

# Natural Language Processing (CSE 517 & 447): Weighted Finite-State Transducers

Noah Smith

© 2022

University of Washington  
nasmith@cs.washington.edu

Winter 2022

Readings: Eisenstein (2019) 9.0–9.1

# Motivation

- ▶ Dominant perspective in NLP in the 1970s–80s: formal language theory
- ▶ Engineering approach: expert-crafted, formally constrained, purely symbolic systems
- ▶ Relevance today: computational models of morphology

# Morphology

Extensive overview: Bender (2013)

race → races

race → racing

race → raced

# Morphology

Extensive overview: Bender (2013)

grace → graceful  
graceful → gracefully  
grace → disgrace  
disgrace → disgraceful  
disgraceful → disgracefully  
friend → unfriend  
Obama → Obamacare

# Morphology

Extensive overview: Bender (2013)

uygarlaştıramadıklarımızdanmışsınızcasına  
“(behaving) as if you are among those whom we could not civilize”

# Reflection

What (natural) languages do you know? What are some examples of the morphology in those languages?

# Aperitif: Finite-State Automata

A finite-state automaton (plural “automata”) consists of :

- ▶ a finite alphabet of input symbols,  $\Sigma$
- ▶ a finite set of states,  $Q$
- ▶ a start state,  $q_0 \in Q$
- ▶ a set of final states,  $F \subseteq Q$
- ▶ a transition function that maps a state and a symbol (or an empty string, denoted  $\varepsilon$ ) to a set of states,  
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$$

We visualize an FSA with a state diagram.

# Aperitif: Finite-State Automata

A finite-state automaton (plural “automata”) consists of (toy example in blue):

- ▶ a finite alphabet of input symbols,  $\Sigma$   $\Sigma = \{a, b\}$
- ▶ a finite set of states,  $Q$   $Q = \{q_0, q_1\}$
- ▶ a start state,  $q_0 \in Q$   $q_0$
- ▶ a set of final states,  $F \subseteq Q$   $F = \{q_1\}$
- ▶ a transition function that maps a state and a symbol (or an empty string, denoted  $\varepsilon$ ) to a set of states,

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$$
$$\delta = \left\{ \begin{array}{ll} (q_0, a) & \rightarrow \{q_0\}, \\ (q_0, b) & \rightarrow \{q_1\}, \\ (q_1, a) & \rightarrow \emptyset, \\ (q_1, b) & \rightarrow \{q_1\} \end{array} \right\}$$

We visualize an FSA with a state diagram.



# State Diagram for our Toy Example FSA

►  $\Sigma = \{a, b\}$

►  $Q = \{q_0, q_1\}$

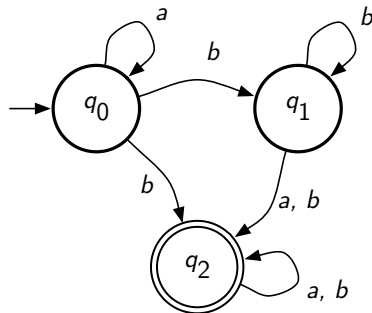
►  $F = \{q_1\}$

►  $\delta = \left\{ \begin{array}{ll} (q_0, a) & \rightarrow \{q_0\}, \\ (q_0, b) & \rightarrow \{q_1\}, \\ (q_1, a) & \rightarrow \emptyset, \\ (q_1, b) & \rightarrow \{q_1\} \end{array} \right\}$

# FSAs and their Languages

- ▶ A language is a set of strings; for FSA  $\mathcal{F}$  we denote by  $L(\mathcal{F})$  the set of strings it accepts.
- ▶ Regular languages: the set of languages recognizable by FSAs.
- ▶ A *path* through the FSA  $\mathcal{F}$  serves as a proof that the path's string is in  $L(\mathcal{F})$ .
- ▶ An FSA is *deterministic* (a “DFA”) if there is exactly one path per string in  $L(\mathcal{F})$ .
- ▶ Given DFA  $\mathcal{F}$  and a string of length  $n$ , we can check membership in  $L(\mathcal{F})$  in  $O(n)$  time and  $O(1)$  space.

# Nondeterministic FSA

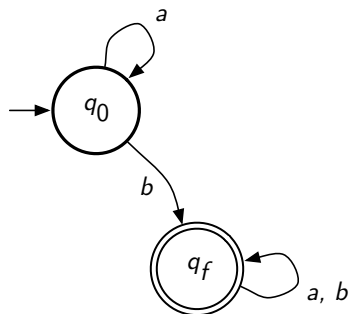
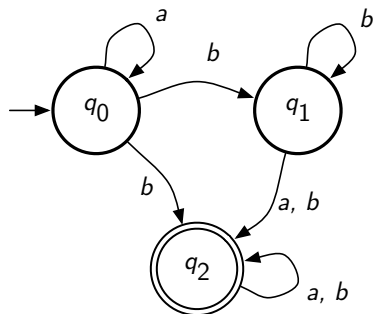


Accepts any string of *as* and *bs* that includes at least one *b*.

# Some Theoretical Properties of Regular Languages

- ▶ Closed under intersection, union, subtraction, concatenation, negation, Kleene closure, reversal, and more operations.
- ▶ There things they cannot do! E.g., counting.  $a^n b^n$  is not a regular language. The pumping lemma is a formal tool used to prove that a language is not regular.
- ▶ Any nondeterministic FSA can be mechanically transformed into a deterministic one with the same language, but the number of states may explode.

# NDFA and DFA with the same language.



# Reflection

Consider the case where  $\Sigma$  is the set of English *words* (not characters).

Do you think English is a regular language? How would you prove it one way or the other?

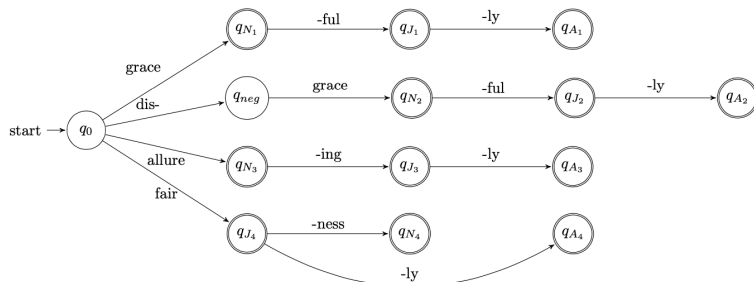
# How can we use it?

“Vocabulary machine”: an FSA whose language includes all (and only) the words in a language (e.g., English). ( $\Sigma$  is the set of characters used in the language.)

Advantage over a simple brute-force list: encode rules that let us generate new words (e.g., *Clintonian*, *Trumpism*, *coronafuckingvirus*).

# Example

Eisenstein (2019) figure 9.2 (p. 187)





# Adding Weights

A powerful generalization is the **weighted** FSA (WFSA), which augments every path with a score. A WFSA consists of:

- ▶ a finite alphabet of input symbols,  $\Sigma$
- ▶ a finite set of states,  $Q$
- ▶ an initial weight function,  $\lambda : Q \rightarrow \mathbb{R}$
- ▶ a final weight function,  $\rho : Q \rightarrow \mathbb{R}$
- ▶ a transition function that weights maps a state pair and a symbol (or  $\varepsilon$ ),  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times Q \rightarrow \mathbb{R}$

# Reflection

Can you show how an unweighted FSA is a special case of a WFSA? Hint: imagine that there are only two values that  $\lambda$ ,  $\rho$ , and  $\delta$  can map to, 0 and  $-\infty$ .

## Scoring a Path

Consider a path of  $n$  transitions,  $q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \cdots q_{n-1} \xrightarrow{x_n} q_n$ .

The score of the path is given by

$$\lambda(q_0) + \left( \sum_{i=1}^n \delta(q_{i-1}, x_i, q_i) \right) + \rho(q_n)$$

You can think of weights as “costs” and imagine trying to find the minimum-cost path through a WFSA for a given string  $x$ .

# Reflection

Can you think of a good use for “costs” or scores associated with the words in our vocabulary machine’s language?

## The Main Dish

# Weighted Finite-State Transducers

WFSTs encode weighted *relations* between strings. They consist of:

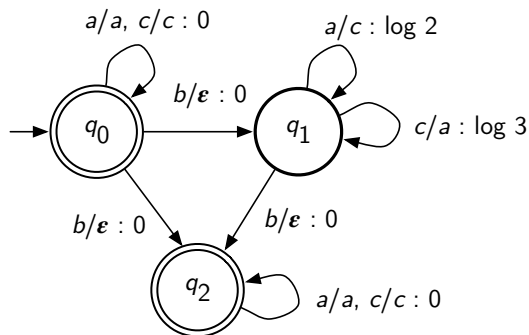
- ▶ a finite alphabet of input symbols,  $\Sigma$
- ▶ **a finite alphabet of output symbols,  $\Omega$**
- ▶ a finite set of states,  $Q$
- ▶ an initial weight function,  $\lambda : Q \rightarrow \mathbb{R}$
- ▶ a final weight function,  $\rho : Q \rightarrow \mathbb{R}$
- ▶ a transition function that weights maps a state pair and a **pair of symbols** (or  $\varepsilon$ ),  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Omega \cup \{\varepsilon\}) \times Q \rightarrow \mathbb{R}$

# Reflection

WFSTs generalize unweighted FSTs, WFSAs, and unweighted FSAs!

Can you sketch out a way to convert any of those into a WFST?

## Example of a WFST



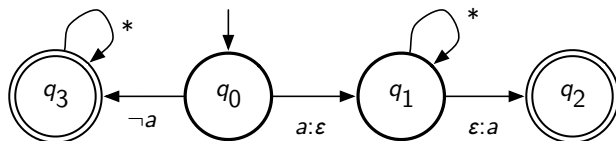


# Properties of WFSTs

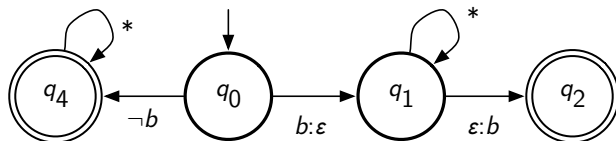
- ▶ If you strip away either the inputs or the outputs, you get a WFSA and the language is regular.
- ▶ Most important property: WFSTs are closed under composition. Consider the unweighted case.
  - ▶ Let  $\mathcal{F}$  be an FST encoding pairs  $F \subseteq \Sigma^* \times \Gamma^*$ .
  - ▶ Let  $\mathcal{G}$  be an FST encoding pairs  $G \subseteq \Gamma^* \times \Omega^*$ .
  - ▶ Then  $\mathcal{G} \circ \mathcal{F}$  denotes  $\{(x, z) \mid \exists y \in \Gamma^*, (x, y) \in F \wedge (y, z) \in G\}$ .
  - ▶ There is an FST that encodes  $\mathcal{G} \circ \mathcal{F}$ .

# Illustrating Composition

$F$  maps  $a\alpha$  to  $\alpha a$  and  $(\neg a)a$  to itself:

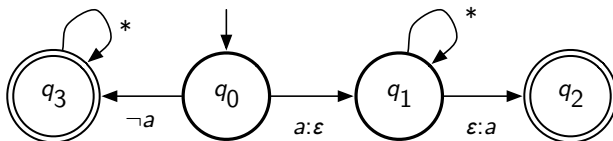


$G$  maps  $b\alpha$  to  $\alpha b$  and  $(\neg b)a$  to itself:

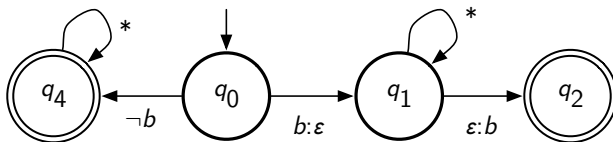


# Illustrating Composition

$F$  maps  $a\alpha$  to  $\alpha a$  and  $(\neg a)a$  to itself:



$G$  maps  $b\alpha$  to  $\alpha b$  and  $(\neg b)a$  to itself:



We can implement both  $G \circ F$  and  $F \circ G$  by applying FST composition, and both will be FSTs.

# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>				
<i>bad</i>				
<i>def</i>				

# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>	<i>bca</i>			
<i>bad</i>				
<i>def</i>				

# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>	<i>bca</i>	<i>cab</i>		
<i>bad</i>				
<i>def</i>				

# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>	<i>bca</i>	<i>cab</i>	<i>abc</i>	
<i>bad</i>				
<i>def</i>				

# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>	<i>bca</i>	<i>cab</i>	<i>abc</i>	<i>bca</i>
<i>bad</i>				
<i>def</i>				



# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>	<i>bca</i>	<i>cab</i>	<i>abc</i>	<i>bca</i>
<i>bad</i>	<i>bad</i>	<i>adb</i>	<i>adb</i>	<i>dba</i>
<i>def</i>				

# Illustrating Composition

input	output of ...			
	F	$G \circ F$	G	$F \circ G$
<i>abc</i>	<i>bca</i>	<i>cab</i>	<i>abc</i>	<i>bca</i>
<i>bad</i>	<i>bad</i>	<i>adb</i>	<i>adb</i>	<i>dba</i>
<i>def</i>	<i>def</i>	<i>def</i>	<i>def</i>	<i>def</i>

# (W)FST as a Declarative System

For convenience, we talk about an “input” and an “output” string, but the same model can also be thought of as:

- ▶ Mapping output strings to input strings
- ▶ Recognizing pairs of strings
- ▶ Generating pairs of strings

You should think of FSTs as primarily a *declarative* framework (not a procedure).

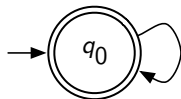
Avoid this confusion: FSTs are *not* functions from inputs to outputs; an input string can pair with more than one output string (and vice versa).

# Putting WFSTs to Work

Levenshtein edit distance: what's the minimum number of single-character deletions, insertions, or substitutions to change  $x$  into  $x'$ ?

You only need one state! The classic dynamic programming algorithm emerges when you apply conventional shortest-path algorithms.

# Levenshtein Distance WFST, $\Sigma = \{a, b\}$



$a/a : 0$

$b/b : 0$

no change, no cost

$a/\epsilon : 1$

$b/\epsilon : 1$

deletions cost 1

$$\lambda(q_0) = \rho(q_0) = 0$$

$\epsilon/a : 1$

$\epsilon/b : 1$

insertions cost 1

$a/b : 1$

$b/a : 1$

substitutions cost 1

# Putting WFSTs to Work

Problem: surface variation in words hides semantic (near) equivalence. E.g., the subtle differences among  $\{\textit{invite}, \textit{invited}, \textit{inviting}, \textit{invites}\}$  do not matter for many applications.

# Putting WFSTs to Work

Problem: surface variation in words hides semantic (near) equivalence. E.g., the subtle differences among  $\{\textit{invite}, \textit{invited}, \textit{inviting}, \textit{invites}\}$  do not matter for many applications.

Porter (1980) stemmer: an algorithm that strips suffixes from English words (without “knowing” any words) according to a set of rules, such as:

$$\begin{aligned} -sses &\rightarrow -ss \\ -ies &\rightarrow -i \\ -ss &\rightarrow -ss \text{ OR } -s \rightarrow \varepsilon \end{aligned}$$

# Putting WFSTs to Work

Problem: surface variation in words hides semantic (near) equivalence. E.g., the subtle differences among  $\{\textit{invite}, \textit{invited}, \textit{inviting}, \textit{invites}\}$  do not matter for many applications.

Stemming lets a system abstract away from words a bit, so that (e.g.) a search engine query for *parties where cats are invited* will match documents with *invite a cat to a party* as well.



# Putting WFSTs to Work

Problem: surface variation in words hides semantic (near) equivalence. E.g., the subtle differences among *{invite, invited, inviting, invites}* do not matter for many applications.

Stemming lets a system abstract away from words a bit, so that (e.g.) a search engine query for *parties where cats are invited* will match documents with *invite a cat to a party* as well.

(Today, people use data-driven methods like byte-pair encoding (Sennrich et al., 2016) to segment words into pieces, sometimes called “wordpieces.”)

# What about today?

Finite-state transducers (sometimes weighted, sometimes not) are arguably the best way to encode the morphological systems of many languages.

Goal: map between words we see in text (“surface” forms) and morphological analyses into a lemma or base/root form of the word plus “features.”

# What about today?

Finite-state transducers (sometimes weighted, sometimes not) are arguably the best way to encode the morphological systems of many languages.

Goal: map between words we see in text (“surface” forms) and morphological analyses into a lemma or base/root form of the word plus “features.”

Example from Spanish (surface  $\leftrightarrow$  analysis):

<i>canto</i>	$\leftrightarrow$	<i>cantar</i> +Verb+PresentIndicative+1stPerson+Singular
<i>como</i>	$\leftrightarrow$	<i>comer</i> +Verb+PresentIndicative+1stPerson+Singular
<i>comes</i>	$\leftrightarrow$	<i>comer</i> +Verb+PresentIndicative+2ndPerson+Singular

# What about today?

Finite-state transducers (sometimes weighted, sometimes not) are arguably the best way to encode the morphological systems of many languages.

Goal: map between words we see in text (“surface” forms) and morphological analyses into a lemma or base/root form of the word plus “features.”

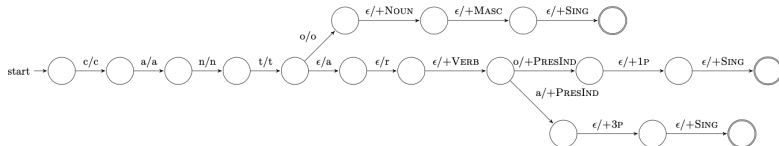
Example from Spanish (surface  $\leftrightarrow$  analysis):

<i>canto</i>	$\leftrightarrow$	<i>cantar</i> +Verb+PresentIndicative+1stPerson+Singular
<i>como</i>	$\leftrightarrow$	<i>comer</i> +Verb+PresentIndicative+1stPerson+Singular
<i>comes</i>	$\leftrightarrow$	<i>comer</i> +Verb+PresentIndicative+2ndPerson+Singular

If you use a (W)FST, you can invert input and output and use the same model for analysis and generation!

# Example

Eisenstein (2019) figure 9.7 (p. 195)



# Designing an FST for Morphology: Notes

The challenge is to avoid both under- and over-generation. E.g., we want *feet/foot+Plural* and *beets/beet+Plural*, but not *foots/foot+Plural* or *beet/boot+Plural*!

# Designing an FST for Morphology: Notes

Because FSTs encode relations, we can elegantly handle optionality (e.g., *colours/color*+Noun+Plural and *colors/color*+Noun+Plural) and ambiguity (e.g., *bears/bear*+Noun+Plural and *bears/bear*+Verb+Present+3rdPerson+Singular).

# Designing an FST for Morphology: Notes

Because of closure under composition, union, concatenation, etc., you can build separate modules for different morphology rules, or parts of the vocabulary.



# Designing an FST for Morphology: Notes

Usually some parts are “lexicons” or FSTs that encode sets of words to which the same rules are applied (e.g., “-er verbs” in French).

# Designing an FST for Morphology: Notes

It's hard to avoid the writing system (orthography) of a language; some of your rules will probably be more about writing conventions than the language as it is spoken. E.g., English past tense adds *-ed* to a verb's base form, but in the writing system we don't do this if the word ends in silent *e*: *bake* becomes *baked*, not *bakeed*.

# Linguistic Note

Examples curated by Fokkens (2009)

Just a few of the phenomena that are less trivial to handle with FSTs:

# Linguistic Note

Examples curated by Fokkens (2009)

Just a few of the phenomena that are less trivial to handle with FSTs:

- ▶ Transfixation, e.g., Maltese has the root *ktb*, from which are formed words like *kiteb* (“he wrote”), *kitbu* (“they wrote”), *miktub* (“written”), *ktieb* (“book”), *kotba* (“books”), and more (Crysmann, 2006)

# Linguistic Note

Examples curated by Fokkens (2009)

Just a few of the phenomena that are less trivial to handle with FSTs:

- ▶ Transfixation, e.g., Maltese has the root *ktb*, from which are formed words like *kiteb* (“he wrote”), *kitbu* (“they wrote”), *miktub* (“written”), *ktieb* (“book”), *kotba* (“books”), and more (Crysmann, 2006)
- ▶ Subtraction, e.g., Koasati has singular *pitaf-fi-n* and plural *pit-li-n* (“to slice up in the middle”) and *acokcana:-kaln* singular and *acokcan-ka-n* plural (“to quarrel with someone”) (Sproat, 1992)

# Linguistic Note

Examples curated by Fokkens (2009)

Just a few of the phenomena that are less trivial to handle with FSTs:

- ▶ Transfixation, e.g., Maltese has the root *ktb*, from which are formed words like *kiteb* (“he wrote”), *kitbu* (“they wrote”), *miktub* (“written”), *ktieb* (“book”), *kotba* (“books”), and more (Crysmann, 2006)
- ▶ Subtraction, e.g., Koasati has singular *pita~~f~~-fi-n* and plural *pit-li-n* (“to slice up in the middle”) and *acokcana:-ka~~n~~* singular and *acokcan-ka-n* plural (“to quarrel with someone”) (Sproat, 1992)
- ▶ Reduplication, e.g., Indonesian has *orang* (“man”) and *orang orang* (“men”) (Crysmann, 2006)

# Notable NLP Tools

- ▶ Foma: <https://fomafst.github.io> (designed for manual programming of FSTs; Hulden, 2009; see also Beesley and Karttunen, 2003)
- ▶ OpenFST: <http://www.openfst.org/> (designed for WFST operations)
- ▶ EpiTran: grapheme-to-phoneme conversion for lots of languages (Mortensen et al., 2018)

# Reflection

The Yiddish language is conventionally written in a variant of the Hebrew alphabet, but it can also be transliterated into the Latin alphabet we use for English. The former is written right-to-left, the latter left-to-right.

Assuming we keep characters in the order they appear on a printed page, could we use a (W)FST to map Yiddish words in either alphabet into the other?



# Cautionary Note

A computational model of a natural language's morphology probably encodes:

- ▶ The rules as known to one particular community of speakers (often a privileged one)
- ▶ Orthographic conventions of one such community

But a language (and writing) vary a lot across communities of its users.

Ask: who was/is this system built for?

# Digestif: Remarks

Current NLP research is not very focused on finite-state methods, but they are worth knowing about because:

- ▶ For some language problems, you can manually program a nearly perfect solution if you choose the right formalism and work hard for good coverage.
- ▶ Morphology is a huge challenge in some languages; the number of possible words can be large, and many won't appear in text collections.
- ▶ Later, you'll hear me say that some methods are “uninterpretable” and “not formally understood.” WFSTs are the opposite of that!

# References I

- Kenneth R. Beesley and Lauri Karttunen. *Finite-State Morphology: Xerox Tools and Techniques*. CSLI, Stanford, 2003.
- Emily M. Bender. *Linguistic Fundamentals for Natural Language Processing: 100 Essentials from Morphology and Syntax*. Morgan and Claypool, 2013.
- Berthold Crysmann. Foundations of language science and technology: Morphology, 2006. URL <http://www.coli.uni-saarland.de/~hansu/courses/FLST05/schedule.html>.
- Jacob Eisenstein. *Introduction to Natural Language Processing*. MIT Press, 2019.
- Antske Fokkens. Introduction to morphology, 2009. URL <https://carrerainglesuce.files.wordpress.com/2019/05/morphology.pdf>.
- Mans Hulden. Foma: a finite-state compiler and library. In *Proc. of EACL*, 2009.
- David R. Mortensen, Siddharth Dalmia, and Patrick Littell. Epitran: Precision G2P for many languages. In *Proc. of LREC*, 2018.
- M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. of ACL*, 2016.
- Richard Sproat. *Morphology and Computation*. MIT Press, 1992.