Lecture 4: Nondeterminism

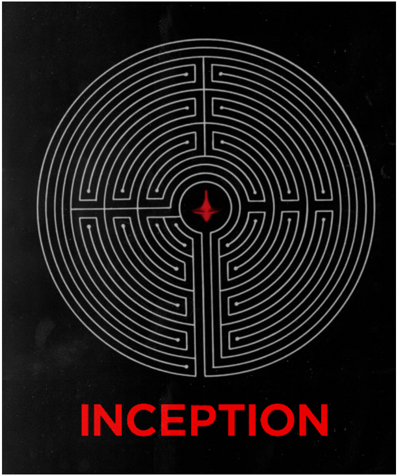
Michael Engling

Department of Computer Science
Stevens Institute of Technology

CS 334 Automata and Computation
Fall 2015

OUTLINE: Nondeterministic Finite Automata

Labyrinth vs. Maze



Determinism

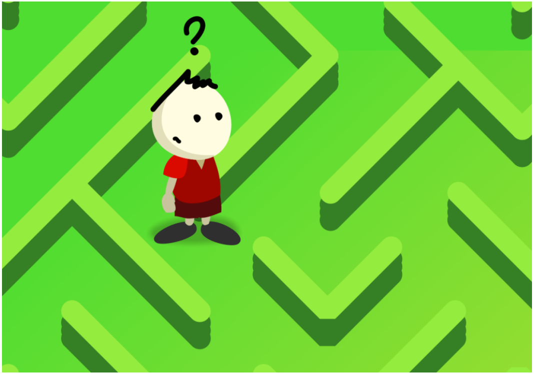
Nondeterminism

Labyrinth vs. Maze

Veriditas Labyrinth

Overlook Hotel Maze

Nondeterminism



The Essence of Nondeterminism

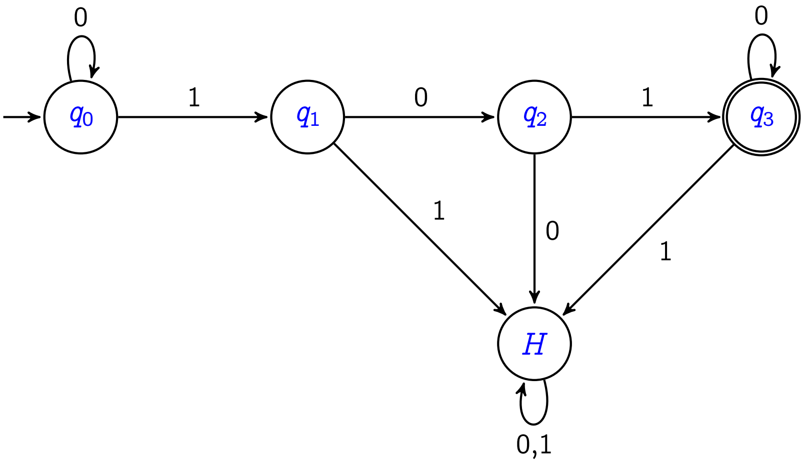
Decisions/Choices

Dead Ends

Potential
of
Nontermination

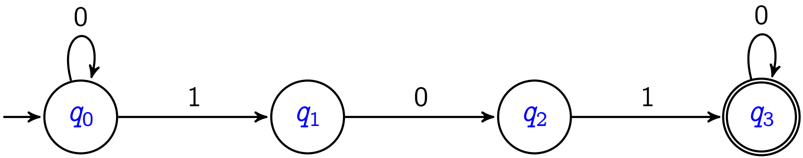
Complexity

DFA for 0^*1010^*



It is easy to verify that this DFA recognizes $L_1 = 0^*1010^*$

NFA 1 recognizing 0^*1010^*



But could we not have simply stopped here?

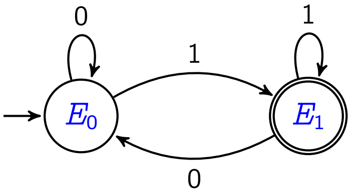
It is even easier to verify that this NFA recognizes $L_1 = 0^*1010^*$

Is this not more elegant?

So we introduce some dead ends?

Do we truly lose anything? (Hint: What is $\overline{L}$?)

DFA for Σ^*1



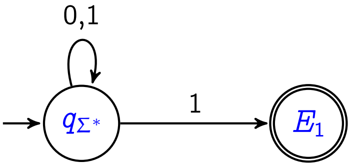
It is easy to verify that this DFA recognizes Σ^*1

All strings ending in 0 finish in state E_0.

All strings ending in 1 finish in state E_1 (and are accepted).

What about that pesky empty string, ε?

NFA for Σ^*1



It is even easier to verify that this NFA recognizes Σ^*1

It practically creates itself.

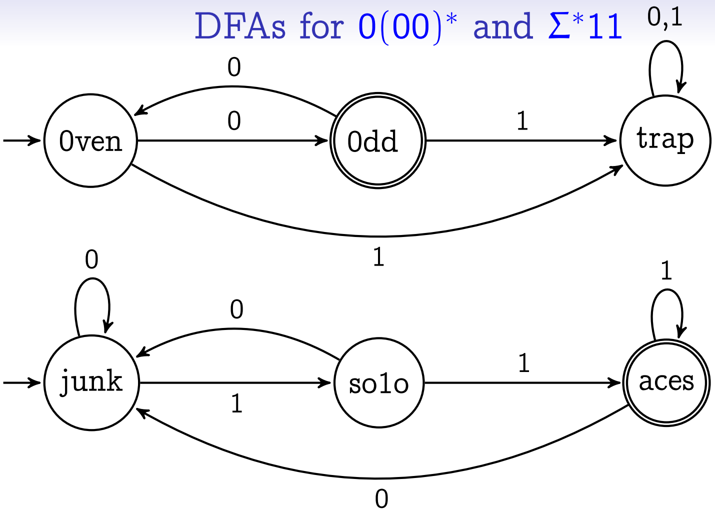
So we introduce *choice*.

Where does the string 0101101 finish?

Do we accept or not?

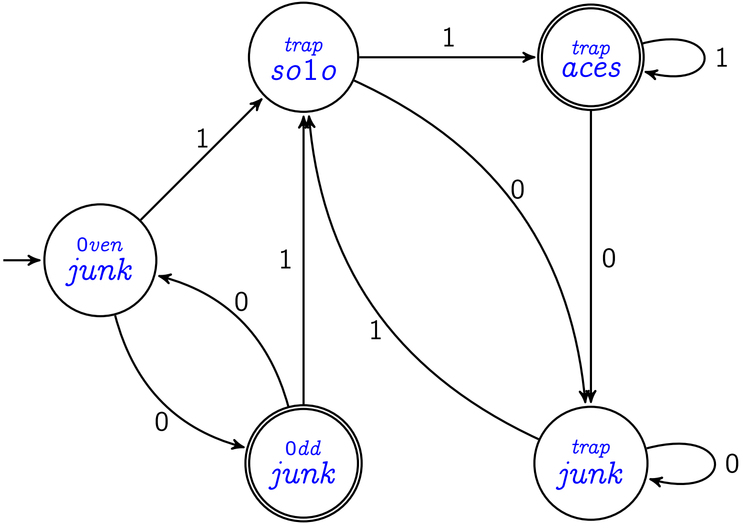
Nondeterminism in a nutshell.

DFAs for $0(00)^*$ and Σ^*11

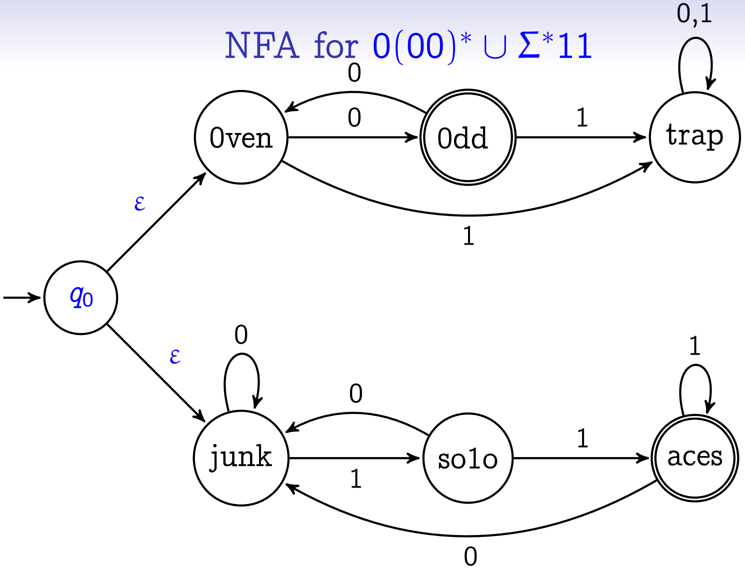


Can we put these together somehow to create an NFA that recognizes $0(00)^* \cup \Sigma^*11$ more elegantly and expressively than the DFA we created last time?

DFA for $0(00)^* \cup \Sigma^*11$



NFA for $0(00)^* \cup \Sigma^*11$



An ε-edge may be traversed without “spending” an input symbol.

Where does 001 terminate?

Definition of Computation with NFAs
Regular Languages

We can formalize our notion of *computation* with an NFA much the way we did for the simpler DFAs:
If $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA, and w is a string over Σ, then M *accepts* w if we can write it as $w = y_1 y_2 ... y_n$ where $y_i \in \Sigma_\varepsilon \quad \forall i \in \{0, 1, ... n\}$ and a sequence of states $r_0 r_1 ... r_n$ exists in Q such that:

1. $r_0 = q_0$ (We start in the start state.),
2. $\delta(r_i, y_{i+1}) = r_{i+1} \quad \forall i \in \{0, 1, ..., n-1\}$, and
3. $r_n \in F$ (We end in a final (accepting) state.)

Definition: A language is a called a **regular language** if some finite automaton recognizes it (DFA *or* NFA).

NFA Formally

Not surprisingly, our formal definition of an NFA looks much like that for a DFA:

Q is a finite set of states
Σ is a finite alphabet
$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is our transition function
$q_0 \in Q$ is the start state
$F \subseteq Q$ is the set of final (accepting) states

The only change is in our transition function.
$\Sigma_\varepsilon = \Sigma \cup \varepsilon$ (This permits ε-edges.)

And we map to the power set of Q. Instead of having each symbols map (deterministically) to a unique state, we allow it to map to a (possibly empty–dead end) *set* of states.
That's nondeterminism.

Concatenation Constructively

Suppose we have two DFAs computing L_1 and L_2, and we wish to compute the concatenation of these languages $L_1 \circ L_2$, that is, the set of all strings $w = w_1 w_2$ where $w1 \in L_1$ and $w_2 \in L_2$.

Constructing a DFA to do this, much like the construction we did for the union/intersection of two languages, would be an arduous task.

But constructing an NFA to recognize the concatenation is child's play:

Let the start state of our NFA be the start state of the DFA recognizing L_1. Next, connect via ε-edges all the final states of this DFA to the start state of the DFA recognizing L_2. Finally, let the final states of this DFA be the final states of our NFA. Such a construction proves that $L_1 \circ L_2$ is a regular language for any regular languages L_1 and L_2.

Concatenation $L_1 \circ L_2$ Formally

Given $A_1 = (Q_1, \Sigma, \delta_1, q_0^{A_1}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_0^{A_2}, F_2)$ We

can form "$A_1 \circ A_2$" recognizing $L_1 \circ L_2$:

$$
\begin{array}{lll}
A_1 \circ A_2 = \text{(as a 5-tuple)} & & \\
(Q = Q_1 \cup Q_2, & \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\
\Sigma = \Sigma_\varepsilon & \delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\
\delta(q, a) = & q_2 & q \in F_1 \text{ and } a = \varepsilon \\
q_0 = q_1 & \delta_2(q, a) & q \in Q_2 \\
F = F_2) & &
\end{array}
$$

(Here we assume the only ε-edges added are those that connect the final states of A_1 to the start state of A_2.)

Thus $L_1 \circ L_2$ is a regular language.

$P \stackrel{?}{=} NP$

The most important unsolved problem in computer science is probably the $P \stackrel{?}{=} NP$ question.

Basically, it asks: If the solution to a problem can be *checked* in polynomial time, can the problem be *solved* in polynomial time?

The idea is expressed in 5 symbols:
You know what a question mark means.
You know what the equal sign is and means.
The two "P"s stands for 'polynomial'–you know what that means.

Care to guess what the N stands for?