsing (comprehension):</td>
</tr>
<tr>
<td>verification (checking):</td>
</tr>
</tbody>
</table>

- Similarly, 3 views of a transducer:
  - Given 0 strings, generate a new string pair (by picking a path)
  - Given one string (upper or lower), transduce it to the other kind
  - Given two strings (upper & lower), decide whether to accept the pair

FST just defines the regular relation (mathematical object: set of pairs). What’s “input” and “output” depends on what one asks about the relation. The 0, 1, or 2 given string(s) constrain which paths you can use.
Functions

\[ ab?d \rightarrow f \rightarrow abcd \rightarrow g \rightarrow \alpha\beta\chi\delta \]
Functions

Function composition: \( f \circ g \)

[first \( f \), then \( g \) – intuitive notation, but opposite of the traditional math notation]

Like the Unix pipe: `cat x | f | g > y`

Example: Pass the input through a sequence of ciphers
From Functions to Relations

\[ \text{ab?d} \rightarrow \text{abcd} \rightarrow \text{abed} \rightarrow \text{abjd} \rightarrow \cdots \]

\[ \text{f} \rightarrow \text{g} \rightarrow \alpha \beta \gamma \delta \rightarrow \alpha \beta \varepsilon \delta \rightarrow \alpha \beta \in \delta \rightarrow \cdots \]
From Functions to Relations

Relation composition: $f \circ g$

$ab?d \rightarrow \alpha\beta\epsilon\delta$

$\alpha\beta\gamma\delta$

$\alpha\beta\epsilon\delta$

$\alpha\beta\in\delta$

...
From Functions to Relations

Relation composition: $f \circ g$

$ab ? d \rightarrow\begin{array}{l}
3+4 \rightarrow \alpha \beta \gamma \delta \\
2+2 \rightarrow \alpha \beta \varepsilon \delta \\
2+8 \rightarrow \alpha \beta \in \delta \\
\ldots
\end{array}$
Often in NLP, all of the functions or relations involved can be described as finite-state machines, and manipulated using standard algorithms.
Inverting Relations

\[ \text{ab?d} \rightarrow \text{abcd} \rightarrow \text{abed} \rightarrow \text{abjd} \]

\[ \text{f} \]

\[ \text{g} \]

\[ \alpha \beta \gamma \delta \]

\[ \alpha \beta \epsilon \delta \]

\[ \alpha \beta \in \delta \]

\[ \ldots \]
Inverting Relations

\[ \begin{align*}
  f^{-1} & : ab?d \\ 
  & \rightarrow 3 \text{abcd} \\ 
  & \rightarrow 2 \text{abed} \\ 
  & \rightarrow 6 \text{abjd} \\ 
  g^{-1} & : \alpha \beta \gamma \delta \\ 
  & \rightarrow 4 \text{abcd} \\ 
  & \rightarrow 2 \text{abed} \\ 
  & \rightarrow 8 \text{abjd} \\ 
  & \quad \ldots
\end{align*} \]
Inverting Relations

\[(f \circ g)^{-1} = g^{-1} \circ f^{-1}\]
Building a lexical transducer

Regular Expression Lexicon -> Lexicon FSA -> Composed Rule FSTs -> Lexical Transducer (a single FST)

Compiler

Regular Expressions for Rules

big | clear | clever | ear | fat | ...

Composition

one path
Building a lexical transducer

- Actually, the lexicon must contain elements like big +Adj +Comp
- So write it as a more complicated expression:
  \[(\text{big} \mid \text{clear} \mid \text{clever} \mid \text{ear} \mid \text{fat} \mid \ldots) +\text{Adj} (\epsilon \mid +\text{Comp} \mid +\text{Sup}) \mid (\text{ear} \mid \text{father} \mid \ldots) +\text{Noun} (\text{+Sing} \mid +\text{PI}) \mid \ldots\]
- Q: Why do we need a lexicon at all?
Weighted version of transducer: Assigns a weight to each string pair

“upper language”

“lower language”

slide courtesy of L. Karttunen (modified)