Finite Automata

Reading: Chapter 2

Finite Automata

- Informally, a state machine that comprehensively captures all possible states and transitions that a machine can take while responding to a stream (or sequence) of input symbols
- Recognizer for "Regular Languages"

- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

Deterministic Finite Automata - Definition

- A Deterministic Finite Automaton (DFA) consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of final states
  - δ ==> a transition function, which is a mapping between Q x ∑ ==> Q
- A DFA is defined by the 5-tuple:
  - $\{Q, \Sigma, q_0, F, \delta\}$

How to use a DFA?

- Input: a word w in Σ^*
- Question: Is w acceptable by the DFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Compute the next state from the current state, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed, the current state is one of the final states (F) then *accept w;*
  - Otherwise, *reject w.*

Regular Languages

- Let L(A) be a language *recognized* by a DFA A.
  - Then L(A) is called a "*Regular Language*".

- Locate regular languages in the Chomsky Hierarchy

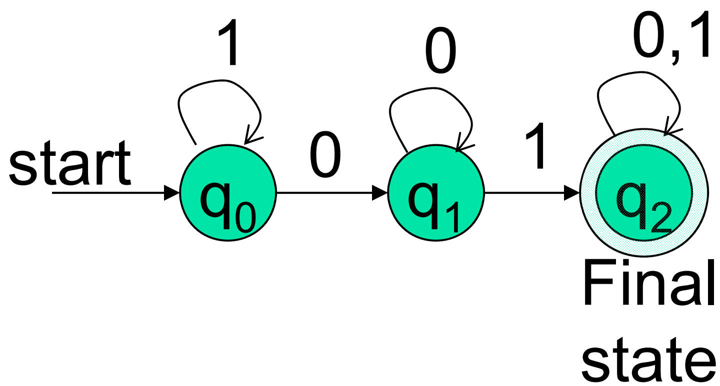
Example #1

- Build a DFA for the following language:
  - L = {w | w is a binary string that contains 01 as a substring}
- Steps for building a DFA to recognize L:
  - $\Sigma = \{0,1\}$
  - Decide on the states: Q
  - Designate start state and final state(s)
  - δ: Decide on the transitions:
- Final states == same as “accepting states”
- Other states == same as “non-accepting states”

Regular expression: (0+1)*01(0+1)*

DFA for strings containing 01

- What makes the DFA deterministic?



- What if the language allows empty strings?

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

symbols

δ	0	1
→ q_0	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_2	q_2

states

Example #2

- Clamping Logic:
  - A clamping circuit waits for a ”1” input, and turns on forever. However, to avoid clamping on spurious noise, we’ll design a DFA that waits for *two consecutive 1s* in a row before clamping on.
  - Build a DFA for the following language:
    L = { w | w is a bit string which contains the substring 11}
- State Design:
  - q_0 : start state (initially off), also means the most recent input was not a 1
  - q_1: has never seen 11 but the most recent input was a 1
  - q_2: has seen 11 at least once

Example #3

- Build a DFA for the following language:

  L = { w | w has an even number of 0s and an even number of 1s}

  Note: Alphabet implied is {0,1}
- ?

Extension of transitions (δ) to Paths ($\hat{\delta}$)

- $\hat{\delta}\ (q_0,w)$ = ending state of the path taken from q_0 on input string *w*

- $\hat{\delta}\ (q_0,wa) = \delta\ (\hat{\delta}(q_0,w), a)$

- Exercise:
  - Work out example #3 using the input sequence w=10010, a=1

Language of a DFA

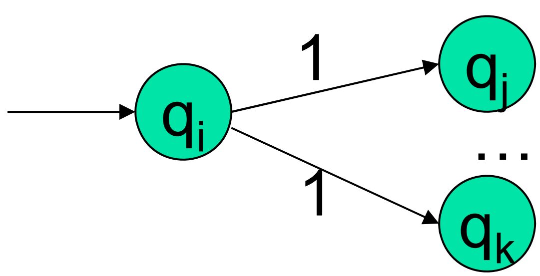
- A DFA A accepts *w* if there is exactly a path from q_0 to an accepting (or final) state that is labeled by *w*

- *i.e.,* $L(A) = \{ w \mid \hat{\delta}(q_0, w) \in F \}$

- *I.e., L(A) = all strings that lead to a final state from* q_0

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA)
  - is of course "non-deterministic"
    - Implying that the machine can exist in more than one state at the same time
    - Outgoing transitions could be non-deterministic



• Each transition function therefore maps to a set of states

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA) consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of final states
  - δ ==> a transition function, which is a mapping between Q x ∑ ==> subset of Q
- An NFA is also defined by the 5-tuple:
  - $\{Q, \Sigma, q_0, F, \delta\}$

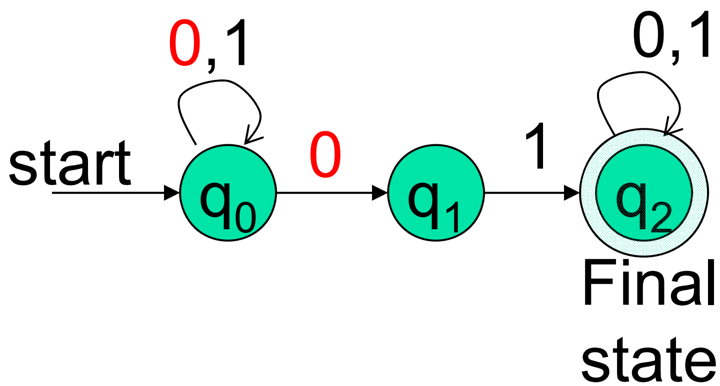
How to use an NFA?

- Input: a word w in Σ^*
- Question: Is w acceptable by the NFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Determine all the possible next states from the current state, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed, at least one of the current states is a final state then *accept w;*
  - Otherwise, *reject w.*

Regular expression: (0+1)*01(0+1)*

NFA for strings containing 01

Why is this non-deterministic?



What will happen if at state q_1 an input of 0 is received?

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

symbols

δ	**0**	**1**
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$

states

Example #2

- Build an NFA for the following language:
  L = { w | w ends in 01}
- ?
- How about a DFA now?

Advantages & Caveats for NFA

- Great for modeling regular expressions
- String processing
  - e.g., grep, lexical analyzer
- But “imaginary”, in the sense that it has to be implemented deterministically in practice
- Could a non-deterministic state machine be implemented in practice?
  - E.g., toss of a coin, a roll of dice

Differences: DFA vs. NFA

- **DFA**

1. All transitions are deterministic
   - Each transition leads to exactly one state
2. For each state, transition on all possible symbols (alphabet) should be defined
3. Accepts input if the last state is in F
4. Sometimes harder to construct because of the number of states
5. Practical implementation is feasible

- **NFA**

1. Transitions could be non-deterministic
   - A transition could lead to a subset of states
2. For each state, not all symbols necessarily have to be defined in the transition function
3. Accepts input if *one of* the last states is in F
4. Generally easier than a DFA to construct
5. Practical implementation has to be deterministic (so needs conversion to DFA)

But, DFAs and NFAs are equivalent (in their power) !!

Extension of δ to NFA Paths

- Basis: $\hat{\delta}(q,\varepsilon) = \{q\}$

- Induction:
  - Let $\hat{\delta}(q_0,w) = \{p_1,p_2\ldots,p_k\}$
  - $\delta(p_i,a) = S_i \quad$ *for* $i=1,2\ldots,k$
  - *Then,* $\hat{\delta}(q_0,wa) = S_1 \cup S_2 \cup \ldots \cup S_k$

Language of an NFA

- An NFA accepts *w* if *there exists at least one* path from the start state to an accepting (or final) state that is labeled by *w*
- $L(N) = \{ w \mid \hat{\delta}(q_0,w) \cap F \neq \Phi \}$

Equivalence of DFA & NFA

- Theorem:
  - A language L is accepted by a DFA *if and only if* it is accepted by an NFA.

Should be true for any L

- Proof:
  1. If part:
     - Proof by subset construction (in the next few slides…)
  2. Only-if part:
     - Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA.

☐

Proof for the if-part

- If-part: A language L is accepted by a DFA if it is accepted by an NFA
- rephrasing…
- Given any NFA N, we can construct a DFA D such that L(N)=L(D)
- How to construct a DFA from an NFA?
  - Observation: the transition function of an NFA maps to *subsets* of states
  - Idea: Make one DFA state for every possible subset of the NFA states

Subset construction

NFA to DFA by subset construction

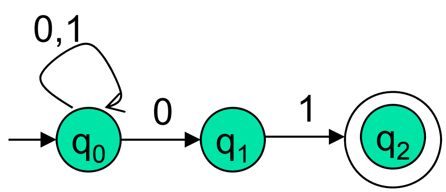
- Let $N = \{Q_N, \sum, \delta_N, q_0, F_N\}$
- <u>Goal:</u> Build $D=\{Q_D, \sum, \delta_D, \{q_0\}, F_D\}$ s.t. $L(D)=L(N)$
- <u>Construction:</u>
  1. Q_D= all subsets of Q_N (i.e., power set)
  2. F_D=set of subsets S of Q_N s.t. $S \cap F_N \neq \Phi$
  3. δ_D: for each subset S of Q_N and for each input symbol a in $\sum$:
     - $\delta_D(S,a) = \bigcup_{p \text{ in } s} \delta_N(p,a)$

Idea: To avoid enumerating all of power set, do "lazy creation of states"

NFA to DFA construction: Example

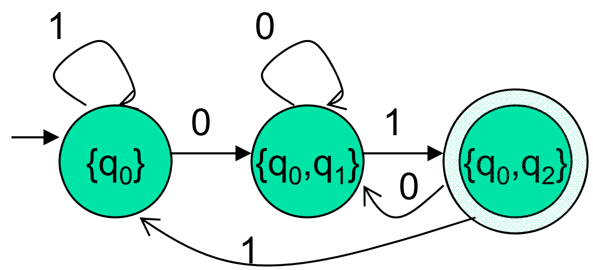
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
$\rightarrow q_0$	$\{q_0,q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

DFA:



δ_D	0	1
~~$\emptyset$~~	~~$\emptyset$~~	~~$\emptyset$~~
$\rightarrow\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
~~$\{q_1\}$~~	~~$\emptyset$~~	~~$\{q_2\}$~~
~~$*\{q_2\}$~~	~~$\emptyset$~~	~~$\emptyset$~~
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$
~~$*\{q_1,q_2\}$~~	~~$\emptyset$~~	~~$\{q_2\}$~~
~~$*\{q_0,q_1,q_2\}$~~	~~$\{q_0,q_1\}$~~	~~$\{q_0,q_2\}$~~

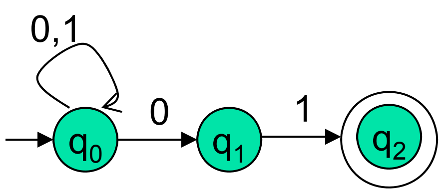
δ_D	0	1
$\rightarrow\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

Remove states unreachable from q_0

NFA to DFA: Repeating the example using *LAZY CREATION*

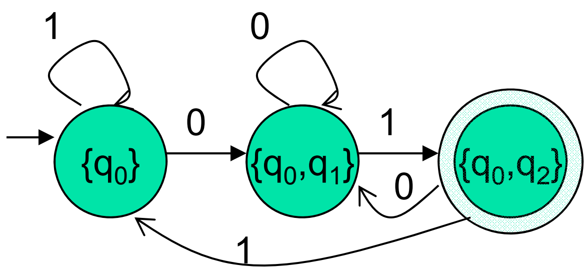
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$^*q_2$	Ø	Ø

DFA:



δ_D	0	1
→ $\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$^*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

Main Idea:
Introduce states as you go (on a need basis)

Correctness of subset construction

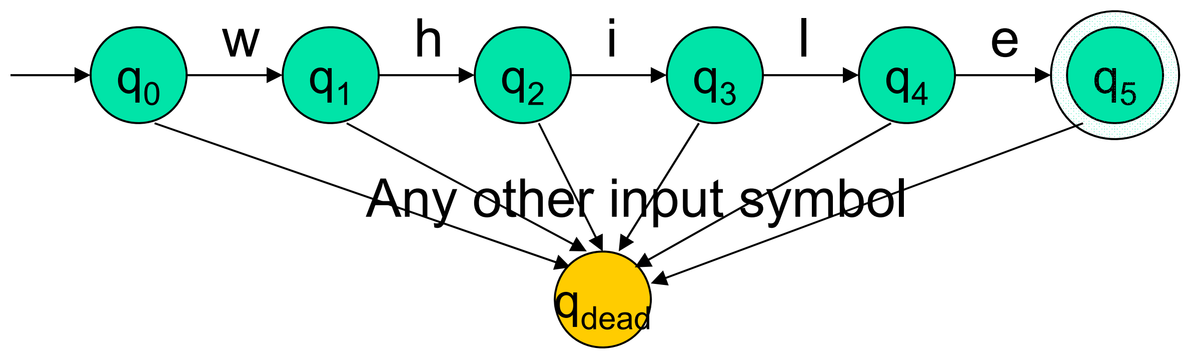
- Theorem: If D is the DFA constructed from NFA N by subset construction, then L(D)=L(N)
- Proof:
  - Show that $\hat{\delta}_D(\{q_0\},w) = \hat{\delta}_N(q_0,w\}$
  - Using induction on w's length:
    - Let w = xa
    - $\hat{\delta}_D(\{q_0\},xa) = \delta_D(\hat{\delta}_N(q_0,x\}, a) = \hat{\delta}_N(q_0,w\}$

A bad case for subset construction

- L = {w | w is a binary string s.t., the n^{th} symbol from its end is a 1}
  - NFA has n+1 states
  - DFA needs to have at least 2^n states
- <u>Pigeon hole principle</u>
  - *m* holes and >*m* pigeons
    - => at least one hole has to contain two or more pigeons

Dead states

- Example:
  - A DFA for recognizing the key word "*while*"



Applications

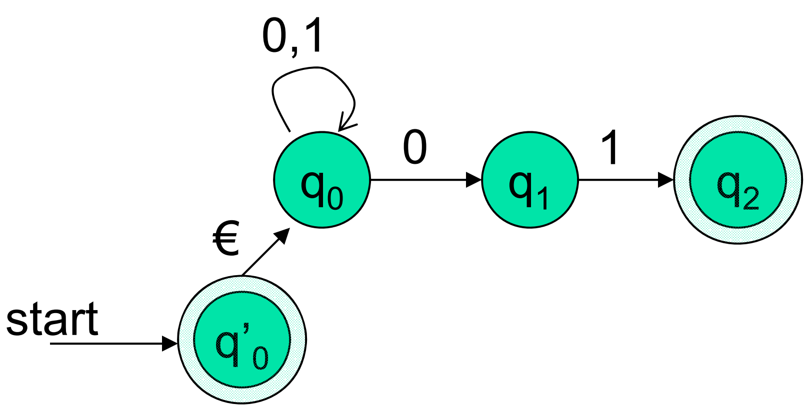
- Text indexing
  - inverted indexing
  - For each unique word in the database, store all locations that contain it using an NFA or a DFA
- Find pattern P in text T
  - Example: Google querying
- Extensions of this idea:
  - PATRICIA tree, suffix tree

FA with Epsilon-Transitions

- We can allow ε-transitions in finite automata
  - i.e., a state can jump to another state without consuming any input symbol
- Use:
  - Makes it easier sometimes for NFA construction
- ε -NFAs have to one more column in their transition table

Example of an ε-NFA

L = {w | possibily empty w s.t. if non-empty will end in 01}



δ_E	0	1	ε
$\rightarrow$ *q'_0	Ø	Ø	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
*q_2	Ø	Ø	$\{q_2\}$

ECLOSE(q'_0)

ECLOSE(q_0)

- ε-closure of a state q, ***ECLOSE(q)***, is the set of all states that can be *reached* from q by repeatedly making ε-transitions (including itself).

Equivalency of DFA, NFA, ε-NFA

- Theorem: A language L is accepted by some ε-NFA if and only if L is accepted by some DFA
- Proof:
  - Idea: Use the same subset construction except include ε-closures

Eliminating ε-transitions

- Let $E = \{Q_E, \sum, \delta_E, q_0, F_E\}$
- Goal: Build $D=\{Q_D, \sum, \delta_D, \{q_D\}, F_D\}$ s.t. $L(D)=L(E)$
- Construction:
  1. Q_D= all reachable subsets of Q_E factoring in €-closures
  2. q_d = ECLOSE(q_0)
  3. F_D=subsets S in Q_D s.t. $S \cap F_E \neq \Phi$
  4. δ_D: for each subset S of Q_E and for each input symbol a in $\sum$:
     - Let $R = \bigcup_{p \text{ in } s} \delta_E(p,a)$
     - $\delta_D(S,a) = \bigcup_{r \text{ in } R} \text{ECLOSE}(r)$

Summary

- DFA
  - Definition
  - Transition diagrams & tables
- Regular language
- NFA
  - Definition
  - Transition diagrams & tables
- DFA vs. NFA
- NFA to DFA conversion using subset construction
- Equivalency of DFA & NFA
- Removal of redundant states and including dead states
- ε-transitions in NFA
- Pigeon hole principles
- Text searching applications