Finite-State Automata, Morphology and Tokenization

Most Slides from Jurafsky and Karttunen
Today

- Finite State Methods
- Non-Determinism (NFSAs)
- Recognition of NFSAs
- Proof that regular expressions = FSAs
- Very brief sketch: Morphology, FSAs, FSTs
- Very brief sketch: Tokenization and Segmentation
- Very brief sketch: Minimum Edit Distance
Finite-State Methods in NLP

- Domains of Application
  - Tokenization
  - Sentence breaking
  - Spelling correction
  - **Morphology (analysis/generation)**
  - Phonological disambiguation (Speech Recognition)
  - Morphological disambiguation ("Tagging")
  - Pattern matching ("Named Entity Recognition")
  - Shallow Parsing

- Types of Finite-State Systems
  - Classical (non-weighted) automata
  - Weighted (associated with weights in a semi-ring)
  - **Binary relations (simple transducers)**
  - N-ary relations (multi-tape transducers)
Substitutions and Memory

- **Substitutions**
  
  - `s/colour/color/`
  - `s/colour/color/g`
  - `s/colour/color/i`

  Substitute as many times as possible!

  Case insensitive matching

- **Memory** ($1$, $2$, etc. refer back to matches)

  - `/the (.*)er they were, the $1$er they will be/`
  - `/the (.*)er they (.*), the $1$er they $2$/`
User: Men are all alike
ELIZA: IN WHAT WAY
User: They’re always bugging us about something or other
ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE?
User: Well, my boyfriend made me come here
ELIZA: YOUR BOYFRIEND MADE YOU COME HERE
User: He says I’m depressed much of the time
ELIZA: I AM SORRY TO HEAR THAT YOU ARE DEPRESSED
Eliza-style regular expressions

Step 1: replace first person with second person references

\[s/\bI('m| am)\b /YOU ARE/g\]
\[s/\bmy\b /YOUR/g\]
\[s/\bmine\b /YOURS/g\]

Step 2: use additional regular expressions to generate replies

\[s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/\]
\[s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/\]
\[s/.* all .*/IN WHAT WAY/\]
\[s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/\]

Step 3: use scores to rank possible transformations

Slide from Dorr/Monz
Summary on REs so far

- Regular expressions are perhaps the single most useful tool for text manipulation
  - Simple but ubiquitous
- Eliza: you can do a lot with simple regular-expression substitutions
Three Views

- Three equivalent formal ways to look at what we’re up to

Diagram:

- Regular Expressions
- Regular Languages
- Finite State Automata
- Regular Grammars
Finite State Automata

- Terminology: Finite State Automata, Finite State Machines, FSA, Finite Automata

- Regular expressions are one way of specifying the structure of finite-state automata.

- FSAs and their close relatives are at the core of most algorithms for speech and language processing.
Finite-state Automata (Machines)

Slide from Dorr/Monz
We can say the following things about this machine:
- It has 5 states
- At least b, a, and ! are in its alphabet
- q0 is the start state
- q4 is an accept state
- It has 5 transitions
But note

There are other machines that correspond to this language.

More on this one later.
More Formally: Defining an FSA

- You can specify an FSA by enumerating the following things.
  - The set of states: $Q$
  - A finite alphabet: $\Sigma$
  - A start state $q_0$
  - A set $F$ of accepting/final states $F \subseteq Q$
  - A transition function $\delta(q,i)$ that maps $Q \times \Sigma$ to $Q$
Yet Another View

State-transition table

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b 1</td>
</tr>
<tr>
<td>1</td>
<td>a 2</td>
</tr>
<tr>
<td>2</td>
<td>3 0</td>
</tr>
<tr>
<td>3</td>
<td>3 4</td>
</tr>
<tr>
<td>4:</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
Recognition

- Recognition is the process of determining if a string should be accepted by a machine.
- Or... it’s the process of determining if a string is in the language we’re defining with the machine.
- Or... it’s the process of determining if a regular expression matches a string.
Recognition

Traditionally, (Turing’s idea) this process is depicted with a tape.
Recognition

- Start in the start state
- Examine the current input
- Consult the table
- Go to a new state and update the tape pointer.
- Until you run out of tape.
Input Tape

Slide from Dorr/Monz
Input Tape

Slide from Dorr/Monz
Adding a failing state
function D-RECOGNIZE (tape, machine) returns accept or reject

index ← Beginning of tape

current-state ← Initial state of machine

loop

if End of input has been reached then
    if current-state is an accept state then
        return accept
    else
        return reject
elsif transition-table [current-state, tape[index]] is empty then
    return reject
else
    current-state ← transition-table [current-state, tape[index]]
    index ← index + 1
end

Slide from Dorr/Monz
Tracing D-Recognize

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 2 0</td>
</tr>
<tr>
<td>2</td>
<td>0 3 0</td>
</tr>
<tr>
<td>3</td>
<td>0 3 4</td>
</tr>
<tr>
<td>4:</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_3 \rightarrow q_4$

Input: b a a a a !
Key Points

- Deterministic means that at each point in processing there is always one unique thing to do (no choices).
- D-recognize is a simple table-driven interpreter
- The algorithm is universal for all unambiguous languages.
  - To change the machine, you change the table.
Key Points

- Crudely therefore... matching strings with regular expressions (ala Perl) is a matter of
  - translating the expression into a machine (table) and
  - passing the table to an interpreter
Generative Formalisms

- **Formal Languages** are sets of strings composed of symbols from a finite set of symbols.
- Finite-state automata define formal languages (without having to enumerate all the strings in the language)
- The term **Generative** is based on the view that you can run the machine as a generator to get strings from the language.
Generative Formalisms

- FSAs can be viewed from two perspectives:
  - Acceptors that can tell you if a string is in the language
  - Generators to produce all and only the strings in the language
Dollars and Cents
Non-determinism

- A deterministic automaton is one whose behavior during recognition is fully determined by the state it is in and the symbol it is looking at.
- Non-determinism: not fully determined, hence choice
Non-Determinism
Non-Determinism cont.

- Yet another technique
  - Epsilon transitions
  - These transitions do not examine or advance the tape during recognition
Non-deterministic machines can be converted to deterministic ones with a fairly simple construction.

That means that they have the same power; non-deterministic machines are not more powerful than deterministic ones.

It also means that one way to do recognition with a non-deterministic machine is to turn it into a deterministic one.
Non-Deterministic Recognition

- In a ND FSA there exists at least one path through the machine for a string that is in the language defined by the machine.
- But not all paths directed through the machine for an accept string lead to an accept state.
- No paths through the machine lead to an accept state for a string not in the language.
Non-Deterministic Recognition

- So **success** in a non-deterministic recognition occurs when a path is found through the machine that ends in an accept.
- **Failure** occurs when none of the possible paths lead to an accept state.
Example
Using NFSA to accept strings

- In general, solutions to the problem of *choice* in non-deterministic models:
  - **Backup:**
    - When we come to a choice point
    - Put a marker indicating:
      - Where we are in the tape
      - What the state is
  - **Lookahead**
  - **Parallelism**
Key AI idea: Search

- We model problem-solving as a search for a solution.
- Through a space of possible solutions.
- The space consists of states.
- States in the search space are pairings of tape positions and states in the machine.
- By keeping track of as yet unexplored states, a recognizer can systematically explore all the paths through the machine given an input.
Two kinds of search

- Depth-first search
  - Explore one path all the way to the end
  - Then backup
  - And try other paths

- Breadth-first search
  - Explore all the paths simultaneously
  - Incrementally extending each tier of the paths
Depth-first search example
Depth-first search example

1

\[ \begin{array}{c}
q_0 \\
baaa!
\end{array} \]

2

\[ \begin{array}{c}
q_0 \\
q_1 \\
baaa!
\end{array} \]
Depth-first search example
Depth-first search example
Depth-first search example

1. \( q_0 \) - \( baaa! \)
2. \( q_0 \) - \( ba a! \)
3. \( q_1 \) - \( baaa! \)
4. \( q_3 \) - \( baaa! \)
5. \( q_3 \) - \( baaa! \)
Depth-first search example

1

2

3

4

5

6
Depth-first search example
Depth-first search example
NFSA Recognition of “baaa!”
Breadth-first Recognition of “baaa!”

should be q₂
Regular languages

- Regular languages are characterized by FSAs.
- For every NFSA, there is an equivalent DFSA.
- Regular languages are closed under concatenation, Kleene closure, union.
Regular languages

- The class of languages characterizable by regular expressions

- Given alphabet $\Sigma$, the regular languages over $\Sigma$ are:
  - The empty set $\emptyset$ is a regular language
  - $\forall a \in \Sigma \cup \varepsilon$, $\{a\}$ is a regular language
  - If $L_1$ and $L_2$ are regular languages, then so are:
    - $L_1 \cdot L_2 = \{xy|x \in L_1, y \in L_2\}$, concatenation of $L_1$ & $L_2$
    - $L_1 \cup L_2$, the union of $L_1$ and $L_2$
    - $L_1^*$, the Kleene closure of $L_1$
Going from regexp to FSA

- Since all regular lgs meet above properties
- And reg lgs are the lgs characterizable by regular expressions
- All regular expression operators can be implemented by combinations of union, disjunction, closure
  - Counters (*,+), are repetition plus closure
  - Anchors are individual symbols
  - [] and () and . are kinds of disjunction
Going from regexp to FSA

So if we could just show how to turn closure/union/concat from regexps to FSAs, this would give an idea of how FSA compilation works.

The actual proof that reg lgs = FSAs has 2 parts

- An FSA can be built for each regular lg
- A regular lg can be built for each automaton

So I’ll give the intuition of the first part:

- Take any regular expression and build an automaton
- Intuition: induction
  - Base case: build an automaton for single symbol (say ‘a’), as well as epsilon and the empty language
  - Inductive step: Show how to imitate the 3 regexp operations in automata
Union

- Accept a string in either of two languages
Accept a string consisting of a string from language L1 followed by a string from language L2.
Kleene Closure

Accept a string consisting of a string from language $L_1$ repeated zero or more times.
Summary so far

- Finite State Automata
  - Deterministic Recognition of FSAs
  - Non-Determinism (NFSAs)
  - Recognition of NFSAs
  - (sketch of) Proof that regular expressions = FSAs
FSAs and Computational Morphology

- An important use of FSAs is for morphology, the study of word parts
Computational morphology

**Analysis**
- leaf N Pl
- leave N Pl
- leave V Sg3

**Generation**
- hang V Past
  - hanged
  - hung
Two challenges

- **Morphotactics**
  - Words are composed of smaller elements that must be combined in a certain order:
    - *piti-less-ness* is English
    - *piti-ness-less* is not English

- **Phonological alternations**
  - The shape of an element may vary depending on the context
    - *pity* is realized as *piti* in *pitilessness*
    - *die* becomes *dy* in *dying*
Morphology is regular (=rational)

- The relation between the **surface forms** of a language and the corresponding **lexical forms** can be described as a **regular relation**.

- A regular relation consists of ordered pairs of strings.
  - *leaf*+N+*Pl* : leaves    *hang*+V+*Past* : hung

- Any finite collection of such pairs is a regular relation.

- **Regular relations are closed** under operations such as **concatenation**, **iteration**, **union**, and **composition**.

- Complex regular relations can be derived from simple relations.
Morphology is finite-state

- A regular relation can be defined using the metalanguage of **regular expressions**.

  \[
  \text{\{talk\} | \{walk\} | \{work\}}
  \]

  \[
  \text{%+Base:0 | %+SgGen3:s | %+Progr:\{ing\} | %+Past:\{ed\}}
  \]

- A regular expression can be compiled into a **finite-state transducer** that implements the relation computationally.
Compilation

Regular expression

```
q [{talk} | {walk} | {work}]
q [%+Base:0 | %+SgGen3:s | %+Progr:{ing} | %+Past:{ed}];
```

Finite-state transducer

[Diagram of a finite-state transducer with states labeled with transitions for the regular expression provided.]
Generation

work+3rdSg --> works

+Base: 0
+3rdSg:s
+Past:e
+Progr:i
0:n
0:g
0:d

w:w → a:a → 1:1 → k:k

o:o → r:r → t:t → a:a
The diagram represents the analysis of the word "talked" as it transforms into the past form "talk+Past". The transformation involves several linguistic changes indicated by the arrows and labels:

- **t** changes to **k** (change in initial sound).
- **a** remains **a** (no change in vowel).
- **w** changes to **l** (change in consonant).
- **o** changes to **r** (change in vowel).
- **e** changes to **e** (no change in vowel).

The diagram shows the stages of this transformation, ending with the label **talk+Past**. The symbols +Base, +3rdSg, +Past, and +Progr are used to denote specific linguistic features of the transformation process.
Lexical transducer

Bidirectional: generation or analysis
Compact and fast
Xerox systems have been built for over 40 languages:

- English, German, Dutch, French
- Italian, Spanish, Portuguese
- Finnish, Russian, Turkish
- Japanese, Korean, Basque
- Greek, Arabic, Hebrew, Bulgarian, ...
Function from strings to ...

### Acceptors (FSAs)
- **Unweighted**
  - States: a, ε
  - Transitions: a → a
  - Accepting states: {false, true}

### Transducers (FSTs)
- **Weighted**
  - States: a, c, ε
  - Transitions: a:x → a, ε:y → ε

- **Unweighted**
  - States: a, ε
  - Transitions: a → a, ε
  - Accepting states: strings

- **Weighted**
  - States: a, c, ε
  - Transitions: a: x/.5, ε: y/.5, c: z

### Labels
- Numbers
  - a/.5, c/.7
- Strings/Num Pairs
  - (string, num) pairs
  - a:x/.5, a: x/.5, ε: y/.5, ε: y/.5

### Weights
- ε: y/.5, ε: y/.5, ε: y/.5
Sample functions

**Acceptors (FSAs)**

- Unweighted
  - Grammatical?
  - {false, true}

**Transducers (FSTs)**

- Weighted
  - How grammatical?
  - Better, how likely?
  - (string, num) pairs

- Unweighted
  - Markup
  - Correction
  - Translation
  - strings
How lexical transducers are made

Morphotactics

Lexicon
Regular Expression

Compiler
composition

Rule
FSTs

Alternations

Lexicon
FST

Lexical Transducer
(a single FST)
Sequential Model

Lexical form

\[
\text{fst 1} \quad \text{Intermediate form} \quad \text{fst 2} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{fst n} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{Surface form}
\]

Ordered sequence of rewrite rules (Chomsky & Halle ‘68) can be modeled by a cascade of finite-state transducers Johnson ‘72 Kaplan & Kay ‘81
Phonological rewrite rules are not as powerful as they appear because of the constraint that a rule does not apply to its own output. (Johnson 1972, Kaplan&Kay 1980).

Figure 1: Two ways of applying $\varepsilon \rightarrow ab / \_b$
Parallel Model

Set of parallel
of two-level rules (constraints)
compiled into finite-state automata
interpreted as transducers
Koskenniemi ‘83
Sequential vs. parallel rules

Chomsky & Halle 1968

Koskenniemi 1983

Lexical form

Rule 1

Intermediate form

Rule 1

... compose

Surface form

Rule n

FST

Lexical form

Rule 1

Rule 2

... intersect

Surface form

Rule n
Rewrite Rules vs. Constraints

• Two different ways of decomposing the complex relation between lexical and surface forms into a set of simpler relations that can be more easily understood and manipulated.

• One approach may be more convenient than the other for particular applications.
English Morphology

- Morphology is the study of the ways that words are built up from smaller meaningful units called morphemes.

- We can usefully divide morphemes into two classes:
  - **Stems**: The core meaning bearing units.
  - **Affixes**: Bits and pieces that adhere to stems to change their meanings and grammatical functions.
Nouns and Verbs (English)

- Nouns are simple (not really)
  - Markers for plural and possessive
- Verbs are only slightly more complex
  - Markers appropriate to the tense of the verb
Ok so it gets a little complicated by the fact that some words misbehave (refuse to follow the rules)

- Mouse/mice, goose/geese, ox/oxen
- Go/went, fly/flew

The terms regular and irregular will be used to refer to words that follow the rules and those that don’t.
Regular and Irregular Nouns and Verbs

- Regulars...
  - Walk, walks, walking, walked, walked
  - Table, tables

- Irregulars
  - Eat, eats, eating, ate, eaten
  - Catch, catches, catching, caught, caught
  - Cut, cuts, cutting, cut, cut
  - Goose, geese
Compute

- Many paths are possible…
- Start with `compute`
  - Computer -> computerize -> computerization
  - Computation -> computational
  - Computer -> computerize -> computerizable
  - Compute -> computee
Why care about morphology?

- `Stemming’ in information retrieval
  - Might want to search for “going home” and find pages with both “went home” and “will go home”

- Morphology in machine translation
  - Need to know that the Spanish words *quiero* and *quieres* are both related to *querer* ‘want’

- Morphology in spell checking
  - Need to know that *misclam* and *antiundoggingly* are not words despite being made up of word parts
Can’t just list all words

- Turkish
- Uygarlastiramadiklarimizdanmissinizicasina
- `(behaving) as if you are among those whom we could not civilize’
- Uygar `civilized’ + las `become’ + tir `cause’ + ama `not able’ + dik `past’ + lar ‘plural’+ imiz ‘p1pl’ + dan ‘abl’ + mis ‘past’ + siniz ‘2pl’ + casina ‘as if’
What we want

- Something to automatically do the following kinds of mappings:
  - Cats: `cat +N +PL`
  - Cat: `cat +N +SG`
  - Cities: `city +N +PL`
  - Merging: `merge +V +Present-participle`
  - Caught: `catch +V +past-participle`
<table>
<thead>
<tr>
<th>English</th>
<th>Morphologically Parsed Output</th>
<th>Spanish</th>
<th>Morphologically Parsed Output</th>
<th>Gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats</td>
<td>cat +N +PL</td>
<td>pavos</td>
<td>pavo +N +Masc +Pl</td>
<td>‘ducks’</td>
</tr>
<tr>
<td>cat</td>
<td>cat +N +SG</td>
<td>pavo</td>
<td>pavo +N +Masc +Sg</td>
<td>‘duck’</td>
</tr>
<tr>
<td>cities</td>
<td>city +N +Pl</td>
<td>bebo</td>
<td>beber +V +PInd +1P +Sg</td>
<td>‘I drink’</td>
</tr>
<tr>
<td>geese</td>
<td>goose +N +Pl</td>
<td>canto</td>
<td>cantar +V +PInd +1P +Sg</td>
<td>‘I sing’</td>
</tr>
<tr>
<td>goose</td>
<td>goose +N +Sg</td>
<td>canto</td>
<td>canto +N +Masc +Sg</td>
<td>‘song’</td>
</tr>
<tr>
<td>goose</td>
<td>goose +V</td>
<td>puse</td>
<td>poner +V +Perf +1P +Sg</td>
<td>‘I was able’</td>
</tr>
<tr>
<td>gooses</td>
<td>goose +V +1P +Sg</td>
<td>vino</td>
<td>venir +V +Perf +3P +Sg</td>
<td>‘he/she came’</td>
</tr>
<tr>
<td>merging</td>
<td>merge +V +PresPart</td>
<td>vino</td>
<td>vino +N +Masc +Sg</td>
<td>‘wine’</td>
</tr>
<tr>
<td>caught</td>
<td>catch +V +PastPart</td>
<td>lugar</td>
<td>lugar +N +Masc +Sg</td>
<td>‘place’</td>
</tr>
<tr>
<td>caught</td>
<td>catch +V +Past</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.2**  Output of a morphological parse for some English and Spanish words. Spanish output modified from the Xerox XRCE finite-state language tools.
FSAs and the Lexicon

- This will actually require a Finite State Transducer (FST)
- First we’ll capture the morphotactics
  - The rules governing the ordering of affixes in a language.
- Then we’ll add in the actual words
Building a Morphological Parser

Three components:
- Lexicon
- Morphotactics
- Orthographic or Phonological Rules
Lexicon: FSA Inflectional Noun Morphology

• English Noun Lexicon

<table>
<thead>
<tr>
<th>reg-noun</th>
<th>Irreg-pl-noun</th>
<th>Irreg-sg-noun</th>
<th>plural</th>
</tr>
</thead>
<tbody>
<tr>
<td>fox</td>
<td>geese</td>
<td>goose</td>
<td>-s</td>
</tr>
<tr>
<td>cat</td>
<td>sheep</td>
<td>sheep</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>mice</td>
<td>mouse</td>
<td></td>
</tr>
</tbody>
</table>

• English Noun Rule
Lexicon and Rules: FSA English Verb Inflectional Morphology

<table>
<thead>
<tr>
<th>reg-verb-stem</th>
<th>irreg-verb-stem</th>
<th>irreg-past-verb</th>
<th>past</th>
<th>past-part</th>
<th>pres-part</th>
<th>3sg</th>
</tr>
</thead>
<tbody>
<tr>
<td>walk</td>
<td>cut</td>
<td>caught</td>
<td>-ed</td>
<td>-ed</td>
<td>-ing</td>
<td>-s</td>
</tr>
<tr>
<td>fry</td>
<td>speak</td>
<td>ate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>talk</td>
<td>spoken</td>
<td>eaten</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>impeach</td>
<td>sing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Initial state: $q_0$
- States: $q_0, q_1, q_2, q_3$
- Transitions:
  - $q_0 \rightarrow q_1$: reg-verb-stem
  - $q_1 \rightarrow q_2$: reg-verb-stem
  - $q_2 \rightarrow q_3$: reg-verb-stem
  - $q_1 \rightarrow q_3$: pst participle (-ed)
  - $q_2 \rightarrow q_3$: prog (-ing)
  - $q_3 \rightarrow q_3$: 3-sing (-s)
  - Irregular past verb form
  - Preterite (-ed)

Examples:
- walk -> -ed
- ate -> -ed
- knot -> -ed
- speak -> -ed
- spoken -> -ed
More Complex Derivational Morphology
Using FSAs for Recognition: English Nouns and Inflection
Parsing/Generation vs. Recognition

- We can only recognize words
- But this isn’t the same as parsing
  - Parsing: building structure
  - Usually if we find some string in the language we need to find the structure in it (parsing)
  - Or we have some structure and we want to produce a surface form (production/generation)
- Example
  - From “cats” to “cat +N +PL”
Finite State Transducers

- The simple story
  - Add another tape
  - Add extra symbols to the transitions

- On one tape we read “cats”, on the other we write “cat +N +PL”
Nominal Inflection FST
4. Tokenization

- Segmenting words in running text
- Segmenting sentences in running text
- Why not just periods and white-space?
  - Mr. Sherwood said reaction to Sea Containers’ proposal has been "very positive." In New York Stock Exchange composite trading yesterday, Sea Containers closed at $62.625, up 62.5 cents.
  - “I said, ‘what’re you? Crazy?’” said Sadowsky. “I can’t afford to do that.’”

Words like:
- cents. said, positive.” Crazy?
Can’t just segment on punctuation

- Word-internal punctuation
  - M.p.h
  - Ph.D.
  - AT&T
  - 01/02/06
  - Google.com
  - 555,500.50

- Expanding clitics
  - What’re -> what are
  - I’m -> I am

- Multi-token words
  - New York
  - Rock ‘n’ roll
Sentence Segmentation

- !, ? relatively unambiguous
- Period “." is quite ambiguous
  - Sentence boundary
  - Abbreviations like Inc. or Dr.

General idea:
- Build a binary classifier:
  - Looks at a “.”
  - Decides EndOfSentence/NotEOS
  - Could be hand-written rules, or machine-learning
Word Segmentation in Chinese

- Some languages don’t have spaces
  - Chinese, Japanese, Thai, Khmer

- Chinese:
  - Words composed of characters
  - Characters are generally 1 syllable and 1 morpheme.
  - Average word is 2.4 characters long.

- Standard segmentation algorithm:
  - Maximum Matching (also called Greedy)
Maximum Matching Word Segmentation

- Given a wordlist of Chinese, and a string.
- Start a pointer at the beginning of the string
- Find the longest word in dictionary that matches the string starting at pointer
- Move the pointer over the word in string
- Go to 2
English example (Palmer 00)

- the table down there
- thetabledownthere
- Theta bled own there

- Words astonishingly well in Chinese
- Far better than this English example suggests
- Modern algorithms better still:
  - probabilistic segmentation
5. Spell-checking and Edit Distance

- Non-word error detection:
  - detecting “graffe”

- Non-word error correction:
  - figuring out that “graffe” should be “giraffe”

- Context-dependent error detection and correction:
  - Figuring out that “war and piece” should be peace
Non-word error detection

- Any word not in a dictionary
- Assume it’s a spelling error
- Need a big dictionary!
- What to use?
  - FST dictionary!!
Isolated word error correction

How do I fix “graffe”?  
- Search through all words:  
  - graf  
  - craft  
  - grail  
  - giraffe  
- Pick the one that’s closest to graffe  
- What does “closest” mean?  
- We need a distance metric.  
- The simplest one: edit distance.  
  - (More sophisticated probabilistic ones: noisy channel)
Edit Distance

- The minimum edit distance between two strings
- Is the minimum number of editing operations
  - Insertion
  - Deletion
  - Substitution
- Needed to transform one into the other
Minimum Edit Distance

- If each operation has cost of 1
- Distance between these is 5
- If substitutions cost 2 (Levenshtein)
- Distance between these is 8
### Dynamic Programming Table

The table represents a dynamic programming table used to calculate the minimum cost of transforming one string into another. The table is used to solve the alignment problem, where the goal is to find the optimal way to align two sequences.

#### Formula

The formula for calculating the distance between two sequences is given by:

$$\text{distance}[i, j] = \min \left\{ \begin{array}{l}
\text{distance}[i-1, j] + \text{ins-cost}(\text{target}_{i-1}) \\
\text{distance}[i-1, j-1] + \text{subst-cost}(\text{source}_{j-1}, \text{target}_{i-1}) \\
\text{distance}[i, j-1] + \text{del-cost}(\text{source}_{j-1})
\end{array} \right. $$

where:
- distance is the gap matrix,
- ins-cost is the insertion cost,
- subst-cost is the substitution cost,
- del-cost is the deletion cost.

#### Example

The table below shows an example of a dynamic programming table for aligning two sequences. The sequences are:

```
#  E  X  E  C  U  T  I  O  N
```

```
N  9
O  8
1  7
```

The table is filled with the initial values and the minimum cost values are calculated row by row and column by column.
<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>9</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>I</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>T</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>N</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>#</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>#</td>
<td>E</td>
<td>X</td>
<td>E</td>
<td>C</td>
<td>U</td>
<td>T</td>
<td>I</td>
<td>O</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suppose we want the alignment too

- We can keep a “backtrace”
- Every time we enter a cell, remember where we came from
- Then when we reach the end, we can trace back from the upper right corner to get an alignment
<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>9</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>I</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>T</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>N</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>E</td>
<td>X</td>
<td>E</td>
<td>C</td>
<td>U</td>
<td>T</td>
<td>I</td>
<td>O</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Minimum Edit Distance
- A “dynamic programming” algorithm
- We will see a probabilistic version of this called “Viterbi”
Summary

- Finite State Automata
- Deterministic Recognition of FSAs
- Non-Determinism (NFSAs)
- Recognition of NFSAs
- Proof that regular expressions = FSAs
- Very brief sketch: Morphology, FSAs, FSTs
- Very brief sketch: Tokenization
- Minimum Edit Distance