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 \(a^* c^*\)

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

- Want to model grammaticality
- A “training” sentence known to be grammatical:
  
  BOS mouse traps catch mouse traps EOS

  trigram model must allow these trigrams

- 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 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)
Finite-state models can “get it”

- Want to model grammaticality
  
  BOS: *mouse traps catch mouse traps* EOS

- Finite-state can capture the generalization here:

  **Noun+ Verb Noun+**

  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)

- preverbal states (still need a verb to reach final state)

- postverbal states (verbs no longer allowed)
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 *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
We’ve already used FSAs this way ...

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

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

\[ X \rightarrow \]

\[ \text{Automaton state replaces dotted rule } (X \rightarrow A G . H P) \]
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
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
- clear
- clever
- ear
- ever
- fat
- father

Network

/usr/dict/words 0.6 sec
- 25K words
- 206K chars

FSM
- 17728 states,
- 37100 arcs

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

Wordlist

<table>
<thead>
<tr>
<th>Word</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>0</td>
</tr>
<tr>
<td>clever</td>
<td>1</td>
</tr>
<tr>
<td>ear</td>
<td>2</td>
</tr>
<tr>
<td>ever</td>
<td>3</td>
</tr>
<tr>
<td>fat</td>
<td>4</td>
</tr>
<tr>
<td>father</td>
<td>5</td>
</tr>
</tbody>
</table>

Network

Computes a perfect hash!
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?

Wordlist

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

Network

A: recursively, like DFS
Example: Unweighted transducer

\[ \text{VP} [\text{head}=\text{vouloir}, \ldots] \]

\[ \text{V} [\text{head}=\text{vouloir}, \ldots\text{tense}=\text{Present}, \text{num}=\text{SG}, \text{person}=\text{P3}] \]

\[ \text{veut} \]

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

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**

![Diagram of Finite-state transducer]

- **vouloir +Pres +Sing + P3**
- **Finite-state transducer**
- **veut**

- **canonical form**
- **inflection codes**
- **the relevant path**
- **inflected form**
Example: Weighted Transducer

Edit distance: Cost of best path relating these two strings?
Regular Relation (of strings)

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

- $b \rightarrow \{b\}$  \hspace{0.1cm}  $a \rightarrow \{}$
- $aaaaa \rightarrow \{ac, aca, acab, acabc\}$

- Invertible?
- Closed under composition?
Regular Relation (of strings)

- Can weight the arcs: $\to$ vs. $\to$
- $b \to \{b\}$ $a \to \{\}$
- $\text{aaaaa} \to \{\text{ac, aca, acab, acabc}\}$

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

Acceptors (FSAs)

Unweighted

{false, true}  

a  

ε  

c

Weighted

numbers  
a/.5  

ε/.5  

.3  

Weighted

(strings, num) pairs  
a:x/.5  

ε:y/.5  

.3  

Transducers (FSTs)

strings  
a:x  

ε:y  

c:z

Weighted

(numbers, num) pairs  
a/.5  

ε/.5  

.3  

Weighted

(strings, num) pairs  
a:x/.5  

ε:y/.5  

.3  

c:z/.7
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
Terminology (acceptors)

Regular language

Regexp

defines

compiles into

implements

String

matches

accepts (or generates)

FSA

matches

recognizes

matches
Terminology (transducers)

- **Regexp**
  - defines
  - matches

- **String pair**
  - matches

- **Regular relation**
  - compiles into
  - implements

- **FST**
  - recognizes
  - accepts (or generates)
  - (or, transduces one string of the pair into the other)
Perspectives on a Transducer

- Remember these CFG perspectives:

  **3 views of a context-free rule**

  - generation (production): $S \rightarrow NP \ VP$ (randsent)
  - parsing (comprehension): $S \leftarrow NP \ VP$ (parse)
  - verification (checking): $S = NP \ VP$

- 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

\[ \text{ab?d} \xrightarrow{f} \text{abcd} \xrightarrow{g} \alpha\beta\chi\delta \]
Functions

Function composition: $f \circ g$

[first f, then g – intuitive notation, but opposite of the traditional math notation]
From Functions to Relations

\[ f \]

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

\[ \text{abjd} \rightarrow \text{abed} \]

\[ g \]

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

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

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

...
From Functions to Relations

Relation composition: $f \circ g$

ab?d → 2 → 6 → 4

3 → 4: $\alpha \beta \gamma \delta$

2 → 2: $\alpha \beta \varepsilon \delta$

8 → 8: $\alpha \beta \in \delta$

...
From Functions to Relations

Relation composition: \( f \circ g \)
From Functions to Relations

Often in NLP, all of the functions or relations involved can be described as finite-state machines, and manipulated using standard algorithms.
Inverting Relations

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

\[ \text{f} \]

\[ \text{g} \rightarrow \text{\alpha\beta\gamma\delta} \rightarrow \text{\alpha\beta\epsilon\delta} \rightarrow \text{\alpha\beta\in\delta} \rightarrow \ldots \]
Inverting Relations

\[ f^{-1} \]

\[ ab?d \leftarrow f^{-1} \]

\[ 3 \]

\[ abcd \]

\[ 2 \]

\[ abed \]

\[ 6 \]

\[ abjd \]

\[ g^{-1} \]

\[ 4 \]

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

\[ 2 \]

\[ \alpha\beta\varepsilon\delta \]

\[ 8 \]

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

\[ \ldots \]
Inverting Relations

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

Regular Expression
Lexicon

Compiler

Lexicon
FSA

composition

Lexical Transducer
(a single FST)

Composed
Rule FSTs

Regular Expressions
for Rules

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

one path
Building a lexical transducer

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

- être+IndP +SG + P1
- suivre+IndP+SG+P1
- suivre+IndP+SG+P2
- suivre+Imp+SG + P2
- payer+IndP+SG+P1

“upper language”

“lower language”

Weighted French Transducer

slide courtesy of L. Karttunen (modified)