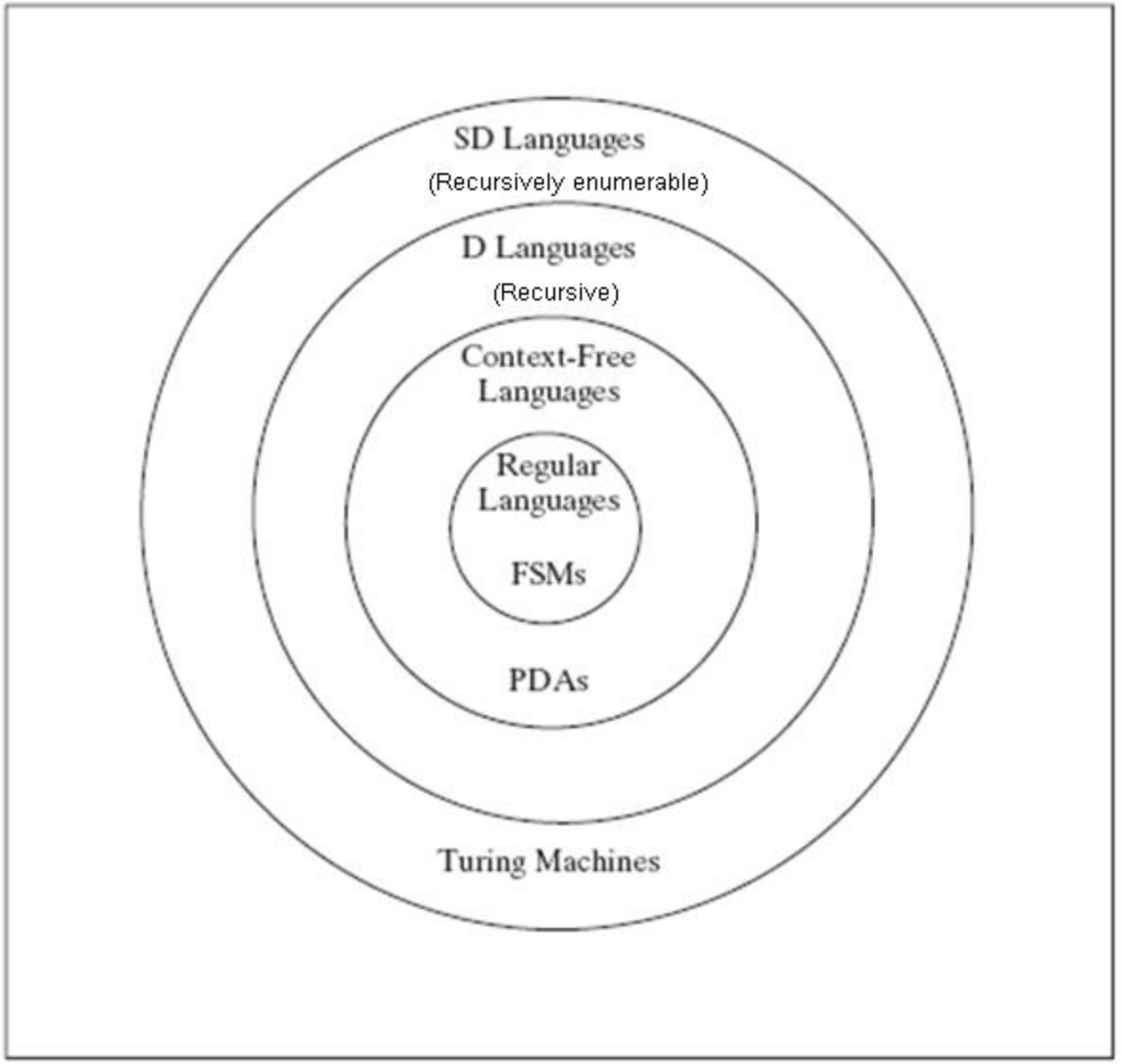
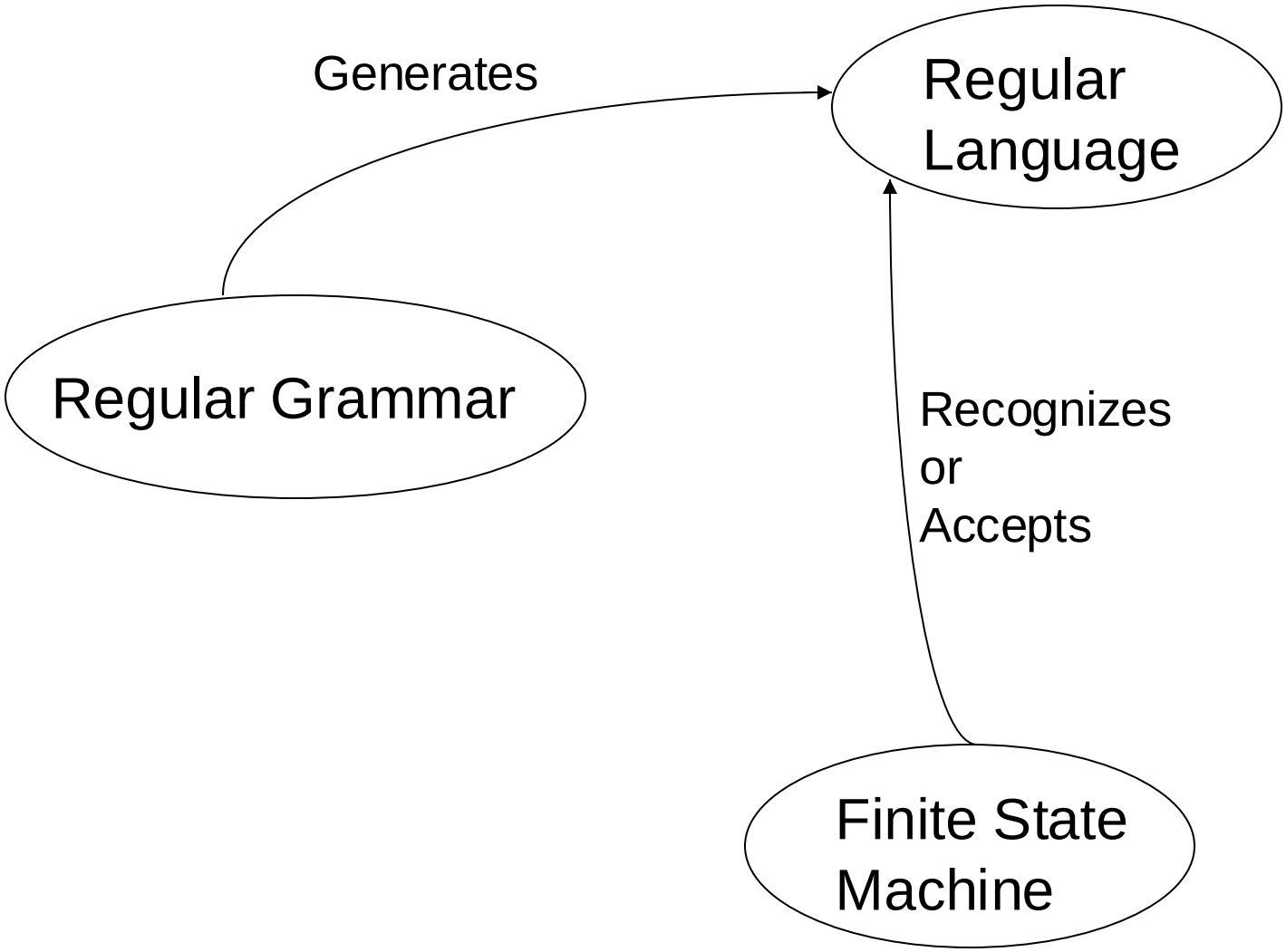
Finite State Machines

Chapter 5

Languages and Machines



Regular Languages



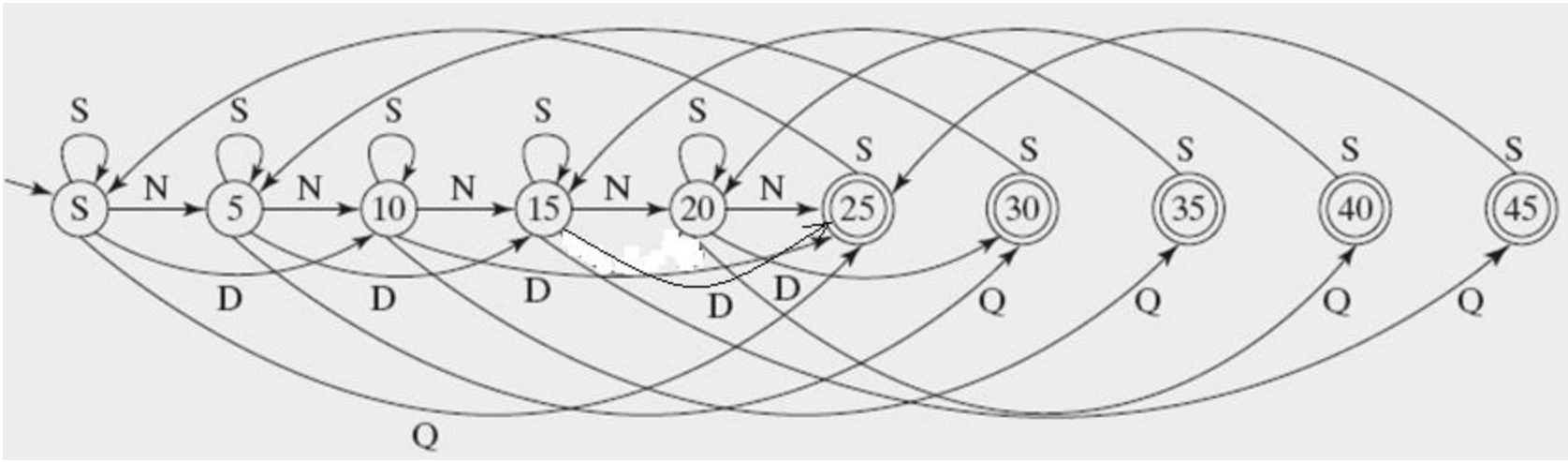
Finite State Machines

An example FSM: a device to solve a problem (dispense drinks);
or a device to recognize a language (the “enough money” language that consists of the set of strings, such as NDD, that drive the machine to an accepting state in which a drink can be dispensed)

N: nickle D: dime Q: quarter S: soda R: return
Accepts up to $.45; $.25 per drink

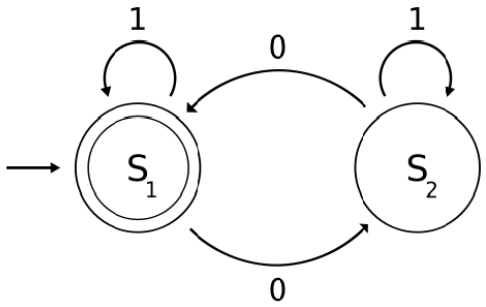
After a finite sequence of inputs, the controller will be in either:
A dispensing state (enough money);
or a nondispensing state (no enough money)

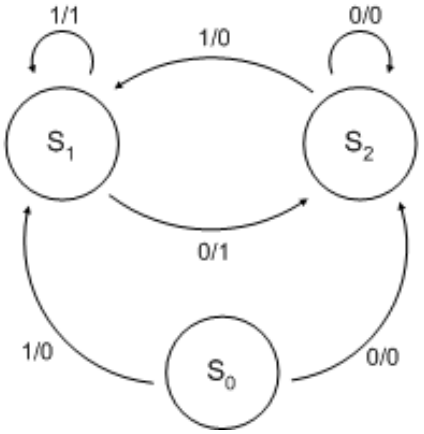
Error!



Representations

- **State diagrams** can be used to graphically represent finite state machines.
  - describe behavior of systems
  - introduced by Taylor Booth in his 1967 book "Sequential Machines and Automata Theory "





- Another representation is the *state transition table*

Input / State	1	0
S_1	S_1	S_2
S_2	S_2	S_1

FSM

• A computational device whose input is a string, and whose output is one of the two values: *Accept* and *Reject*

• Also called FSA (finite state automata)

• Input string *w* is fed to *M* (an FSM) one symbol at a time, left to right
• Each time it receives a symbol, *M* considers its current state and the new symbol and chooses a next state
• One or more states maybe marked as accepting states
• Other states are rejecting states
• If *M* runs out of input and is in an accepting state, it accepts

• Begin defining the class of FSMs whose behavior is deterministic.
    • move is determined by current state and the next input character

Definition of a DFSM

$M = (K, \Sigma, \delta, s, A)$, where:

K is a finite set of states
Σ is an alphabet
$s \in K$ is the initial state
$A \subseteq K$ is the set of accepting states, and
δ is the transition function from $(K \times \Sigma)$ to K

Configurations of DFSMs

To describe the execution of machine *M* on input *w*, we need a few definitions.

A ***configuration*** of a DFSM *M* is an element of:

$$K \times \Sigma^*$$

- It captures the two things that decide *M*'s future behavior:
  - current state
  - the remaining, unprocessed input
- It provides a "snapshot" of the system at a particular execution/processing step.

The ***initial configuration*** is (*s*, *w*)

The Yields Relations

During execution, when a state transition occurs, the system moves from one configuration to another.

The *yields-in-one-step* relation |-

(q, w) |- (q', w') iff

- $w = a\ w'$ for some symbol $a \in \Sigma$, and
- $\delta\ (q, a) = q'$

After the transition, state q -> q', remaining string w -> w'

The relation *yields* |- * is the reflexive, transitive closure of |-
If C_i |- * C_j, iff M can go from C_i to C_j in zero (due to "reflexive") or more (due to "transitive") steps.

Execution Path

The following definitions and concepts simply the ones from the textbook. They are applicable to all machines we talk about in this course.

An execution ***path*** by machine *M* on input *w* is a **maximal** sequence of configurations C_0, C_1, C_2 … such that:

- C_0 is an initial configuration
- C_0 |- C_1 |- C_2 |- …

- In other words, a path is just a sequence of execution/processing steps (described by configurations) from the start going as far as possible (maximal). As long as a state transition is defined, go for it.
- An execution path ***accepts*** *w* if it ends in an accepting configuration, where a set of predefined accepting conditions are met.
  - Accepting conditions vary from machine to machine. e.g., FSM, PDA, TM are different types of machines by definition with different predefined accepting conditions.
- An execution path **rejects** *w* if it ends in a non-accepting configuration
  - When it ends, it either accepts (say yes) or rejects (say no). If not yes, then no.
- When would an execution path **end** (halt, terminate)?
  - When it has no where to go, i.e., no transition is defined
  - A path may not end (infinite path), in which case it cannot accept or reject
- Summary (all machines) of halting behavior for an execution path P
  - P always ends: DFSM, NDFSM without ε-transitions, DPDA, NDPDA without ε-transitions
  - P can be infinite: NDFSM with ε-transitions, NDPDA with ε-transitions, TM, NDTM

Path vs Machine

- For deterministic machines (where a transition function is defined), there's only one execution path. The accepting/rejecting/halting behavior of machine *M* solely depends on the accepting/rejecting/halting behavior of the path P.
  - If P halts and accepts, M halts and accepts.
  - If P halts and rejects, M halts and rejects.
  - If P does not halt, M does not halt.

- For non-deterministic machines (where a transition relation is defined), there can be multiple execution paths. The accepting/rejecting/halting behavior of machine *M* depends on the collective accepting/rejecting/halting behavior of all paths.
  - If one path halts and accepts, M halts and accepts.
  - If all paths halt and reject, M halts and rejects.
  - Otherwise (i.e., no path accepts, and not all paths reject), M does not halt.

- Recall relation generalizes function, so non-deterministic machines generalize deterministic machines.

Again, these concepts apply to all machines we talk about in the course. We will see how they apply in following lectures.

Accepting

- A DFSM *M* ***accepts*** a string *w* iff the path accepts it.
  - **The** path, because there is only one.
- Predefined accepting conditions: (1) all symbols in w have been processed/consumed. (2) in an accepting state

  More formally, accepting configuration for DFSM:

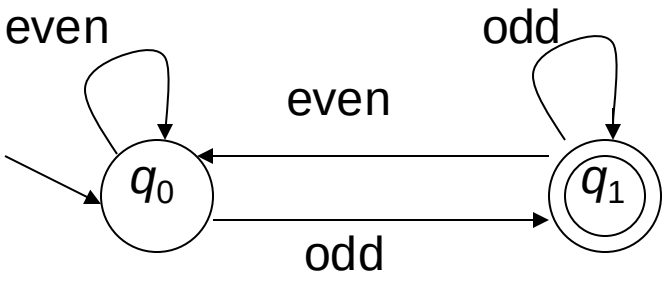
  (q, ε) where $q \in A$

- A DFSM *M* ***rejects*** a string *w* iff the path rejects it.
- The ***language accepted by*** *M*, denoted *L*(*M*), is the set of all strings accepted by *M*.

Theorem: Every DFSM *M*, on input *w*, halts in at most |*w*| steps.

Accepting Example

An FSM to accept odd integers:



On input `235`, the configurations are:

$(q_0, 235)$ |- $(q_0, 35)$

|-

|- (q_1, ε) which is an accepting configuration

If M is a DFSM and $\varepsilon \in L(M)$, what simple property must be true of M?

- The start state of M must be an accepting state

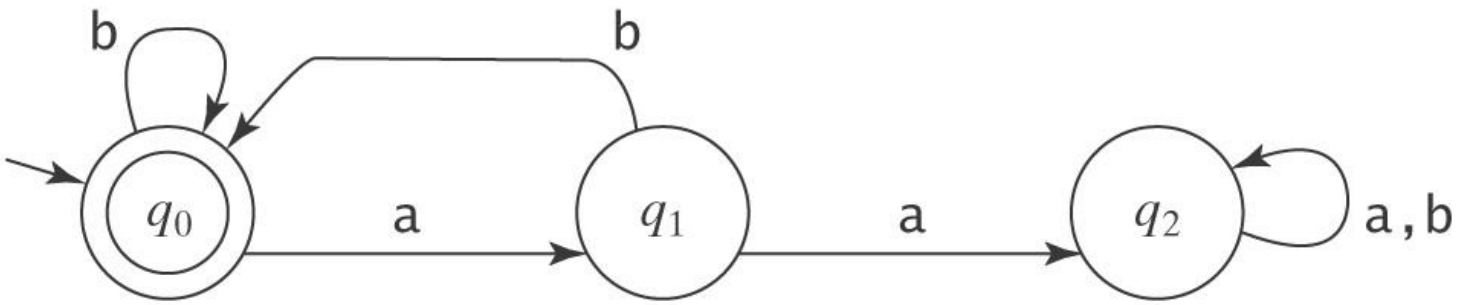
Regular Languages

A language is ***regular*** iff it is accepted by some FSM.

A Very Simple Example

$L = \{w \in \{\texttt{a}, \texttt{b}\}^* :$

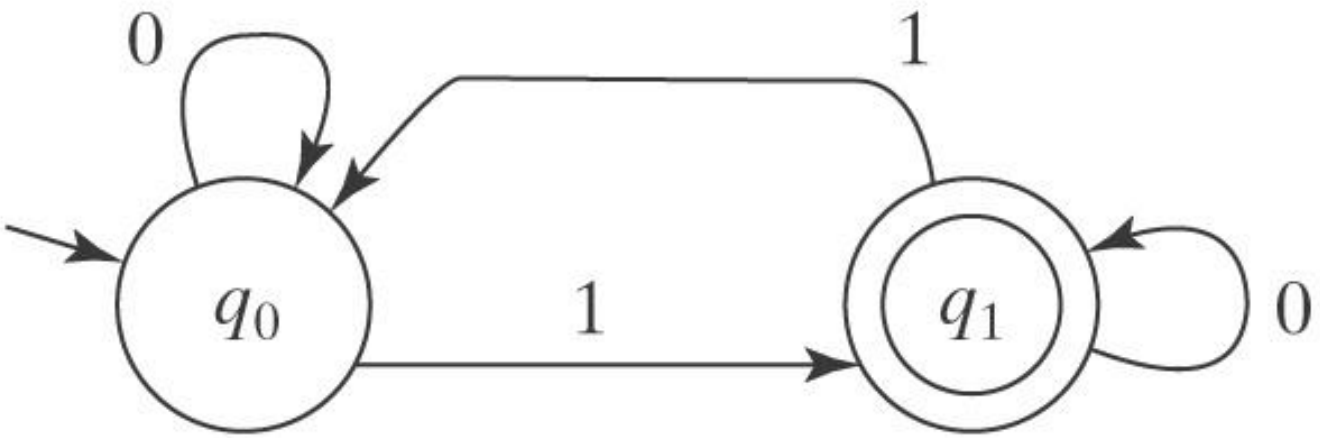
every a is immediately followed by a b}.



Parity Checking

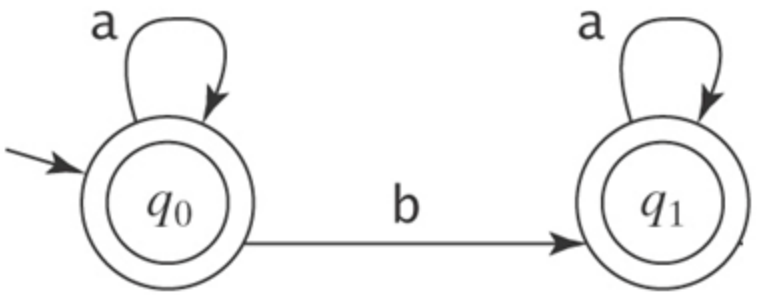
$L = \{w \in \{0, 1\}^* : w \text{ has odd parity}\}$.

A binary string has odd parity iff the number of 1's is odd



No More Than One b

$L = \{w \in \{\texttt{a}, \texttt{b}\}^* : w \text{ contains no more than one } \texttt{b}\}$.

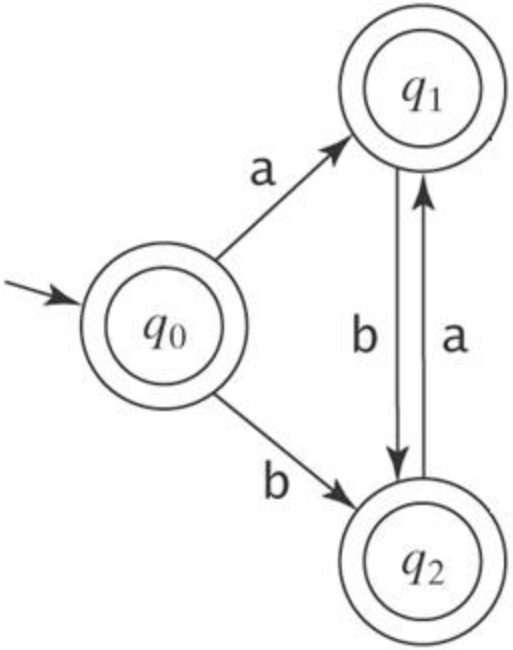


- *Some rejecting states are ignored for clarity*
  - A full state diagram would allow the path to exhaust all input symbols, not ending prematurely. But it can be messy.

Checking Consecutive Characters

$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* :$

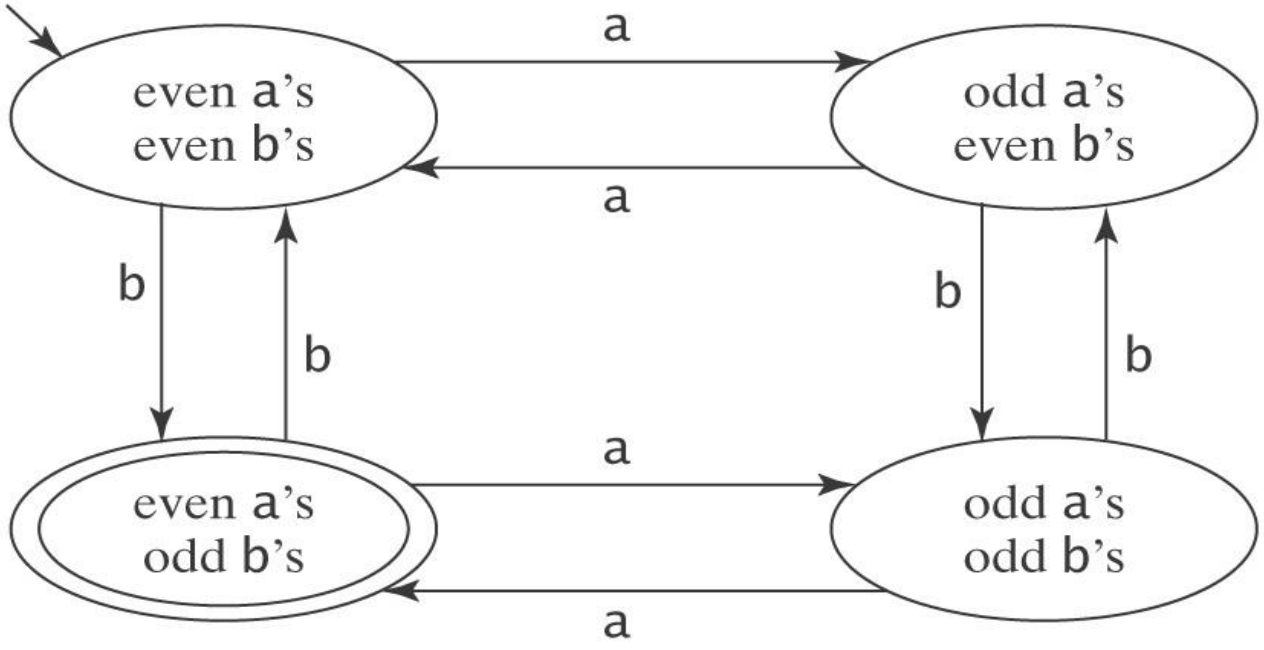
no two consecutive characters are the same}.



Programming FSMs

L is infinite but *M* has a finite number of states, strings must cluster: Cluster strings that share a "future".

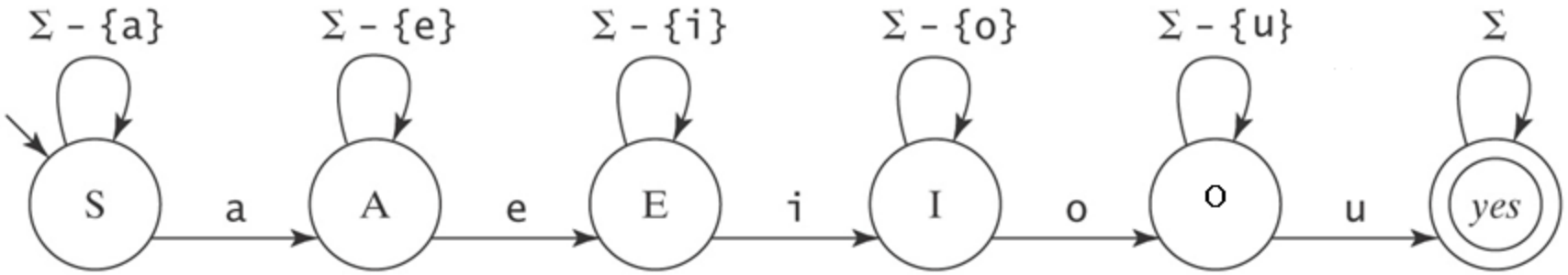
Let *L* = {*w* ∈ {a, b}* : *w* contains an even number of a's and an odd number of b's}



Vowels in Alphabetical Order

L = {$w \in$ {a - z}* : can find five vowels, a, e, i, o, and u, that occur in w in alphabetical order}.

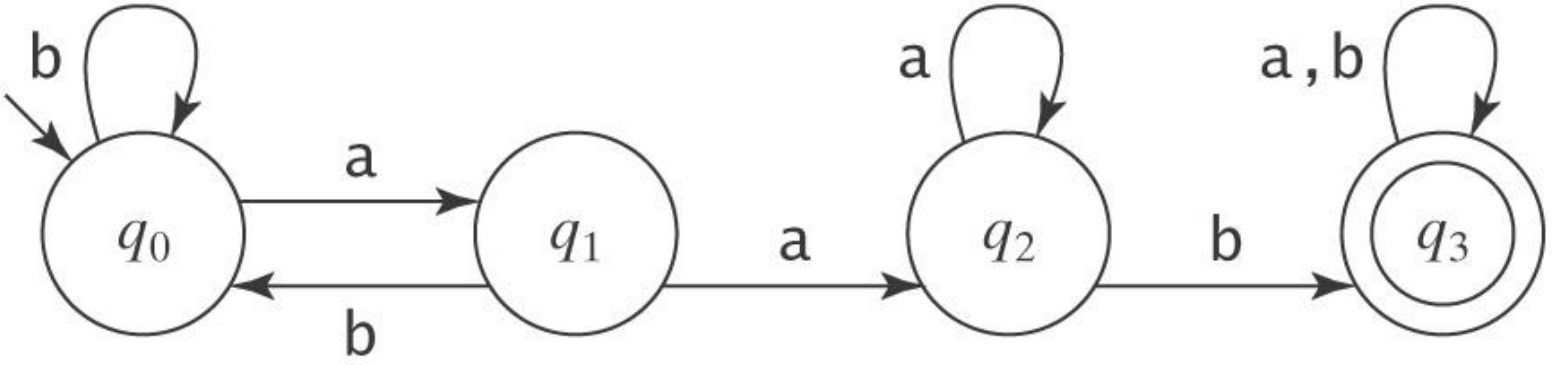
abstemious, facetious, sacrilegious



Programming FSMs

$L = \{w \in \{\texttt{a}, \texttt{b}\}^* : w$ does not contain the substring $\texttt{aab}\}$.

Start with a machine for $\neg L$:

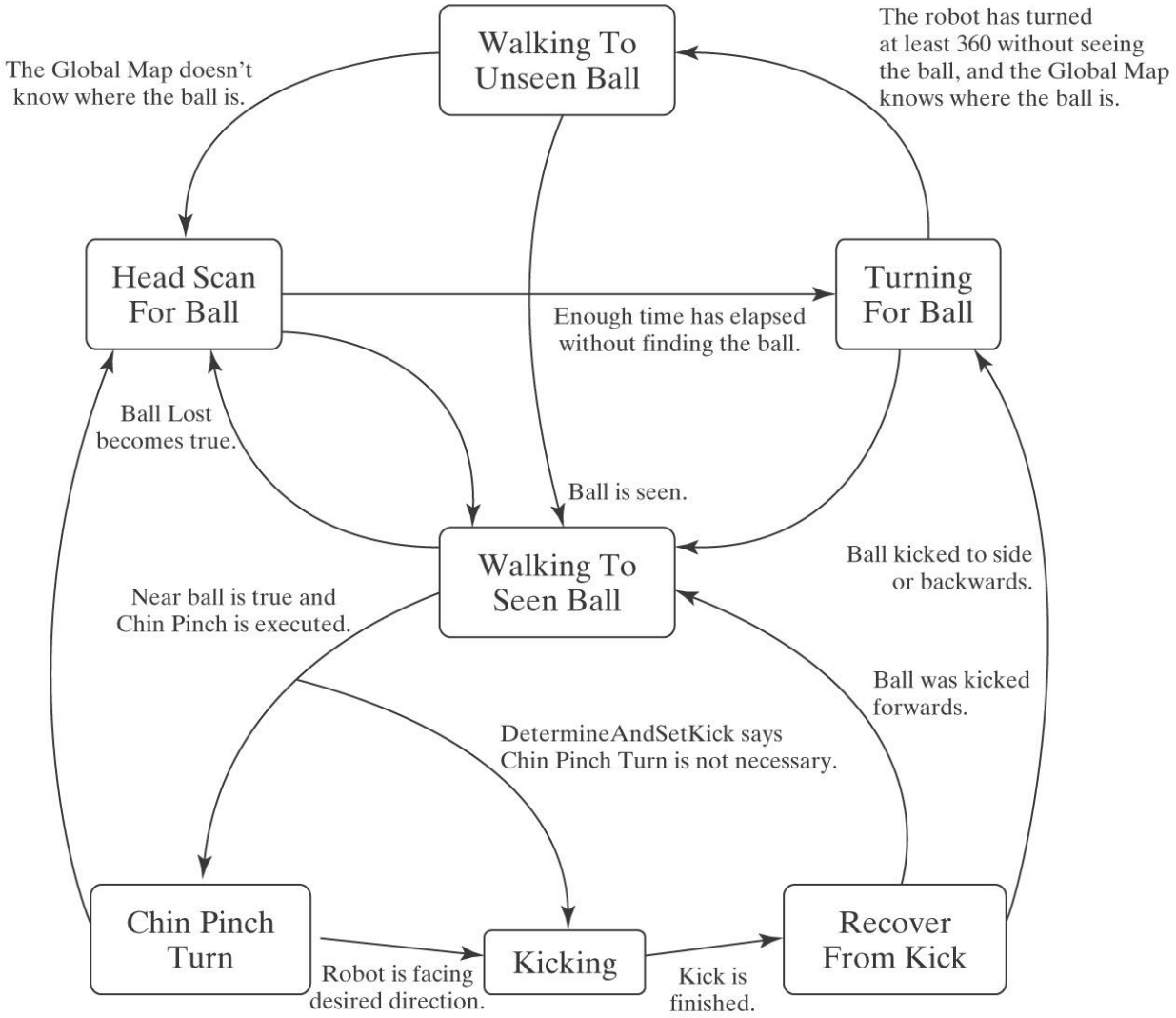


How to convert it to a machine for L?

Caution: This example shows a full state diagram where all possible states and transitions are specified. In other examples, if we want to use the trick, need to build a full state diagram first.

Controlling a Soccer-Playing Robot

A Simple Controller



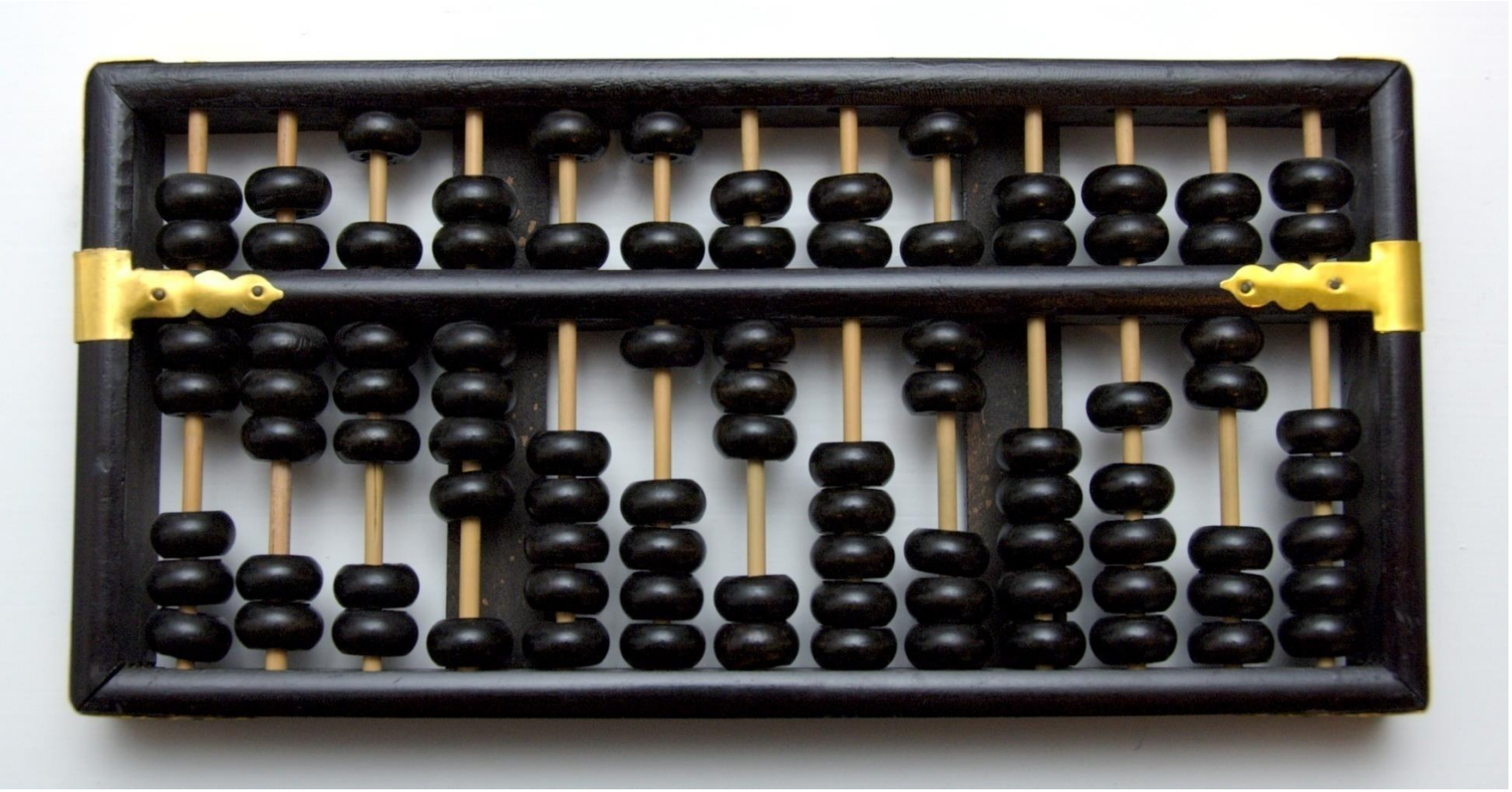
FSMs Predate Computers

The Prague Orloj, originally built in 1410.

The Jacquard Loom

Invented in 1801.

The Abacus



The Missing Letter Language

Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$.

Let $L_{Missing}$ =
$\{w$: there is a symbol $a_i \in \Sigma$ not appearing in $w\}$.

Try to make a DFSM for $L_{Missing}$

- Doable, but complicated. Consider the number of accepting states
  - all missing (1)
  - 3 missing (4)
  - 2 missing (6)
  - 1 missing (4)

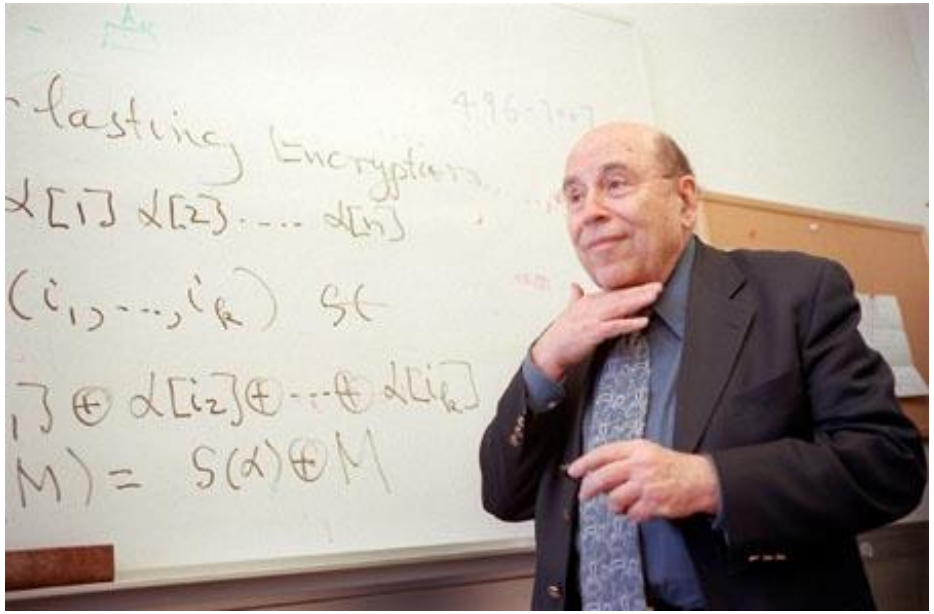
Nondeterministic FSM

- In the theory of computation, a **nondeterministic finite state machine** (NDFSM) is a finite state machine where for each pair of state and input symbol there may be several possible next states.
  - This distinguishes it from the deterministic finite state machine (DFSM), where the next possible state is uniquely determined.
  - Although DFSM and NDFSM have distinct definitions, it may be shown in the formal theory that they are equivalent, in that, for any given NDFSM, one may construct an equivalent DFSM, and vice-versa
  - Both types of automata recognize only regular languages.
  - Nondeterministic machines are a key concept in computational complexity theory, particularly with the description of complexity classes *P* and *NP*.
- Introduced by Michael O. Rabin and Dana Scott in 1959
  - also showed equivalence to deterministic automata
  - co-winners of Turing award, citation:
    - *For their joint paper "Finite Automata and Their Decision Problem," which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their (Scott & Rabin) classic paper has been a continuous source of inspiration for subsequent work in this field.*

Nondeterministic Machines

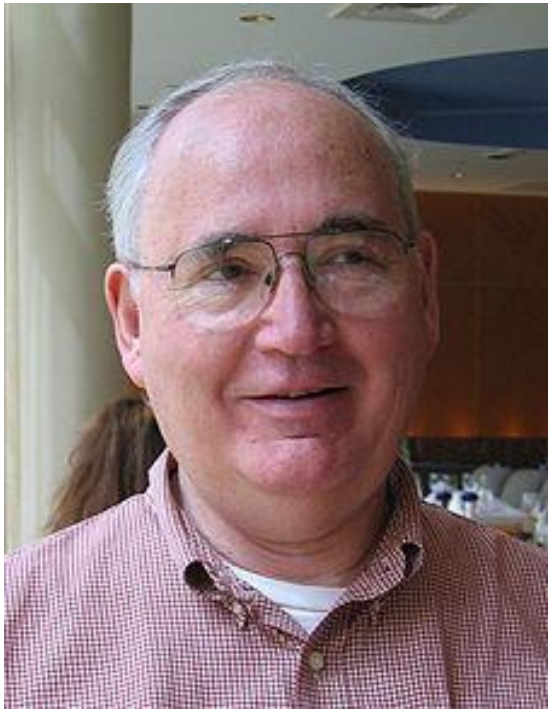
Michael O. Rabin (1931 -)

- son of a rabbi, PhD Princeton
- currently Harvard
- contributed in Cryptograph



Dana Stewart Scott (1932 -)

- PhD Princeton (Alonzo Church)
- retired from Berkley



Definition of an NDFSM

$M = (K, \Sigma, \Delta, s, A)$, where:

K is a finite set of states

Σ is an alphabet

$s \in K$ is the initial state

$A \subseteq K$ is the set of accepting states, and

Δ is the transition relation. It is a finite subset of

$$(K \times (\Sigma \cup \{\varepsilon\})) \times K$$

NDFSM and DFSM

Δ is the transition relation. It is a finite subset of

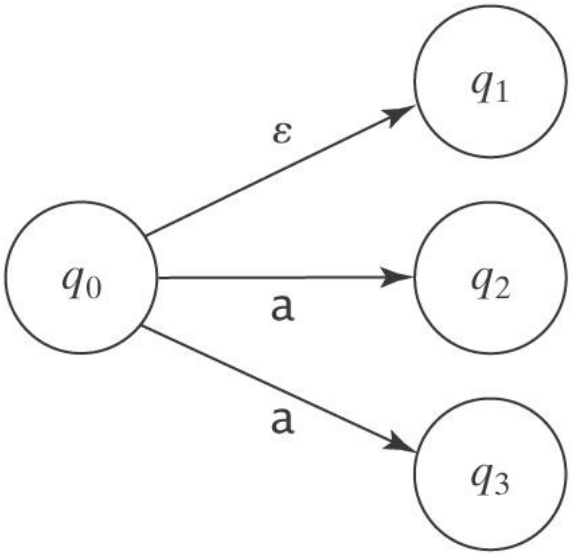
$$(K \times (\Sigma \cup \{\varepsilon\})) \times K$$

Recall the definition of DFSM:
$M = (K, \Sigma, \delta, s, A)$, where:
K is a finite set of states
Σ is an alphabet
$s \in K$ is the initial state
$A \subseteq K$ is the set of accepting states, and
δ is the transition function from $(K \times \Sigma)$ to K

Key difference:

- In every configuration, a DFSM can make exactly one move; this is not true for NDFSM
  - M may enter a config. from which two or more competing moves are possible. This is due to (1) ε-transition (2) relation, not function

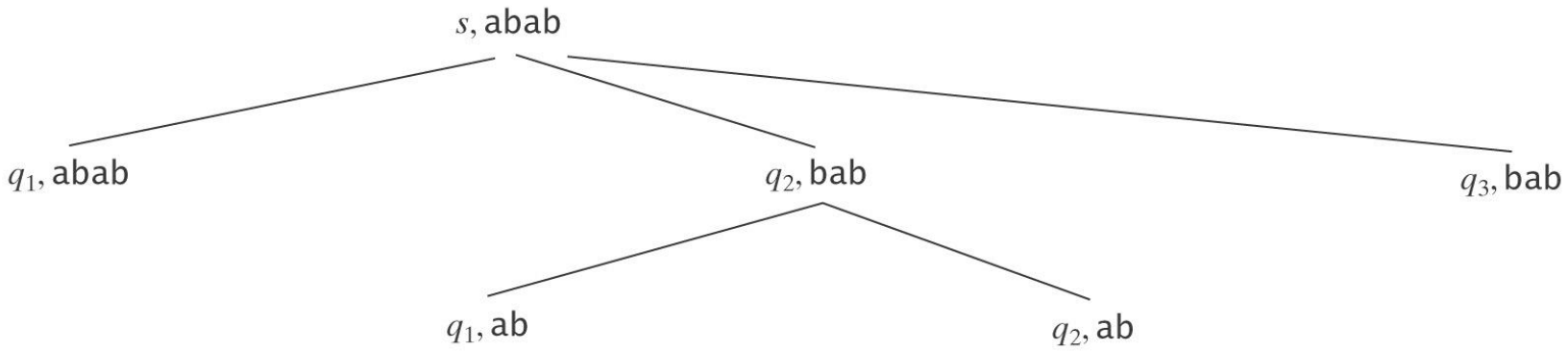
Sources of Nondeterminism



- Nondeterminism is a generalization of determinism
  - Every DFSM is automatically an NDFSM
- Can be viewed as a type of parallel computation
  - Multiple independent threads run concurrently
- Recall ***Theorem:*** Every DFSM *M*, on input *w*, halts in at most |*w*| steps. Can we say the same for NDFSM?

Envisioning the operation of *M*

- Explore a search tree (depth-first):
  - Each node corresponds to a configuration of *M*
  - Each path from the root corresponds to the *path* we have defined

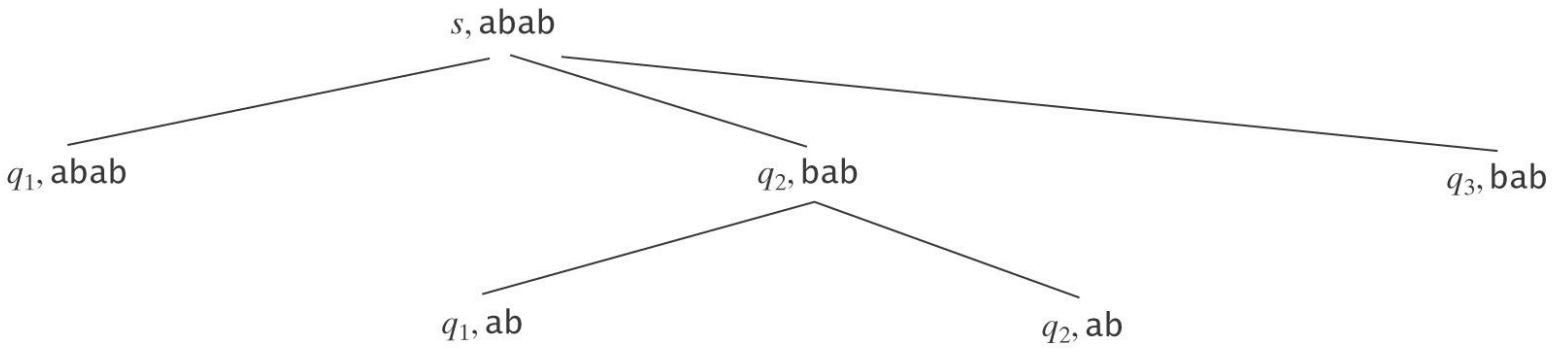


- Alternatively, imagine following all paths through *M* in parallel (breath-first)

Analyzing Nondeterministic FSMs

Given an NDFSM *M*, how can we analyze it to determine if it accepts a given string?

- Depth-first explore a search tree:



- Follow all paths in parallel (breath-first)

Accepting

Recall: a path is a maximal sequence of steps from the start configuration.

- *M* accepts a string *w* iff there exists *some path* that accepts it.
  - Same as DFSM, (q, ε) where $q \in A$ is an accepting configuration

M halts upon acceptance.

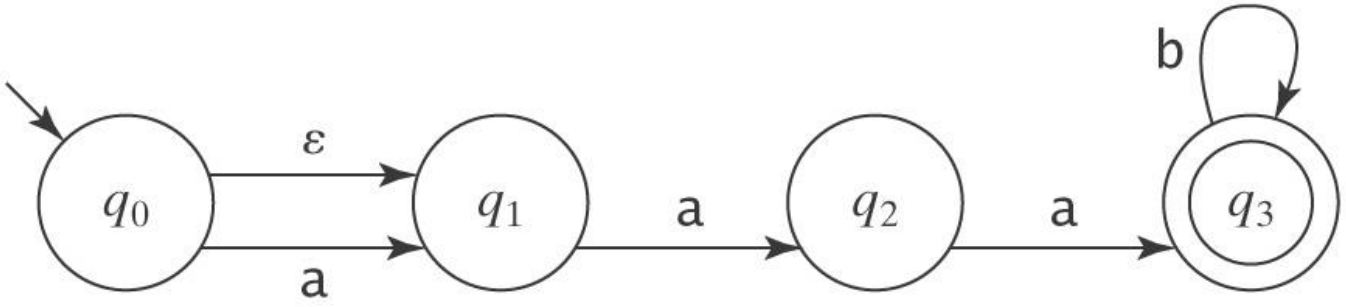
- Other paths may:
  - Read all the input and halt in a nonaccepting state
  - Reach a dead end where no more input can be read
  - Loop forever and never finish reading the input

The language accepted by *M*, denoted *L*(*M*), is the set of all strings accepted by *M*.

- *M* rejects a string *w* iff all paths reject it.
- It is possible that, on input $w \notin L(M)$, *M* neither accepts nor rejects. In that case, no path accepts and some path does not reject.

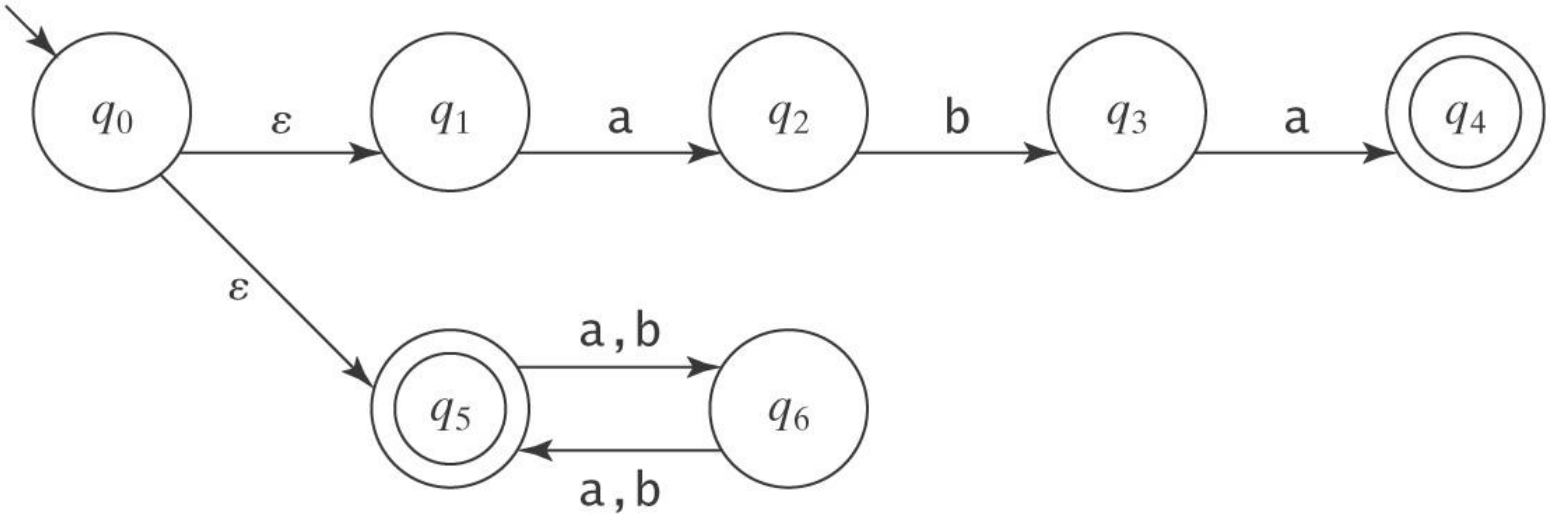
Optional Initial a

L = {*w* ∈ {a, b}* : *w* is made up of an optional a followed by aa followed by zero or more b's}.



Two Different Sublanguages

$L = \{w \in \{\texttt{a}, \texttt{b}\}^* : w = \texttt{aba}$ or $|w|$ is even$\}$.



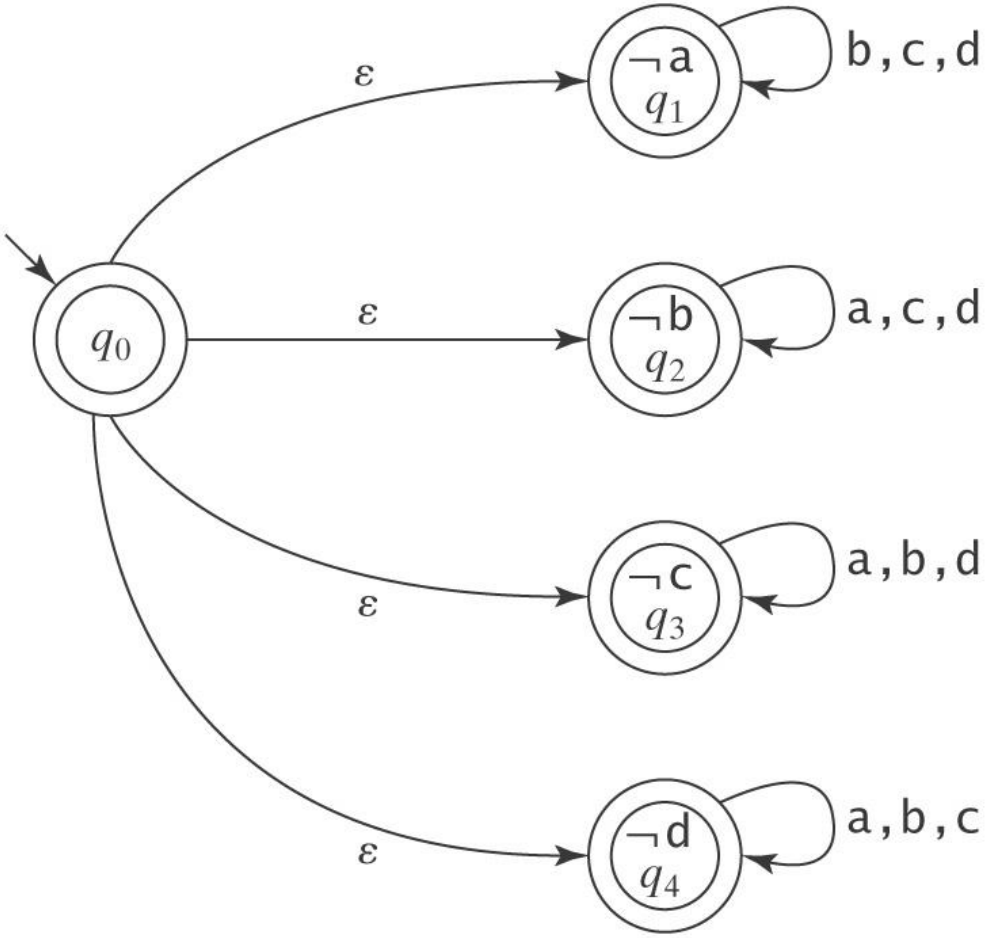
If M is a NDFSM and $\varepsilon \in L(M)$, can we say the start state of M must be an accepting state?

Why NDFSM?

- High level tool for describing complex systems
- Can be used as the basis for constructing efficient practical DFSMs
    - Build a simple NDFSM
    - Convert it to an equivalent DFSM
    - Minimize the result

The Missing Letter Language

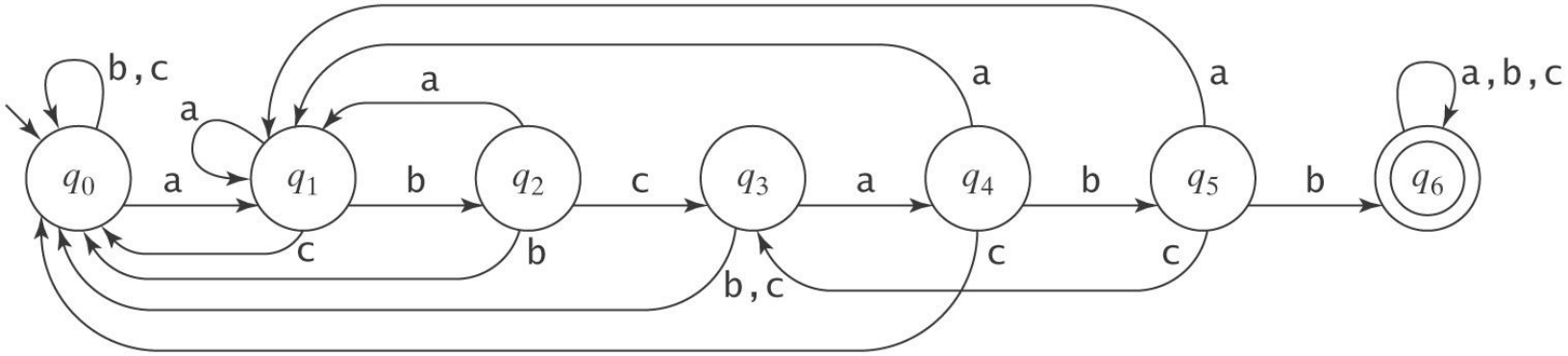
Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$. Let $L_{Missing} = \{w$: there is a symbol $a_i \in \Sigma$ not appearing in $w\}$



Pattern Matching

$L = \{w \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^* : \exists x, y \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^* \ (w = x \ \texttt{abcabb} \ y)\}$.

A DFSM:

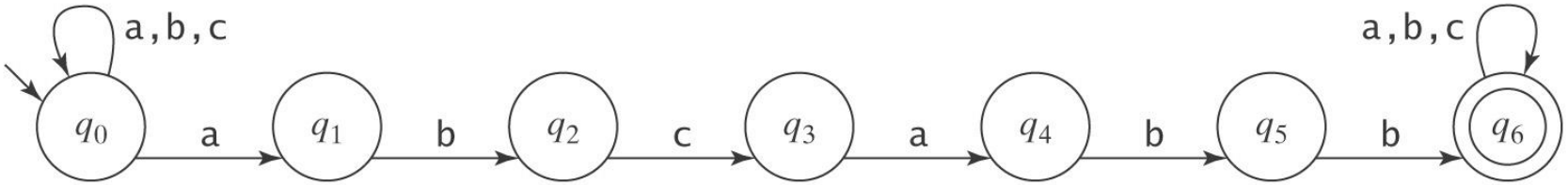


Works, but complex to design, error prone

Pattern Matching

$L = \{w \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^* : \exists x, y \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^* (w = x\ \texttt{abcabb}\ y)\}$.

An NDFSM:

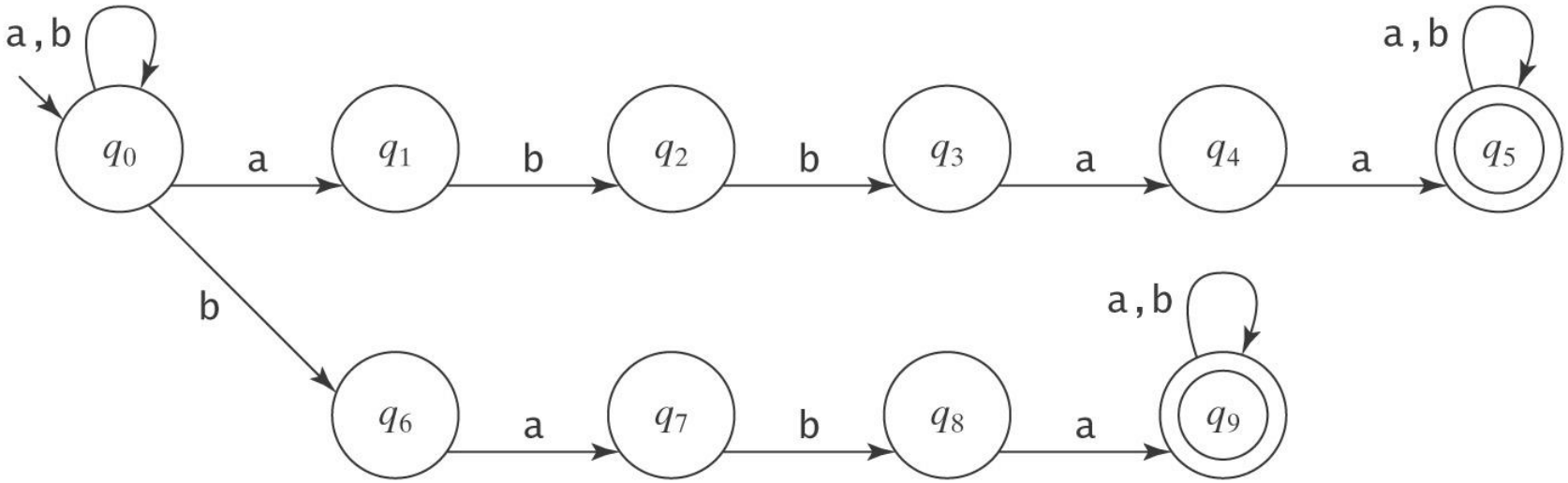


Why ND but not D?
Why is it hard to create a DFSM?
Nondeterminism: “lucky guesses”

Multiple Keywords

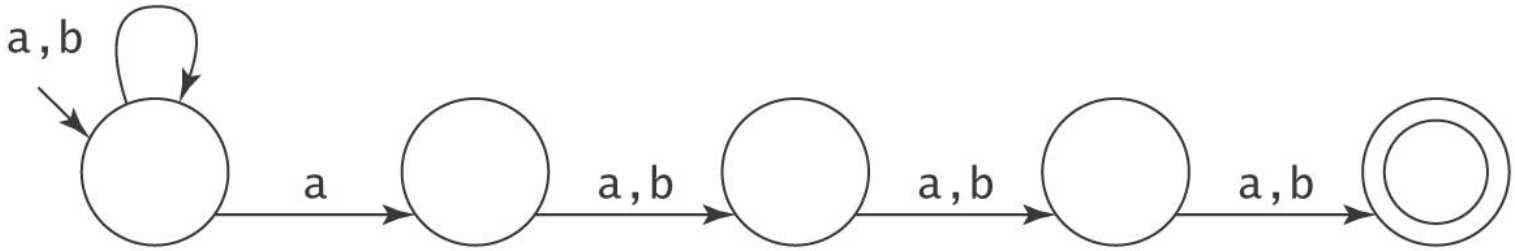
$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \exists x, y \in \{\mathtt{a}, \mathtt{b}\}^*$

$((w = x\ \mathtt{abbaa}\ y) \vee (w = x\ \mathtt{baba}\ y))\}.$



Checking from the End

$L = \{w \in \{\texttt{a}, \texttt{b}\}^* :$
the fourth to the last character is $\texttt{a}\}$



Nondeterministic and Deterministic FSMs

Clearly: {Languages accepted by a DFSM} $\subseteq$ {Languages accepted by an NDFSM}

Theorem:
For each DFSM *M*, there is an equivalent NDFSM *M'*.

- $L(M') = L(M)$

More interestingly:

Theorem:
For each NDFSM, there is an equivalent DFSM.

Nondeterministic and Deterministic FSMs

Theorem*:* For each NDFSM, there is an equivalent DFSM.

Proof*:* By construction:

Given an NDFSM $M = (K, \Sigma, \Delta, s, A)$,
we construct $M' = (K', \Sigma, \delta', s', A')$, where

$K' = \mathcal{P}(K)$

$s' = eps(s)$

$A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$

$\delta'(Q, a) = \bigcup\{eps(p): p \in K$ and
$(q, a, p) \in \Delta$ for some $q \in Q\}$

An Algorithm for Constructing the Deterministic FSM

1. Compute the *eps*(*q*)'s.

2. Compute $s' = eps(s)$.

3. Compute δ'.

4. Compute K' = a subset of $\mathcal{P}(K)$.

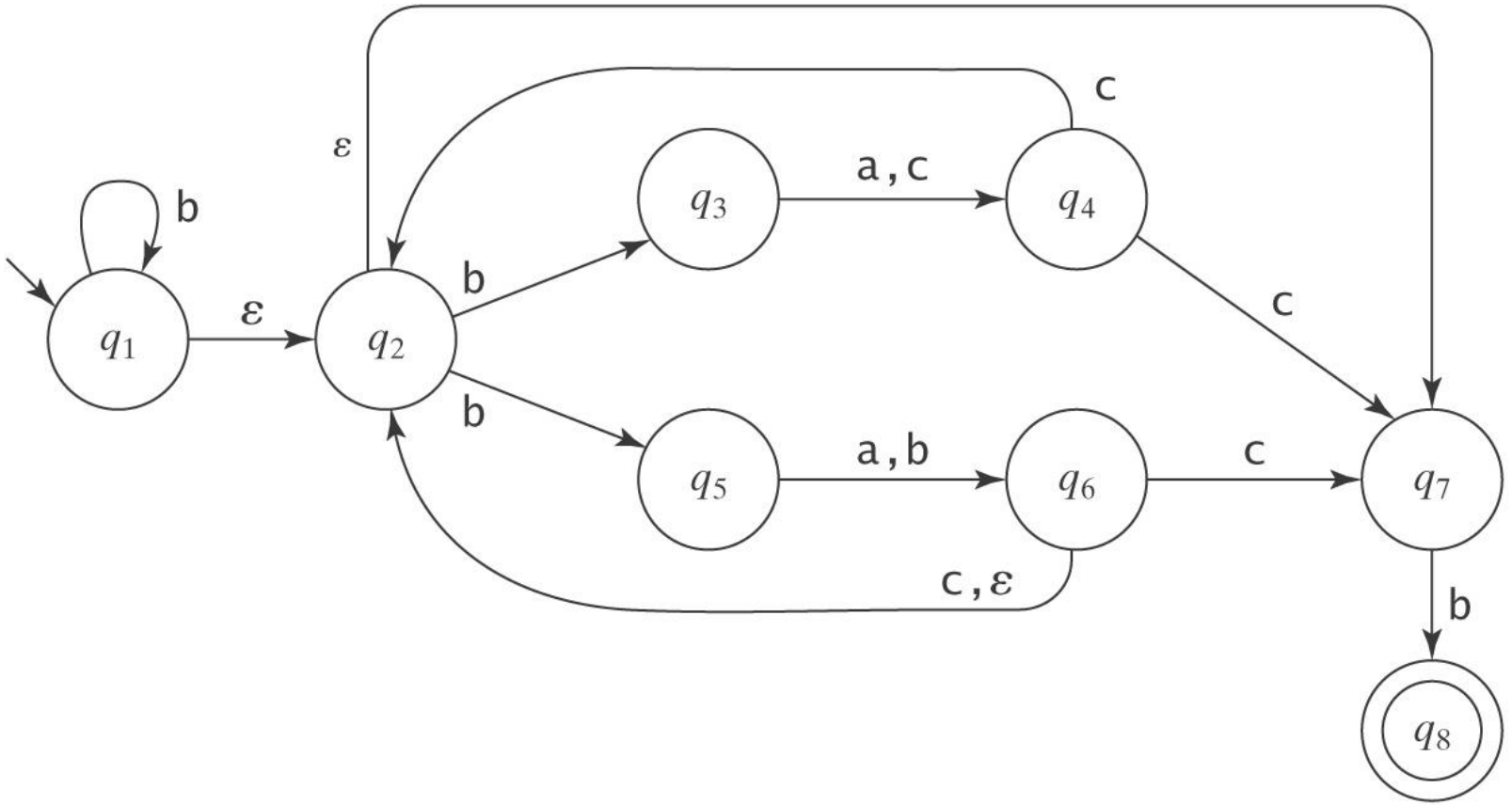
5. Compute $A' = \{Q \in K' : Q \cap A \neq \varnothing\}$.

The Algorithm *ndfsmtodfsm*

ndfsmtodfsm(*M*: NDFSM) =

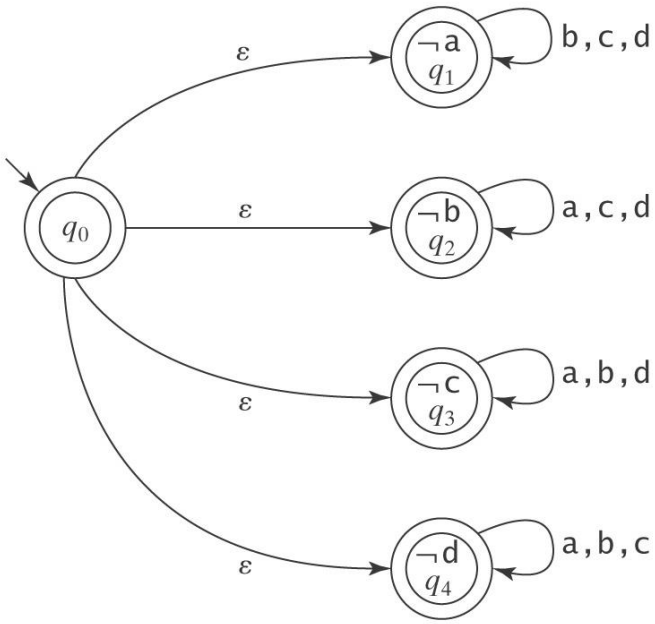
1. For each state *q* in K_M do:
   1.1 Compute *eps*(*q*).
2. *s′* = *eps*(*s*)
3. Compute δ':
   3.1 active-states = {*s′*}.
   3.2 δ' = $\varnothing$.
   3.3 While there exists some element *Q* of *active-states* for which δ' has not yet been computed do:
      For each character *c* in Σ_M do:
         new-state = $\varnothing$.
         For each state *q* in *Q* do:
            For each state *p* such that (*q, c, p*) $\in \Delta$ do:
               new-state = *new-state* $\cup$ *eps*(*p*).
         Add the transition (*Q*, *c*, *new-state*) to δ'.
         If *new-state* $\notin$ *active-states* then insert it.
4. *K′* = *active-states.*
5. *A′* = {*Q* $\in$ *K′* : *Q* $\cap$ *A* $\neq \varnothing$ }.

An Example



The Number of States May Grow Exponentially

$|\Sigma| = n$



No. of states after 0 chars: = 1

No. of new states after 1 char: $\binom{n}{n-1}$ = n

No. of new states after 2 chars: $\binom{n}{n-2}$ = $n(n\text{-}1)/2$

No. of new states after 3 chars: $\binom{n}{n-3}$ = $n(n\text{-}1)(n\text{-}2)/6$

Total number of states after n chars: 2^n

Nondeterministic FSMs as Algorithms

Real computers are deterministic, so we have three choices if we want to execute an NDFSM:

1. Convert the NDFSM to a deterministic one:
   - Conversion can take time and space $2^{|K|}$.
   - Time to analyze string w: $\mathcal{O}(|w|)$

2. Simulate the behavior of the nondeterministic one by constructing sets of states "on the fly" during execution
   - No conversion cost
   - Time to analyze string w: $\mathcal{O}(|w| \times |K|^2)$

3. Do a depth-first search of all paths through the nondeterministic machine.

Note on Nondeterminism

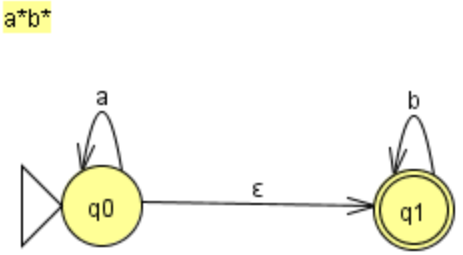
Used in computability/decidability:

- NDFSM: does not add power
- NDPDA: a bit messy, adds some power
- NDTM: does not add power
- Summary: TM is the most powerful machine, w.r.t. computability / decidability. So in general, ND does not add power.

Used in complexity where efficiency matters:

- Use NP as an example
- The class NP is the set of languages that are polynomially decidable by a nondeterministic Turing machine.
- Here, we can think of a nondeterministic algorithm as acting in two phases:
  - Guess a solution (called a certificate) from a finite number of possibilities
  - Test whether it indeed solves the problem (verification algorithm)
- Verification must take polynomial time for NP
- Summary: it adds power (efficiency), as we can take "lucky guesses" instead of exploring all paths

7. JFLAP



- What's JFLAP? http://www.jflap.org/whatis.html
- Download, tutorial: http://www.jflap.org/
- Can also use applet: http://www.cs.duke.edu/csed/jflap/jflaptmp/applet/demo.html
- Preferences: set empty string to epsilon
- FSM, TM, Mealy, all fine. Just PDA has different definition from ours:
  - with Z, a stack marker (we don't have it)
  - Either finite state or empty stack acceptance (we use both)
  - To make our PDAs run in JFLAP: choose acceptance option properly. Sometimes may need to remove Z.
- To run a machine: step (step with closure for ND), fast run, **multiple run**
- Grammar:
  - Test for grammar type
  - Brute force parse, multiple brute force parse
  - Convert

Finite State Machines

Transducers
Markov Models
Hidden Markov Models

Finite State Transducers

- A finite state transducer (FST) is a finite state machine, that transduces (translates) an input string into an output string.
  - instead of {0,1} as in FSMs (acceptors / recognizers)
  - input tape, output tape
  - Moore machine and Mealy machine

- Moore machine: outputs are determined by the current state alone (and do not depend directly on the input)
  - Advantage of the Moore model is a simplification of the behavior

- Mealy machine: output depends on current state and input

Moore and Mealy

Edward F. Moore (1925 – 2003)

- Professor of Math and CS in UW-Madison
- Memorial resolution by Jin-Yi Cai, Larry Landweber, Olvi Mangasarian https://kb.wisc.edu/images/group222/shared/2003-09-29FacultySenate/1727(mem_res).pdf

George H. Mealy (1927 – 2010)

worked at the Bell Laboratories in 1950's and was a Harvard University professor in 1970's

Moore Machine

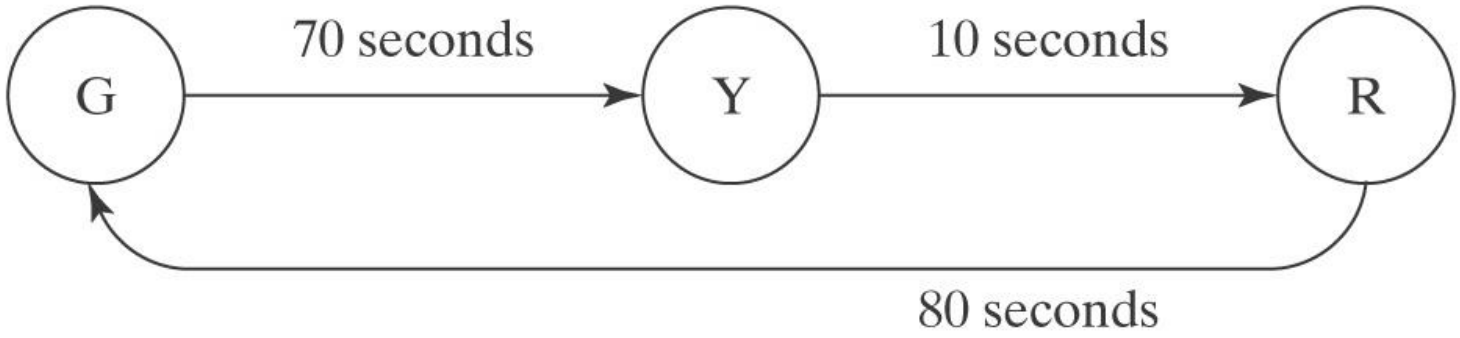
A ***Moore machine*** $M = (K, \Sigma, O, \delta, D, s, A)$, where:

- K is a finite set of states
- Σ is an input alphabet
- O is an output alphabet
- $s \in K$ is the initial state
- $A \subseteq K$ is the set of accepting states, (not important for some app.)
- δ is the transition function from $(K \times \Sigma)$ to K,
- D is the output function from K to O^*.

M outputs each time it lands in a state.

A Moore machine M computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

A Simple US Traffic Light Controller



Mealy Machine

A ***Mealy machine*** $M = (K, \Sigma, O, \delta, s, A)$, where:

- K is a finite set of states
- Σ is an input alphabet
- O is an output alphabet
- $s \in K$ is the initial state
- $A \subseteq K$ is the set of accepting states (not important for some app.)
- δ is the transition function from $(K \times \Sigma)$ to $(K \times O^*)$

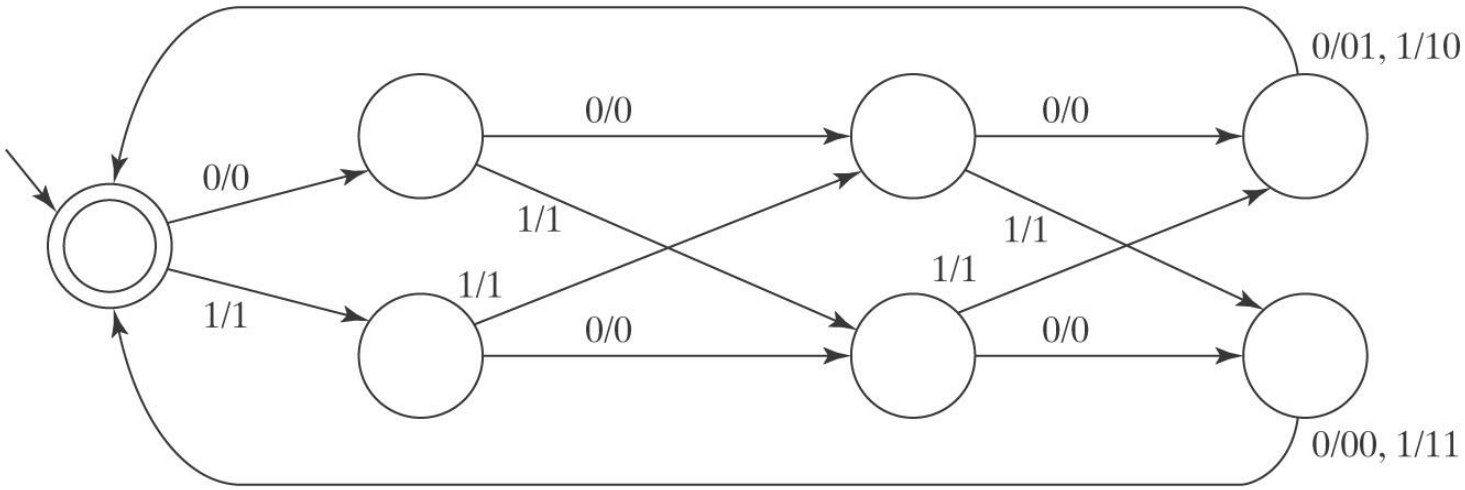
M outputs each time it takes a transition.

A Mealy machine M computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

An Odd Parity Generator

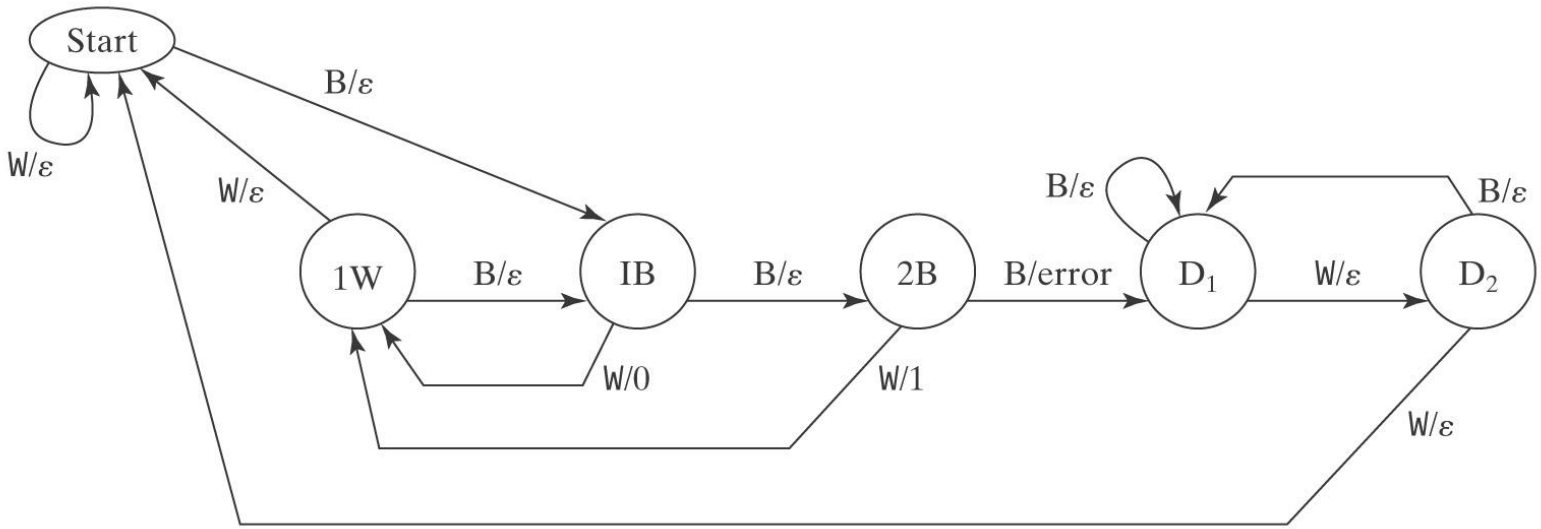
After every four bits, output a fifth bit such that each group of five bits has odd parity.

0 0 0 0 1 0 0 0 1 1 1 1



A Bar Code Scanner

A Bar Code Scanner



Stochastic FSMs

Markov Models
Hidden Markov Models (HMM)

- ***Stochastic*** (from the Greek "Στόχος" for "aim" or "guess")
  - means random
  - based on theory of probability

- A ***stochastic process*** is one whose behavior is non-deterministic in that a system's subsequent state is determined both by the process's predictable actions and by a random element.

Andrey Markov

- 1856 – 1922
- Russian mathematician
- Stochastic process, Markov chain

- With younger brother, proved Markov brothers' inequality
- Son, another Andrey Andreevich Markov (1903-1979), was also a notable mathematician (Markov algorithm).

Markov Models

- A random process where all information about the future is contained in the present state
  - i.e. one does not need to examine the past to determine the future
  - can be represented by FSM

- A Markov model is an NDFSM in which the state at each step can be predicted by a probability distribution associated with the current state.
  - Markov property: behavior at time *t* depends only on its state at time *t*-1
  - sequence of outputs produced by a Markov model is called a ***Markov chain***

Formally, a Markov model is a triple $M = (K, \pi, A)$:

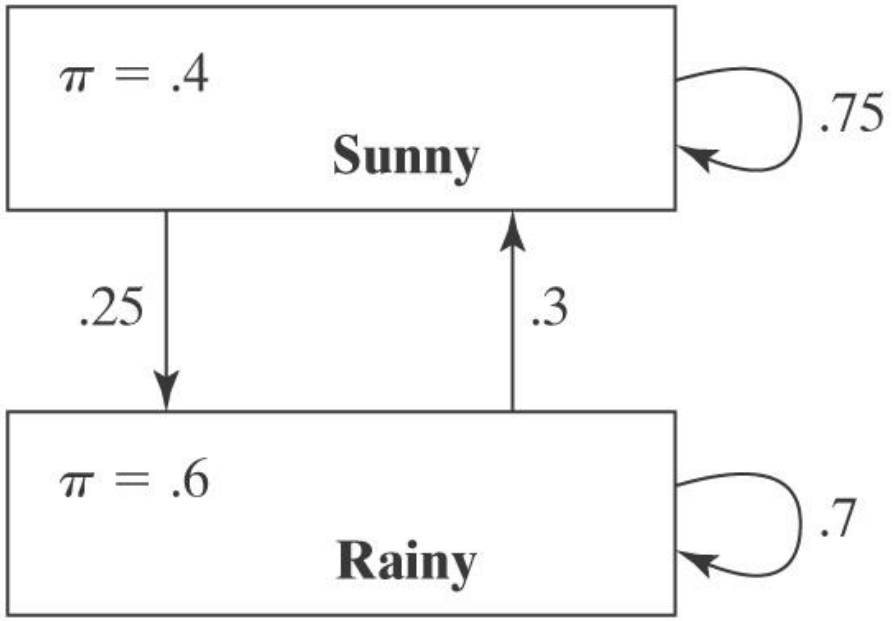
- K is a finite set of states
- π is a vector of initial probabilities of each of the states
- $A[p, q] = \Pr(\text{state } q \text{ at time } t \mid \text{state } p \text{ at } t - 1)$
  - the probability that, if M is in p, it will go to q next

Markov Models

π = (0.4, 0.6)

A =

	Sunny	Rainy
Sunny	0.75	0.25
Rainy	0.3	0.7



To use a Markov model, we first need to use data to create the matrix *A* (discuss later)

What can we do with a Markov model?

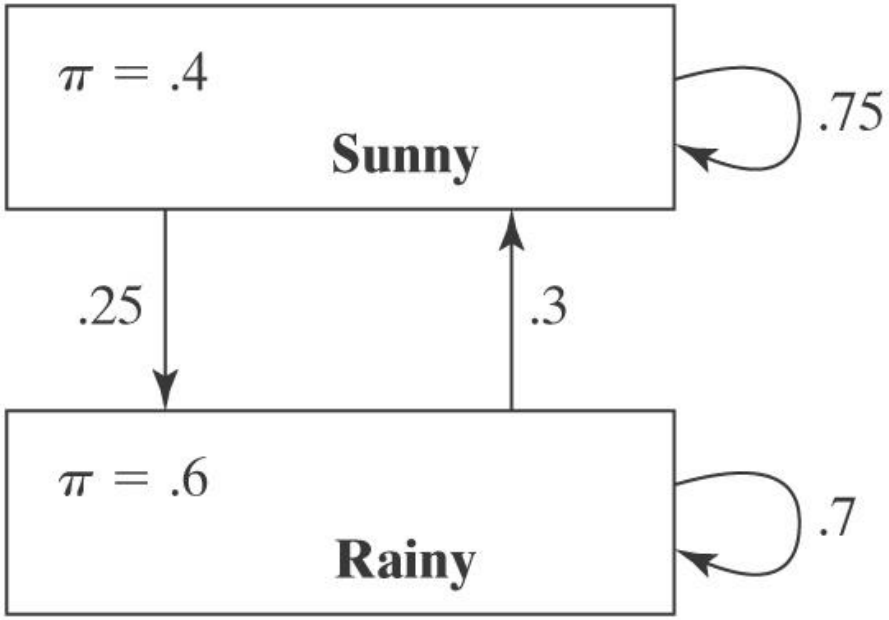
- Generate almost natural behavior
- Estimate the probability of some outcome

Estimating Probabilities

Given a Markov model that describes some random process, what is the probability that we will observe a particular sequence S_1 S_2 … S_n of states?

$$\Pr(s_1 s_2 \ldots s_n) = \pi[s_1] \cdot \prod_{i=2}^{n} A[s_{i-1}, s_i]$$

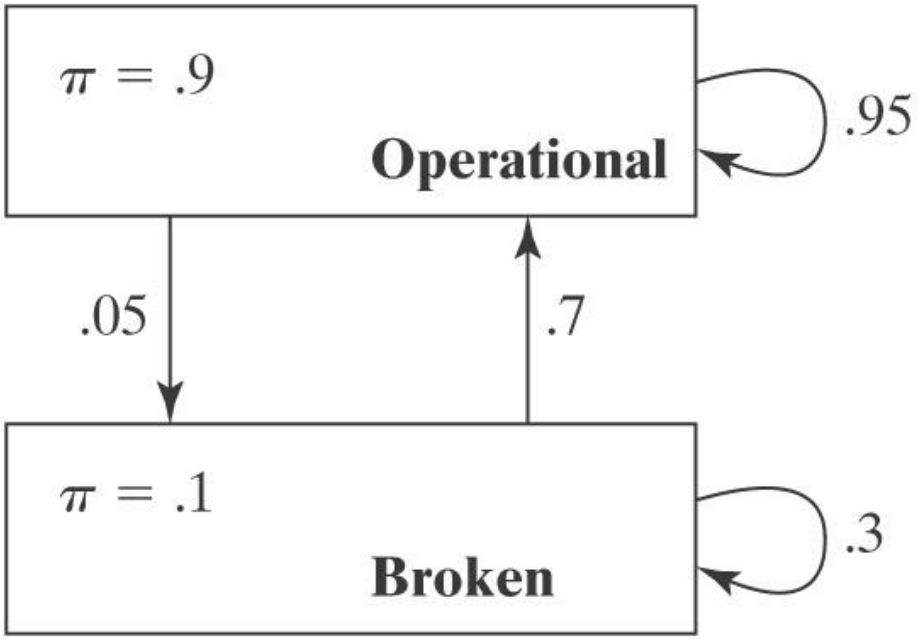
Markov Models



Assumes that the weather on day t is influenced only by the weather on day t - 1

- What's the probability that it will be sunny 5 days in a row?
- Given it's sunny today, what's the probability that it will be sunny 4 more days?

Modeling System Performance



If up now, what is probability of staying up for an hour (3600 time steps)?

$$\Pr(s_1 s_2 \ldots s_n) = \pi[s_1] \cdot \prod_{i=2}^{n} A[s_{i-1}, s_i] = .95^{3600} = 6.3823 \cdot 10^{-81}$$

Where do the Probabilities in a Markov Model Come From

- Examining real datasets and discover the probabilities that best describe those data
  - A log of system behavior over some recent period of time
- Suppose we have observed the output sequences: TPTQPQT and SSPTPQQPSTQPTTP
  - A[P,Q] = the number of times the pair PQ appears / total number of times P appears in any position except the last
  - π [P] is the number of times P is the first symbol / total number of sequences
- Models are huge and evolve over time

N^{th} Order Markov Models

- 0^{th} order models depend on no prior state.

- 1^{st} order models depend on one previous state.
  - If k states, need to specify k^2 transition probabilities
  - k x k
- …

- n^{th} order models depend on n previous states.
  - If k states, need to specify $k^{(n+1)}$ transition probabilities
  - $\underbrace{k \text{ x } k \text{ x } k \ldots}_{n} \text{ x } k$

Markov Text Generators

• Markov processes can be used to generate superficially "real-looking" text given a sample document

• They are used in a variety of recreational "parody generator" software

• These processes are also used by spammers to inject real-looking hidden paragraphs into unsolicited email in an attempt to get these messages past spam filters.

• ***Markov*-chain *text generator***
http://projects.haykranen.nl/markov/demo/

A Letter-Level Model of English

- (*n* = 1): a a idjume Thicha lanbede f nghecom isonys rar t r ores aty Ela ancuny, ithi, witheis weche
- (*n* = 2): Ther to for an th she con simach a so a impty dough par we forate for len postrit cal nowillopecide allexis inteme numbectionsityFSM Cons onste on codere elexpre ther
- (*n* = 3): Ouput that the collowing with to that we'll in which of that is returesult is alway ther is id, the cal on the Prove be and N.
- (*n* = 4): Notice out at least to steps if new Turing derived for explored. What this to check solved each equal string it matrix (i, k, y must be put part can may generated grammar in D.
- (*n* = 5): So states, and Marting rules of strings. We may have been regions to see, a list. If ? ? unrestricted grammars exist a devices are constructive-state i back to computation
- (*n* = 6): We'll have letter substituted languages that L(G) since we drop the address to the rule1 were counterexample, that is that we are true when we switched in how we
- (*n* = 7): If it does a context-free language 3. If the model of which corresponding b's. M must have chosen strings as a tree such characters of some p.

A Word-Level Model of English

- (*n* = 1): there exists at the idea was presented for some finite state 3 together. So U begins in this approach, it is labeled with wj as some model to position-list, solve-15 can reduce every derivation becomes M1 and the number of A building efficient algorithms.
- (*n* = 2): The language to be if the various grammar formalisms in which they were deposited make no moves are possible. The competition can come from somewhere. Fortunately, there are edges from level nodes to level nodes. Now suppose that we do with a successful value.
- (*n* = 4): Again, let st be the working string at any point in its computation it will have executed only a finite number of squares can be nonblank. And, even if M never halts, at any point in its computation it will have executed only a finite number of choices at each derivation step and since each path that is generated must eventually end, the Turing machine M that computes it.
- (*n* = 5): Is there any computational solution to the problem? • If there is, can it be implemented using some fixed amount of memory? • If there is no such element, then choose will: • Halt and return False if all the actions halt and return False. • Fail to halt if there is no mechanism for determining that no elements of S that satisfy P exist. This may happen either because v and y are both nonempty and they both occur in region n

Internet Applications: Google

- The PageRank of a webpage as used by Google is defined by a Markov chain.
  - Pagerank values are basically converged long term visit rates by a Markov chain random surfer on the internet graph.

- Markov models have also been used to analyze web navigation behavior of users.
  - A user's web link transition on a particular website can be modeled using first- or second-order Markov models
  - make predictions regarding future navigation and to personalize the web page for an individual user.

Generating Music

Markov chains are employed in algorithmic music composition, to generate random music

Musikalisches Würfelspiel (A musical dice game): https://dice.humdrum.org/

Many on Youtube:

https://www.youtube.com/watch?v=IIOiAK0x4vA

https://www.youtube.com/watch?v=Z0lmk3FjLwA

https://www.youtube.com/watch?v=H3xgdDTvvlc

WolframTones: a commercial application of Cellular Automaton (a discrete model of computation studied in automata theory) to music composition.

http://tones.wolfram.com/

Generative AI: Suno

- Other applications? Poetry?

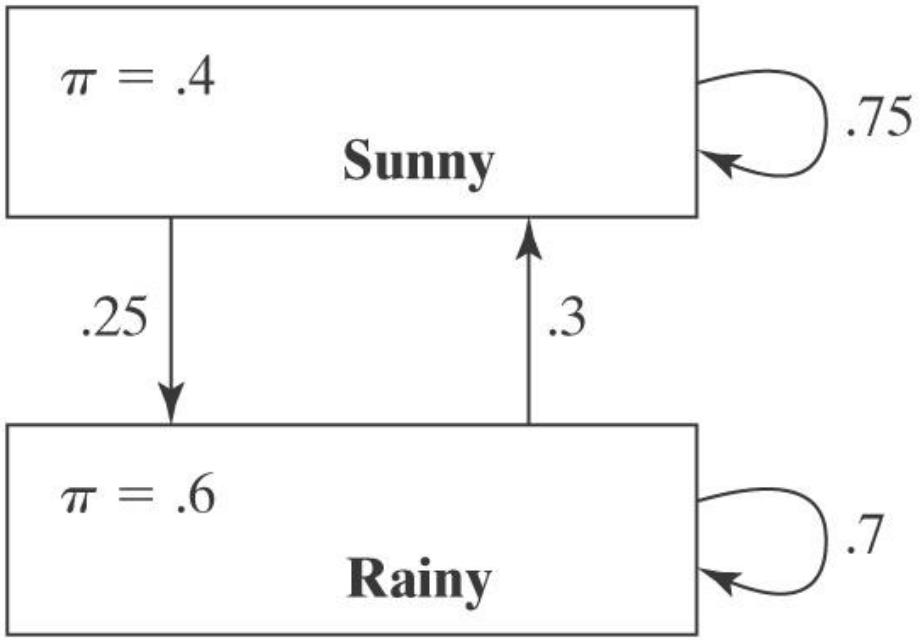
Hidden Markov Models

Suppose that the states themselves are not visible. But states emit outputs with certain probabilities and the outputs are visible:

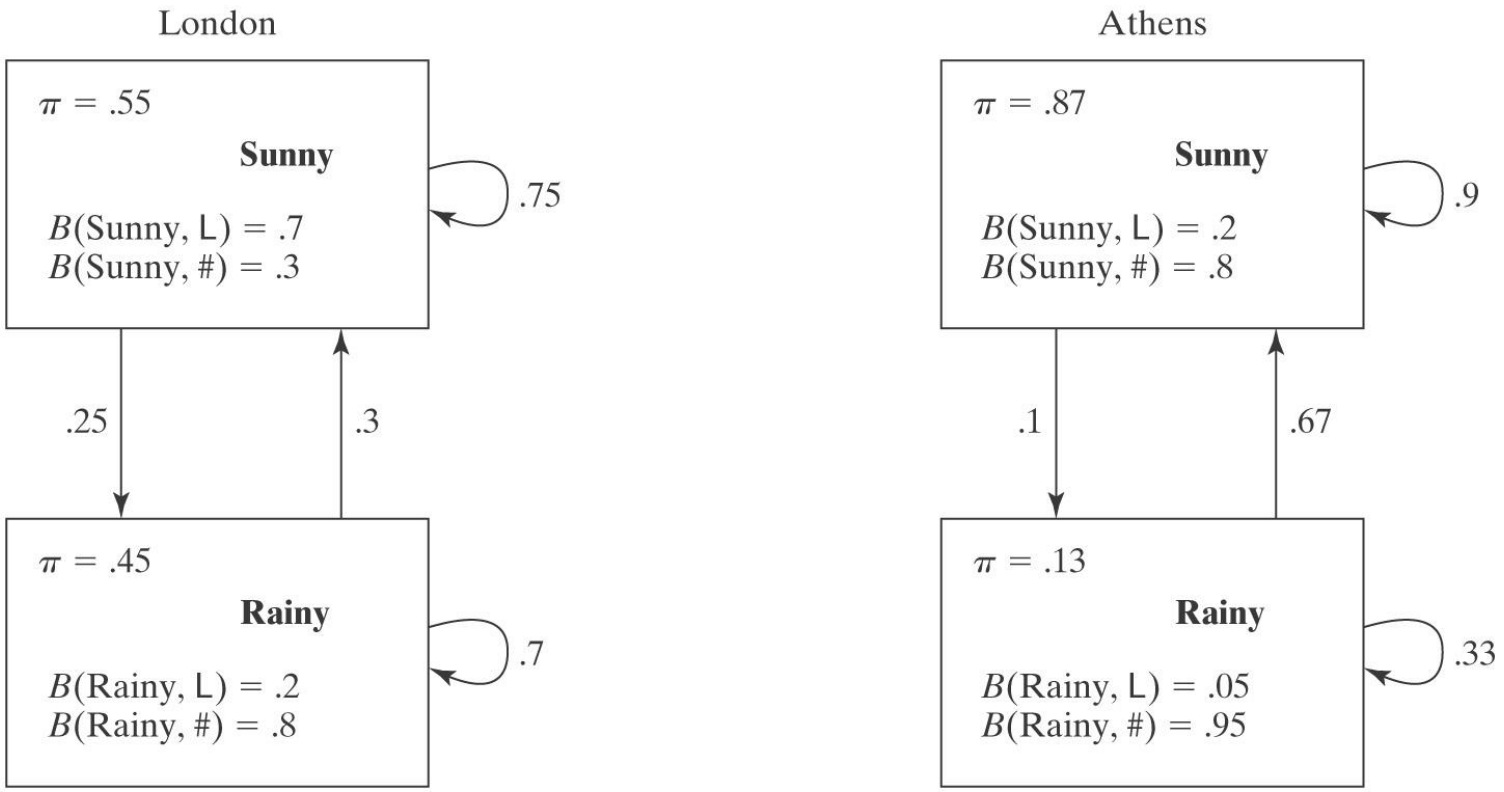
If we could observe the states:

Sunny and Rainy

e.g., mood



When We Cannot Observe the States



- Also two states: Sunny and Rainy, but not visible
- Output symbols: L (passport loss) and # (no loss)
- Cannot observe the weather (states), but want to infer them from passport loss
  - Probability of a passport loss is a function of weather
- *B*(Sunny, *L*) = .7: the probability *M* will emit (or output) L if it is in *Sunny*
- One HMM for London, one HMM for Athens

Hidden Markov Models

An HMM M is a quintuple (K, O, π, A, B), where:

- K is a finite set of states,
- O is the output alphabet,
- π is a vector of initial probabilities of the states,
- A is a matrix of transition probabilities:
  $A[p, q] = \Pr(\text{state } q \text{ at time } t \mid \text{state } p \text{ at time } t - 1)$,
- B, the confusion matrix of output probabilities.
  $B[q, o] = \Pr(\text{output } o \mid \text{state } q)$.

Recall, a Markov model is a triple $M = (K, \pi, A)$:

- K is a finite set of states
- π is a vector of initial probabilities of each of the states
- $A[p, q] = \Pr(\text{state } q \text{ at time } t \mid \text{state } p \text{ at } t - 1)$
  - the probability that, if M is in p, it will go to q next

HMM Associated Problems

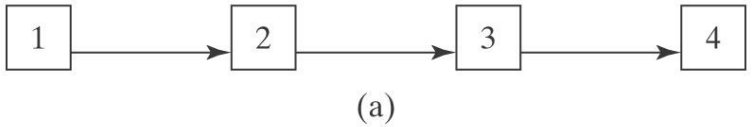
To use an HMM, we typically have to solve some or all of the following problems:

- The decoding problem: Given an observation sequence *O* and an HMM *M*, discover the path through *M* that is most likely to have produced *O*
  - we observe the report ###L from London, what is the most likely sequence of weather states
  - can be solved efficiently with the **Viterbi** algorithm (DP)
- The evaluation problem: Given an observation *O* and a set of HMMs that describe a collection of possible underlying models, choose the HMM that is most likely to have generated *O*
  - *w*e observe the report ###L from somewhere.
  - can be solved efficiently with the **forward algorithm**, similar to Viterbi except that it considers all paths through a candidate HMM, rather than just the most likely one
- The training problem: learning π, *A*, and *B*
  - Baum-Welch algorithm that employs expectation maximization (EM)

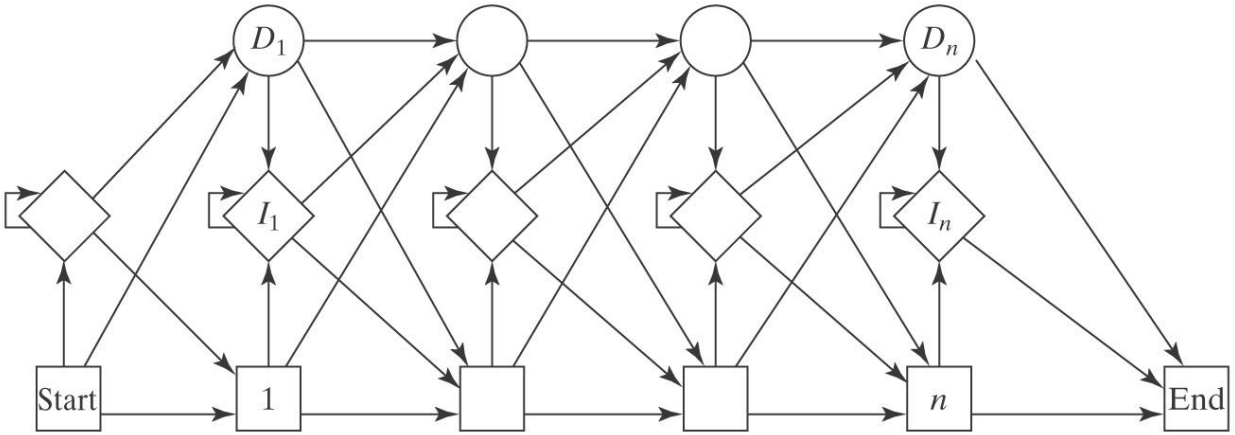
An Example from Biology

K.3.3 HMMs for sequence matching (p973):

```
      A G H   T   Y W D N R
      A G H D T   Y E N N R Y
  Y P A G Q D T   Y W N N
      A G H D T T Y W N N
```



(a)



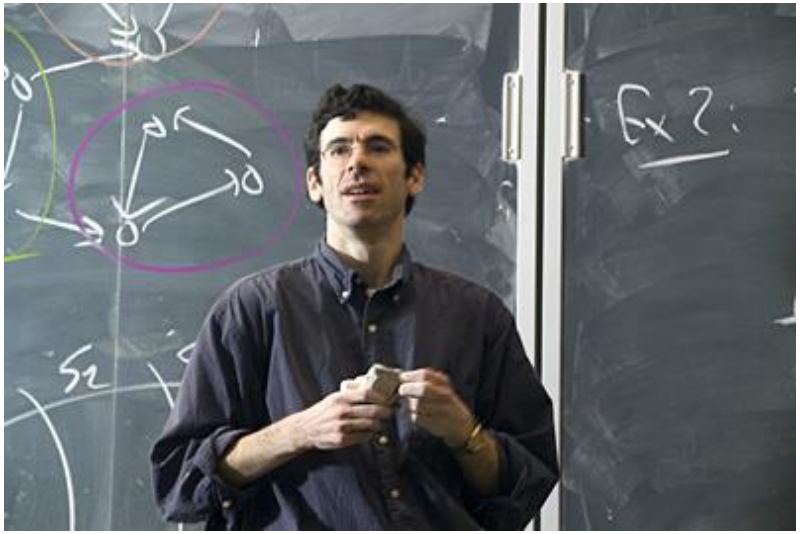
(b)

The Google Thing

Larry Page and Sergey Brin

Sergey was with Jeff Ullman
http://infolab.stanford.edu/~sergey/

Jon Kleinberg
Rebel King (anagram for “Kleinberg”)



HITS
Father of Google?

Origins of PageRank: Citation analysis (1)

- Citation analysis: analysis of citations in the scientific literature
- Example citation: "Miller (2001) has shown that physical activity alters the metabolism of estrogens."
- "Miller (2001)" is a hyperlink linking two scientific articles.
- One application of these "hyperlinks" in the scientific literature:
  - Measure the similarity of two articles by the overlap of other articles citing them.
  - This is called cocitation similarity.
- Cocitation similarity on the web?

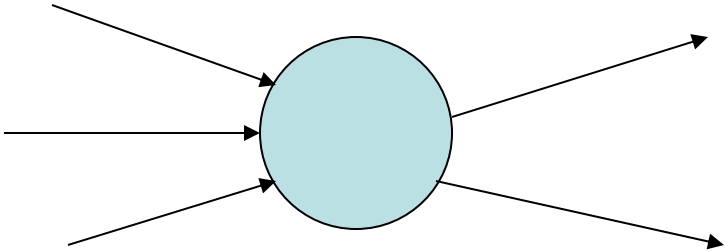
Cocitation similarity on Google: similar pages

Origins of PageRank: Citation analysis (2)

- Citation frequency can be used to measure the impact of an article.
  - Each article gets one vote.
  - Not a very accurate measure
- Better measure: weighted citation frequency / citation rank
  - An article's vote is weighted according to its citation impact.
  - Circular? No: can be formalized in a well-defined way.
  - This is basically PageRank.
  - PageRank was invented in the context of citation analysis by Pinsker and Narin in the 1960s.

Query-independent ordering

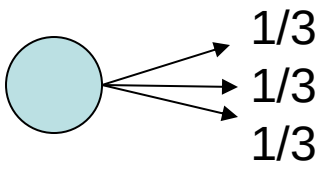
- First generation link-based ranking for web search
  - using link counts as simple measures of popularity.
  - simple link popularity: number of in-links



  - First, retrieve all pages meeting the text query (say ***venture capital***).
  - Then, Order these by the simple link popularity
- Easy to spam. Why?

Basic for Pagerank: random walk

- Imagine a web surfer doing a random walk on the web page:
  - Start at a random page
  - At each step, go out of the current page along one of the links on that page, equiprobably

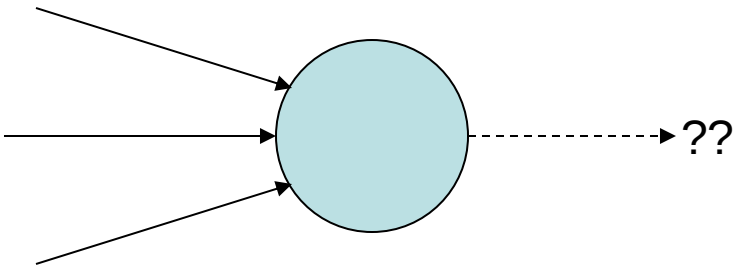


- "In the steady state" each page has a long-term visit rate - use this as the page's score.
- So, pagerank = steady state probability

  = long-term visit rate

Not quite enough

- The web is full of dead-ends.
  - Random walk can get stuck in dead-ends.
  - Makes no sense to talk about long-term visit rates.

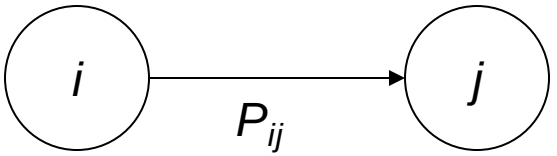


Teleporting

- Teleport operation: surfer jumps from a node to any other node in the web graph, chosen uniformly at random from all web pages
- Used in two ways:
  - At a dead end, jump to a random web page.
  - At any non-dead end, with probability $0 < \alpha < 1$ (say, $\alpha = 0.1$), jump to a random web page; with remaining probability $1 - \alpha$ (0.9), go out on a random link
- Now cannot get stuck locally
- There is a long-term rate at which any page is visited
  - Not obvious, explain later
  - How do we compute this visit rate?

Markov chains

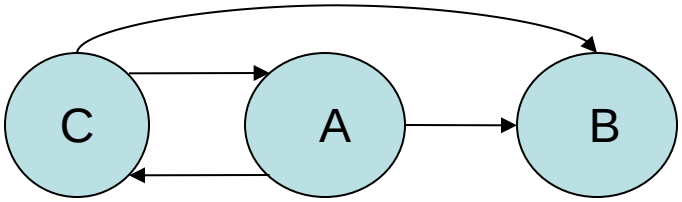
- A Markov chain consists of n states, plus an $n \times n$ transition probability matrix **P**.
- At each step, we are in exactly one of the states.
- For $1 \leq i, j \leq n$, the matrix entry P_{ij} tells us the probability of j being the next state, given we are currently in state i.



- Clearly, for each i, $\sum_{j=1}^{n} P_{ij} = 1.$
- Markov chains are abstractions of random walk
  - State = page

Exercise

Represent the teleporting random walk as a Markov chain, for the following case, using transition probability matrix



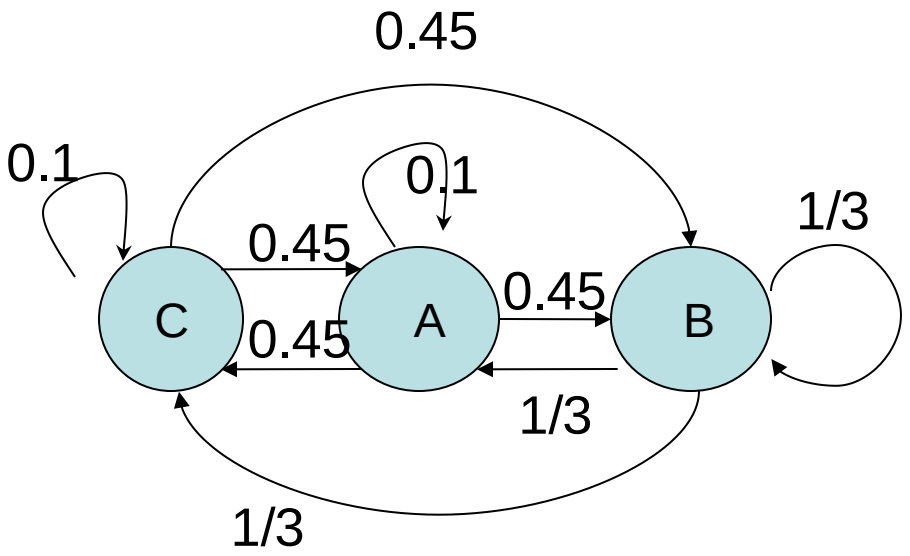
Link structure

$\alpha = 0.3$:

Transition matrix

0.1	0.45	0.45
1/3	1/3	1/3
0.45	0.45	0.1

State diagram



Formalization of visit: probability vector

- A probability (row) vector **x** = $(x_1, \ldots x_n)$ tells us where the walk is at any point.
- E.g., (000…1…000) means we're in state *i*.

1 *i* *n*

More generally, the vector **x** = $(x_1, \ldots x_n)$ means the walk is in state *i* with probability x_i.

$$\sum_{i=1}^{n} x_i = 1.$$

Change in probability vector

- If the probability vector is $\mathbf{x} = (x_1, \ldots x_n)$ at this step, what is it at the next step?
- Recall that row *i* of the transition prob. matrix **P** tells us where we go next from state *i*.
- So from **x**, our next state is distributed as **xP**.

Steady state example

- The steady state is simply a vector of probabilities

  a = $(a_1, \ldots a_n)$:

  - a_i is the probability that we are in state *i*.
  - a_i is the long-term visit rate (or pagerank) of state (page) *i*.
  - So we can think of pagerank as a long vector, one entry for each page

How do we compute this vector?

- Let **a** = $(a_1, \ldots a_n)$ denote the row vector of steady-state probabilities.
- If our current position is described by **a**, then the next step is distributed as **aP**.
- But **a** is the steady state, so **a**=**aP**.
- Solving this matrix equation gives us **a**.
  - So **a** is the (left) eigenvector for **P**.
  - (Corresponds to the "principal" eigenvector of **P** with the largest eigenvalue.)
  - Transition probability matrices always have largest eigenvalue 1.

One way of computing

- Recall, regardless of where we start, we eventually reach the steady state **a**.
- Start with any distribution (say **x**=(*10…0*)).
- After one step, we're at **xP**;
- after two steps at $\mathbf{xP}^2$, then $\mathbf{xP}^3$ and so on.
- "Eventually" means for "large" k, $\mathbf{xP}^k = \mathbf{a}$.
- Algorithm: multiply **x** by increasing powers of **P** until the product looks stable.

- This is called the power method

Power method: Example

- Two-node example: $\vec{x} = (0.5, 0.5)$, $P = \begin{pmatrix} 0.25 & 0.75 \\ 0.25 & 0.75 \end{pmatrix}$
- $\vec{x}P = (0.25, 0.75)$
- $\vec{x}P^2 = (0.25, 0.75)$
- Convergence in one iteration!

Pagerank summary

- Preprocessing:
  - Given graph of links, build matrix **P**.
  - From it compute **a**.
  - The entry a_i is a number between 0 and 1: the pagerank of page *i*.
- Query processing:
  - Retrieve pages meeting query.
  - Rank them by their pagerank.
  - Order is query-*independent*.
- **In practice, pagerank alone wouldn't work**
- Google paper: http://infolab.stanford.edu/~backrub/google.html