



Non-Deterministic Finite-State Automata

Robert M. Keller
Harvey Mudd College
6 September 2003



What is Non-Determinism?

- ❑ Can an automaton have choice or free-will?
- ❑ Non-determinism supposes it can.



Uses of Non-Determinism?

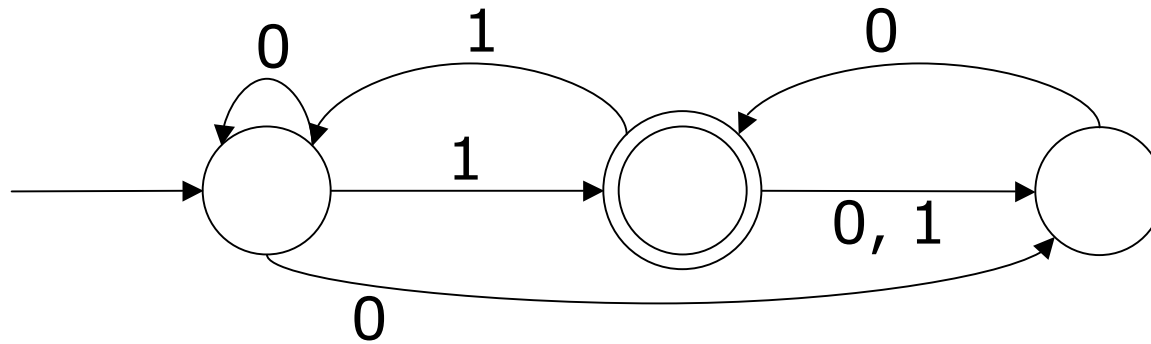
- ❑ Modeling asynchronous events, wherein we don't know the order in which things happen.
- ❑ Modeling certain kinds of parallel computing.
- ❑ As a mathematical construction device.



NFA's

- ❑ NFA = Non-deterministic finite-state automaton
- ❑ From any given state, we can have **more than one** transition for the same letter.
- ❑ From any given state, we can have **no** transitions for a given letter.
- ❑ We can have free transitions that occur without using any letter. (These are labeled with ϵ .)
- ❑ We can have multiple starting states, freely chosen.

Random Example of an NFA

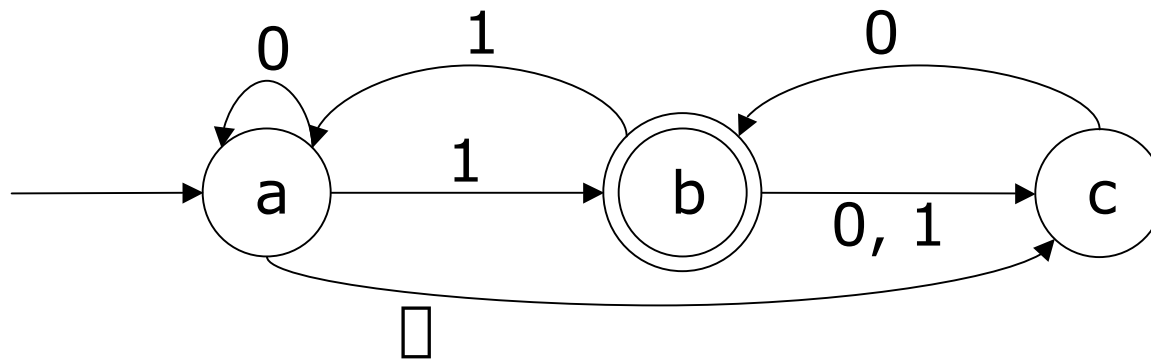




Paths in an NFA

- Let x be a **string** over the alphabet of N .
- Let q and q' be states of N .
- We say there is a “**path from q to q' with label x** ” provided that there is a series of arcs connecting q to q' and the labels on that series, when concatenated, form x .
- Note that ϵ -transitions can be concatenated and do not change the string to which they are concatenated.

Paths in an NFA (not identical to previous)



Some Paths

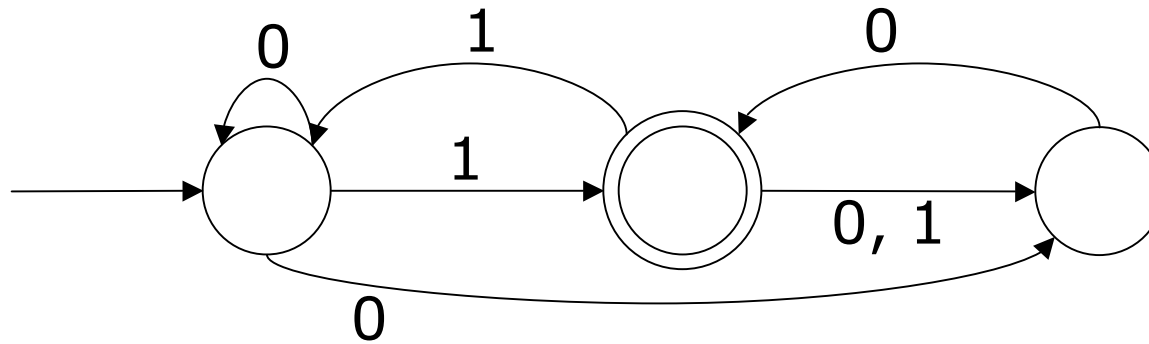
- 1 from a to b
- \square from a to c
- 0 from a to c
- 00 from a to b
- 11 from a to a
- 1 from b to c
- 010 from a to a
- ...



Acceptance by an NFA

- An NFA accepts a string x if there is **a** path from **some** start state to **some** accepting state with label x .
- If there is no such path, then the NFA does not accept the string.
- The **language** accepted by an NFA is the set of all strings accepted by the NFA.

Original Example of an NFA



Strings accepted:

1
00
01
000
001
100
110
111
...

Strings not accepted:

□
0
10
11
010
011
101
...



Observation

- A DFA can be viewed as a special case of an NFA.
- If L is a language accepted by a DFA, then L is also accepted by a NFA.



The NFA Theorem

- If L is accepted by an NFA, there is also a DFA that accepts L .



Proof of the NFA Theorem (called the “Subset Construction”)

- Suppose we have an NFA N . We want to construct a DFA D that accepts the same language exactly.
- Let S be the set of states of N . Choose as the states of D the set $P(S) =$ the **set of all subsets** of the states of N . Since S is finite, so is $P(S)$.
- (In practice, we won't usually need **all subsets**.)

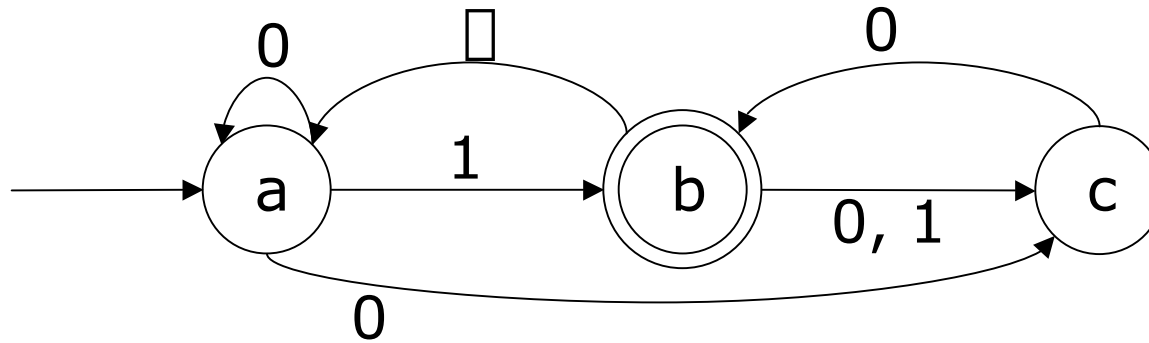


Transitions of D

- Let R be a state of D . So $R \in S$.
- We must define a transition from R for each symbol σ in our alphabet.
- To designate the **next state** from R given symbol σ , use:

$$R_\sigma = \{q' \in S \mid (\exists q \in R) \exists \text{ path from } q \text{ to } q' \text{ with label } \sigma\}$$

Examples of DFA Transitions



$$\{a, b\}_0 = \{a, c\}$$

$$\{a, b\}_1 = \{a, b, c\}$$

$$\{b\}_0 = \{a, c\}$$

$$\{b\}_1 = \{c\}$$

$$\{c\}_0 = \{a, b\}$$

$$\{c\}_1 = \emptyset$$

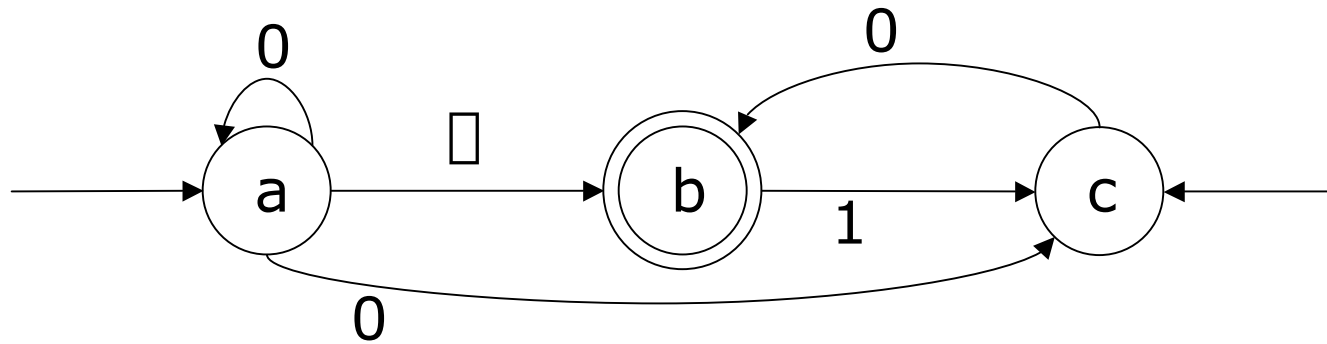


Initial State of D

- Letting S_0 be the set of initial states of N , the initial state of D is defined to be the subset

$$\{q' \in S \mid (\exists q \in S_0) \exists \text{ path from } q \text{ to } q' \text{ with label } \lambda\}$$

Examples of Initial State



The initial state of D is $\{a, b, c\}$.



Conclusion of the NFA Theorem

- The preceding construction shows how we capture all paths of the original NFA N in a DFA D .
- The only part remaining is defining the **accepting states** of D . These will be the subsets of S that **contain at least one accepting state** of N .



Interpretation of the DFA Construction

- The DFA constructed in the proof of the theorem can be interpreted as an automaton that simulates all possible choices of the NFA **in parallel**.
- This idea will have at least one other application (to be seen).



A Computational Shortcut

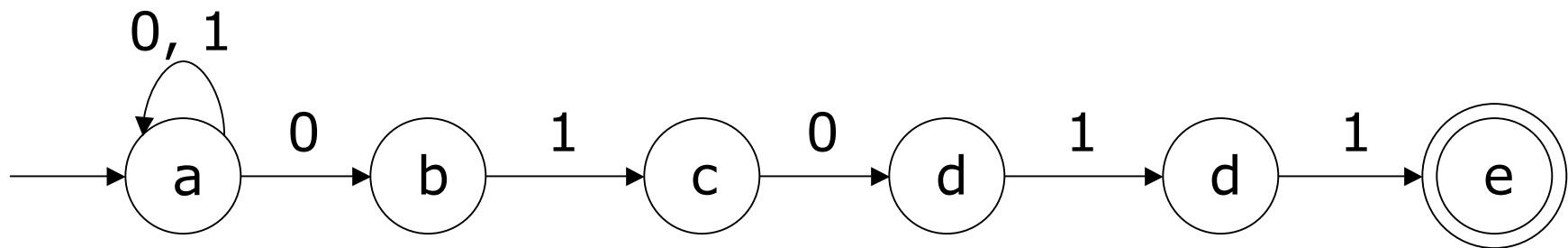
- By beginning with the initial state of the DFA and working from there, we can eliminate the need to generate all states in $P(S)$ in most cases.
- In other words, we need only generate the states **reachable from** the initial state of D .



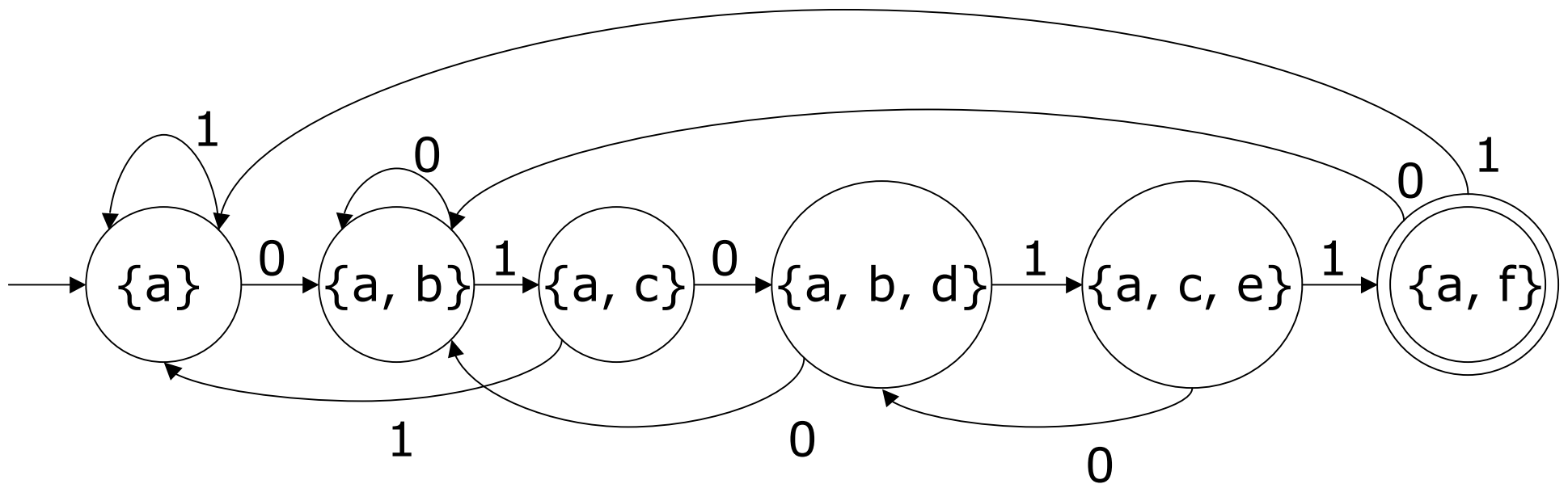
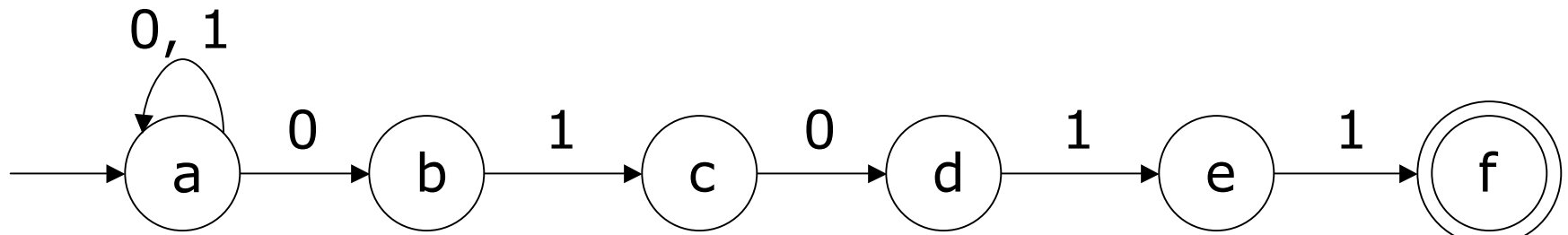
Application of the Construction

- Suppose we want to construct a DFA that will accept the language of **strings ending with a given string**, for example: 01011.
- The tricky part here is that if we get as input, for example, 01010, while this is not accepted, the last part 010 can possibly be used as the initial part of a string that is, for example 0101011.
- By constructing an appropriate NFA and converting it, it is easy to get the DFA right.

An NFA for $\{0, 1\}^* 01011$



Constructing a DFA for $\{0, 1\}^* 01011$





Notes on the Previous Example

- ❑ Not all subsets were reachable. Those that weren't were not generated.
- ❑ The number of states is the same. This is a coincidence; it may be more or fewer.
- ❑ The transition structure is more complex in the DFA. This is typical.



Algorithmic Application

- The principle underlying the illustrated method for text matching was the insight and basis for two text searching algorithms:
 - **Knuth-Morris-Pratt:** Search for a single string
 - **Aho-Corasick:** Search for a finite set of strings



Another Possibility for Search

- ❑ Rather than go through the DFA construction and simulate the result to do search a text, it is possible to simulate the NFA directly.
- ❑ In this case there would be a **set** of “current states” rather than a single one.
- ❑ Just use **multiple state pointers** for the simulation rather than a single one.



More applications of the subset construction are forthcoming.

- Closure properties
- DFA from regular expressions



Families of Languages

- Let F be a **family** (set) of languages.
- Examples:
 - The family of finite languages.
 - The family of regular languages.
 - The family of co-finite languages (complement within Σ^* is finite).
 - The family of all languages.

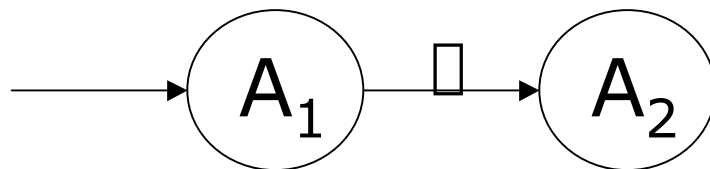


Closure Properties

- A family F is **closed under** an operator if the application of that operator to a language or languages in F results in a language which is also in F .
- Example: The family of regular languages is closed under \cup , concatenation, and $*$.
- Exercise: Determine whether the other families on the preceding page are closed under these same operators. (Create a matrix.)

Proof of Some Closure Properties using the Subset Construction

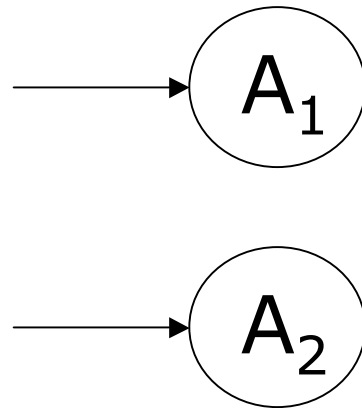
- The family of languages accepted by DFA (or NFA; we know they are the same) is closed under **concatenation**.
- **Proof:** Suppose L_1 and L_2 are accepted by DFA's. To show that $L_1 L_2$ is also, connect the corresponding acceptors A_1 and A_2 by adding \square transitions from each accepting state of A_1 to each start state of A_2 . Then modify the start and accepting states appropriately and convert this NFA to a DFA. **Figuratively,**





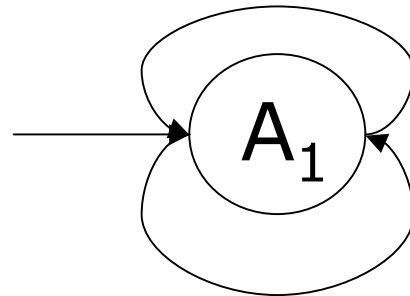
Proof of Some Closure Properties using the Subset Construction

- Show that the family of DFA languages is closed under union. Hint:



Proof of Some Closure Properties using the Subset Construction

- Show that the family of DFA languages is closed under $*$. Hint:





Summary

- The family of languages accepted by DFA's is closed under concatenation, union, and $*$.



Proof that every regular language is accepted by a DFA.

- The family of languages accepted by DFA's is closed under concatenation, union, and $*$. These correspond exactly to the regular operators.
- Furthermore, the languages of a single 1-letter string are accepted by DFA's, as are \emptyset and Σ .
- Therefore every regular language is accepted by a DFA.



Kleene's Theorem

- A language is regular iff it is accepted by some DFA.

- Proof:


- We showed DFA \Rightarrow regular by solving a system of equations.

- We showed regular \Rightarrow DFA by the subset construction and closure properties.



The family of DFA languages is closed under complement.

- For complement, we only need reverse the roles of accepting and non-accepting states in the accepting automaton.
- Thus the family of regular languages is closed under complement, even though complement is not a regular operator. (It is sometimes seen in an **extended** version of regular expressions.)

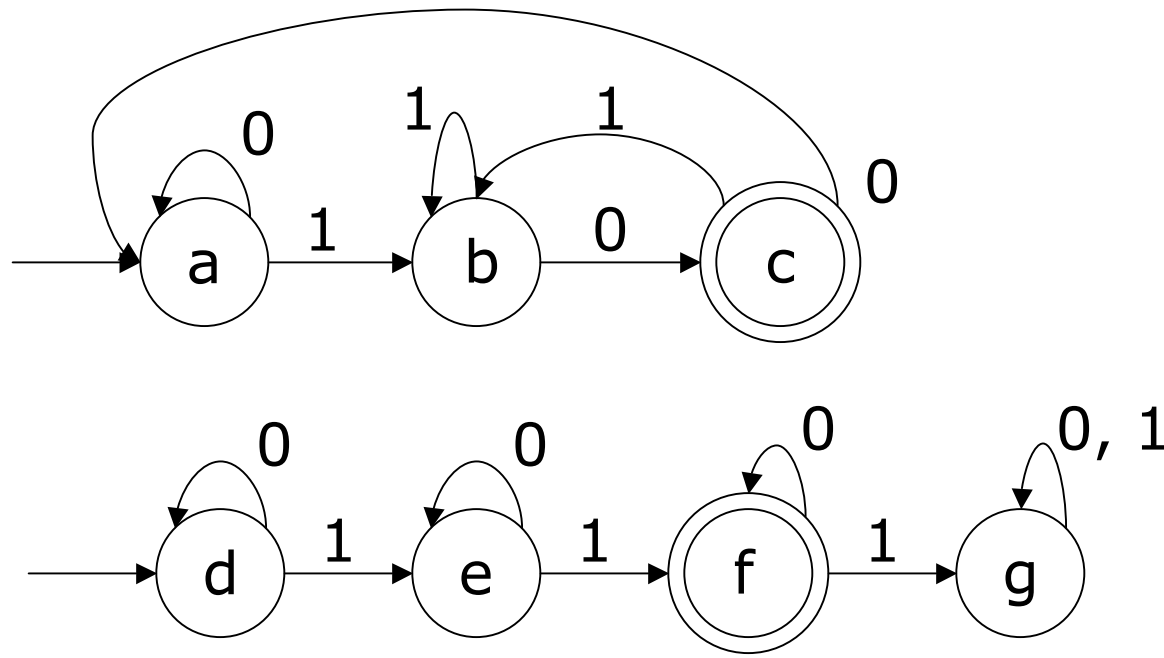


The family of DFA languages is closed under intersection.

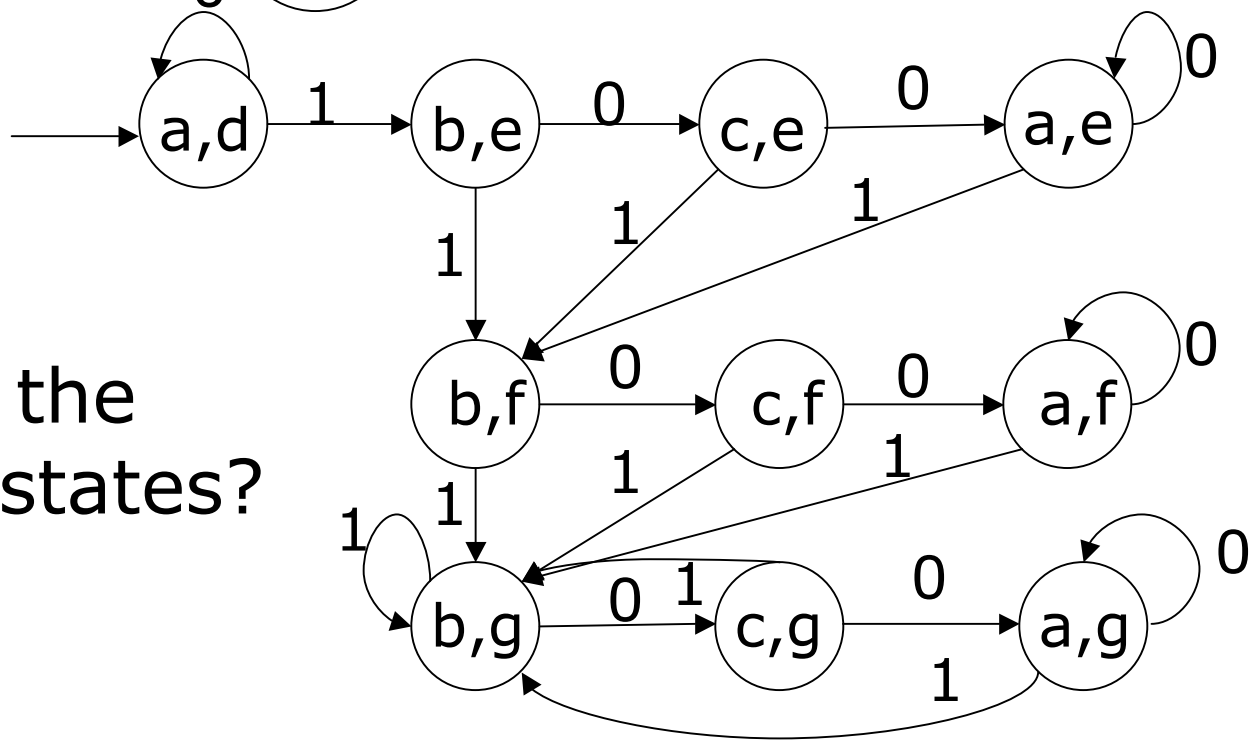
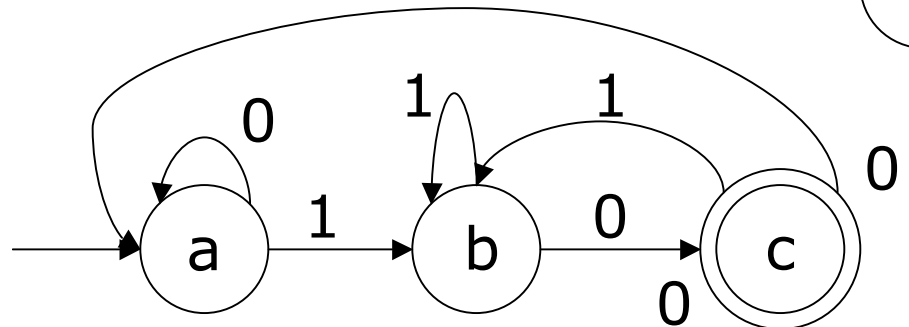
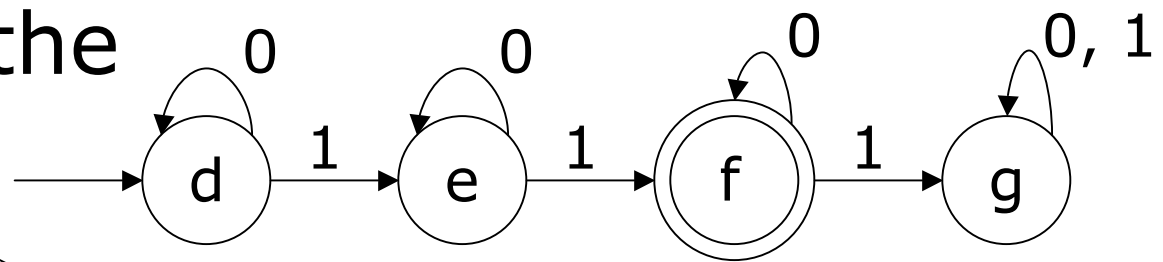
- Proof: Adjoin the two automata and use the subset construction with the set of two start states as the start state. This effectively constructs an automaton, called the **product automaton**, that simulates the behavior of both of the original automata in parallel.
- Choose accepting states appropriately. The new automaton will automatically be deterministic, as the originals were.
- Thus the family of regular languages is closed under intersection, although \cap is not a regular operator.
- The product automaton can be used for any binary set operator (\cap , \cup , $-$, \oplus , \equiv , etc.). Only the accepting states are different.

Example: A DFA for the intersection of languages given by regular expressions $(0 \square 1)^* 10$ and $0^* 10^* 10^*$

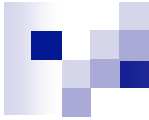
The individual DFA are:



Constructing the product DFA



Which are the accepting states?



Notes on Product Construction

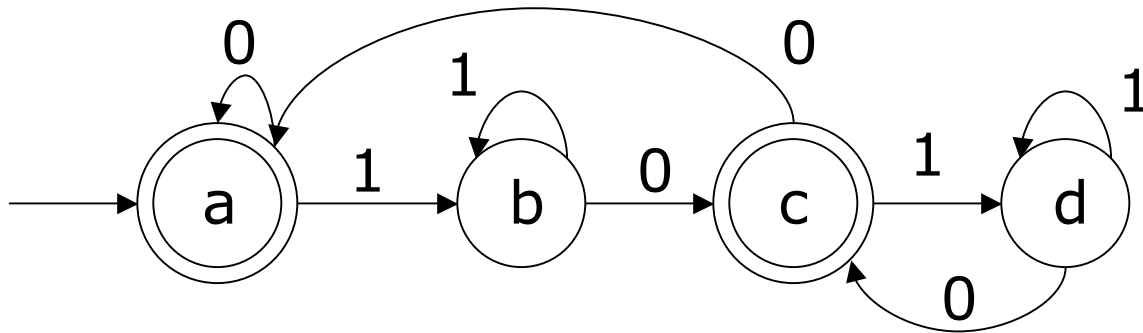
- ❑ Depending on the intended implementation, it may be better **not** to construct the composite machine explicitly, but rather leave it decomposed.
- ❑ The rationale is that there are generally fewer states in the sum of the two machines than in the composite.
- ❑ We sometimes try to go the other way: **decompose** a complex machine into a product of simpler machines.



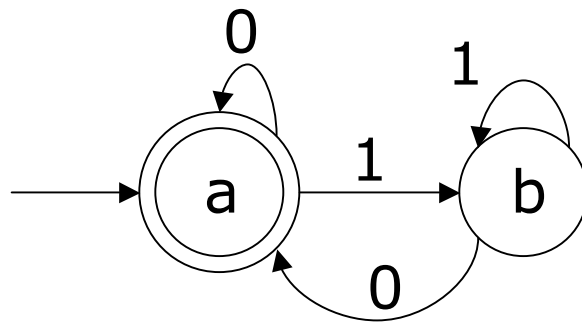
Equivalence of States

- ❑ For any state, not just the start state, we can define a **language accepted from** that state.
- ❑ Two states are equivalent if their languages are equal.
- ❑ If an automaton contains distinct equivalent states, those states can be “merged” to get a simplified automaton with fewer states.

Example of State Equivalence



In this example, state a is equivalent to state c and state b is equivalent to state d. The simplified automaton is:

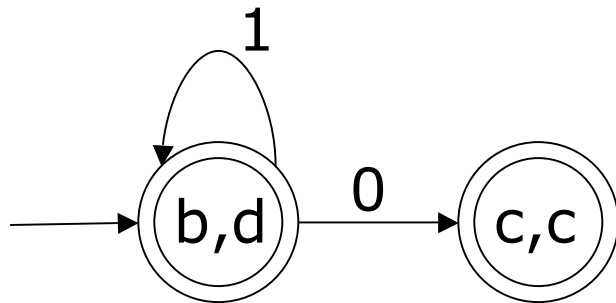
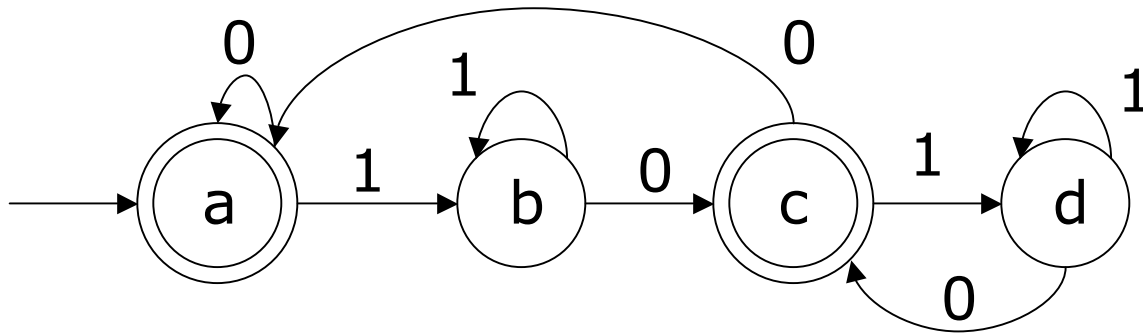




Equivalence Testing

- ❑ How can we test whether two states are equivalent or not?
- ❑ One way: Think of \equiv as a binary operator, just like \cap , \cup , etc. Construct the product machine for two automata, one having each of the two states as starting states. Accepting states are those with both pairs accepting or both pairs non-accepting.
- ❑ The two states are equivalent iff the product machine accepts Σ^* .

Example: Equivalence Testing of b and d:



Note: Once we get to a state where both components are the same, we don't need to complete that branch of the construction because the graph will be that of the original machine.

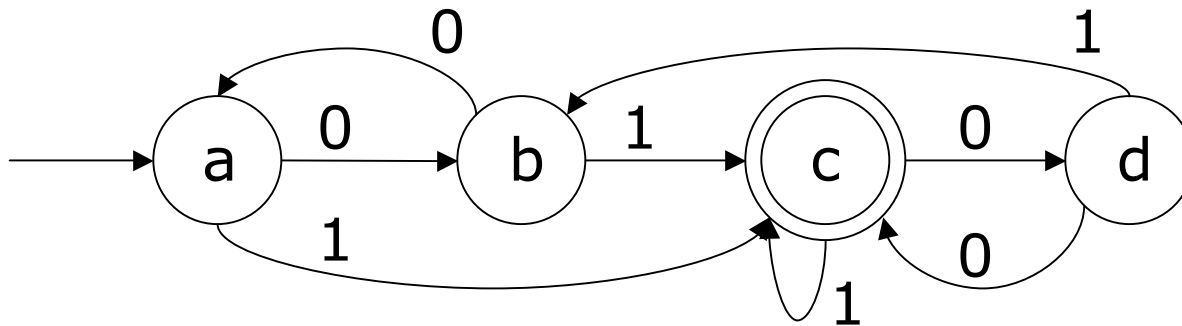
Conclusion: b and d are equivalent.



State Partitioning Algorithm

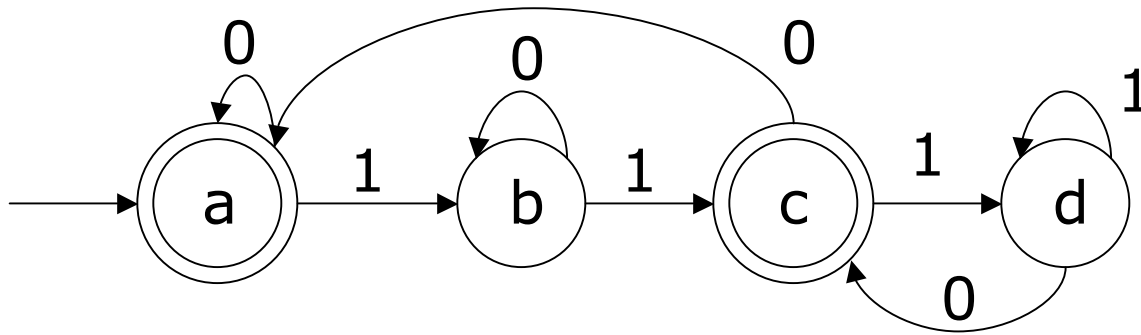
- ❑ This algorithm finds the equivalence classes of states in a DFA.
- ❑ Start with a partition P of the states into accepting and non-accepting.
- ❑ Repeat until there is no change in P :
 - ❑ For each element of P , determine whether for each input symbol the next states are in the same element of P . If not, partition the element into states having the same next-state partition. Replace P with the result of partitioning.

State Partitioning Example 1



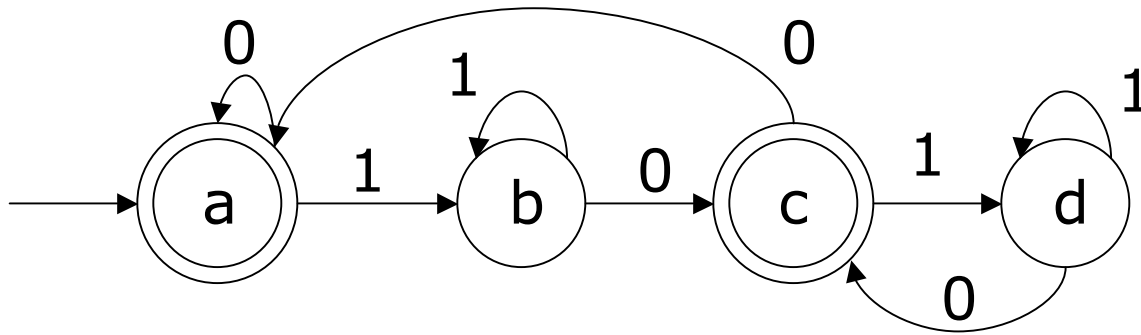
- ❑ $P = \{\{a, b, d\}, \{c\}\}$
- ❑ Next step: $P = \{\{a, b\}, \{c\}, \{d\}\}$,
since $b:1 = c$ but $d:1 = b$ and b and c are in different elements.
- ❑ Now there can be no change.

State Partitioning Example 2



- $P = \{\{a, c\}, \{b, d\}\}$
- Next step: $P = \{\{a, c\}, \{b\}, \{d\}\}$,
since $b:1 = c$ but $d:1 = d$ and c and d are in different elements.
- Next step: $P = \{\{a\}, \{c\}, \{b\}, \{d\}\}$,
since $a:1 = b$ but $c:1 = d$, and b and d are in different elements.
- Now there can be no change. No two states are equivalent.

State Partitioning Example 3



- ❑ $P = \{\{a, c\}, \{b, d\}\}$
- ❑ Each element of P has the indicated next-state property, so the algorithm terminates.
- ❑ a is equivalent to c , b is equivalent to d .



Comments on State Partitioning Algorithm

- ❑ The cases where there is only one set in the partition, or where all sets in the partition are singletons, will not change.
- ❑ We don't need to check singleton sets for splitting.
- ❑ Sets only split, never merge back together.
- ❑ At each step of the algorithm, there must be a split. Otherwise the algorithm terminates. Therefore, the algorithm requires at most $n-1$ steps for an n -state machine.
- ❑ The size of the partition at the start of the i^{th} ($i = 1, 2, \dots$) step must be at least $i+1$ for the algorithm to have another step.

Examples of How States Can Partition

□ Shortest: $\{\{s_1, s_2, \dots, s_n\}\}$ No further splitting possible.

□ Nominal:

$\{\{s_1, s_2, \dots, s_{n-1}\}, \{s_n\}\}$	1 st step
$\{\{s_1, s_2\}, \{s_3, s_4\}, \dots, \{s_{n-1}\}, \{s_n\}\}$	2 nd step
⋮	
$\{\{s_1, s_2\}, \{s_3\}, \{s_4\}, \dots, \{s_{n-1}\}, \{s_n\}\}$	$(n/2)^{\text{nd}}$ step
$\{\{s_1\}, \{s_2\}, \{s_3\}, \{s_4\}, \dots, \{s_{n-1}\}, \{s_n\}\}$	$(1+n/2)^{\text{nd}}$ step

□ Longest:

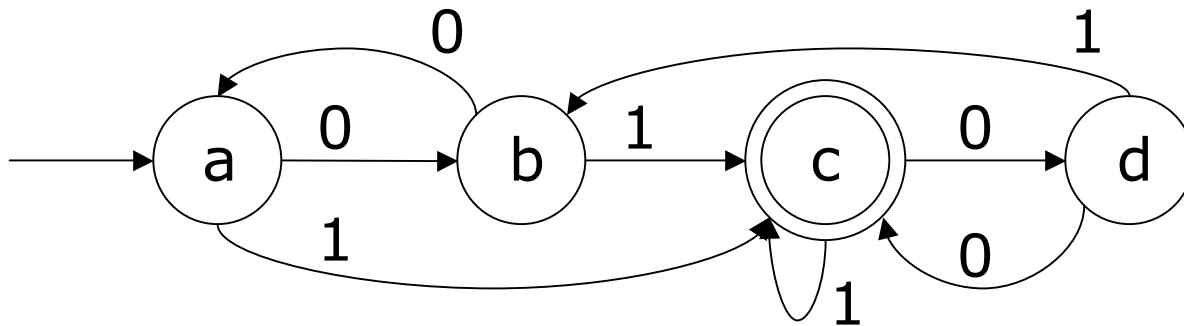
$\{\{s_1, s_2, \dots, s_{n-1}\}, \{s_n\}\}$	1 st step
$\{\{s_1, s_2, \dots, s_{n-2}\}, \{s_{n-1}\}, \{s_n\}\}$	2 nd step
$\{\{s_1, s_2, \dots, s_{n-2}\}, \{s_{n-2}\}, \{s_{n-1}\}, \{s_n\}\}$	3 rd step
⋮	
$\{\{s_1\}, \{s_2\}, \dots, \{s_{n-2}\}, \{s_{n-1}\}, \{s_n\}\}$	$(n-1)^{\text{th}}$ step



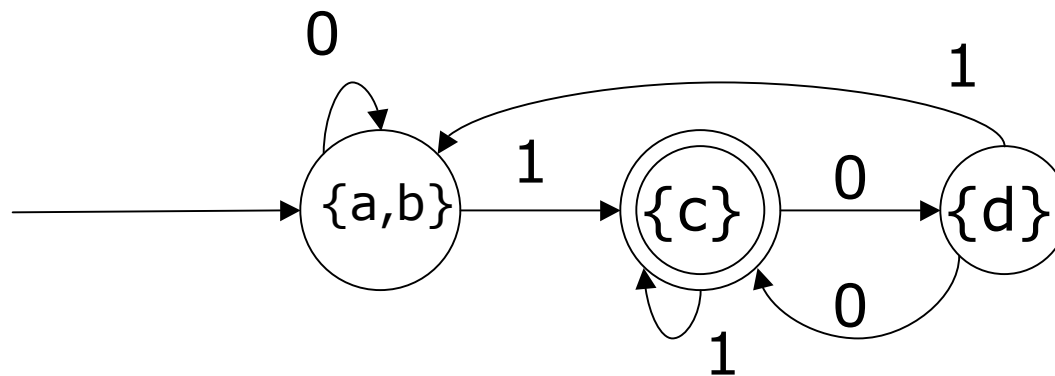
Finding the Minimum State Acceptor

- Once equivalent states have been determined, an equivalent acceptor with the minimum number of states can be constructed.
- Simply let the **equivalence classes** of the original machine **be the states** of the new machine. Define transitions between such states to be consistent with those in the original. Define acceptance to be those classes containing accepting states.

State Minimization Example



$P = \{\{a, b\}, \{c\}, \{d\}\}$





Abstract States of a Language

- By looking at a language, we can tell something about what kind of a machine is required to recognize it.
- For any string x , the **abstract state** of language L for x is the set $L/x = \{y \mid xy \in L\}$.
- If this set is the same for two strings x and x' , then it is ok for the machine to **share the same state** with both inputs x and x' . The “past history” doesn’t matter.
- Otherwise, the machine must **distinguish** x and x' by being in different states after those strings are input.



Examples of L/x

□ Suppose $L = \{01\}^*\{10\}$

$$L/1 = \{0\} = L/011 = L/01011 = \dots$$

$$L/10 = \{\epsilon\}$$

$$L/0 = \{1\}\{01\}^*\{10\}$$

$$L/00 = \emptyset = L/11 = \dots$$

$$L/\epsilon = \{01\}^*\{10\} = L/01 = L/0101 = \dots$$

The Abstract Machine for L

- The abstract states can be used to form an abstract machine. For each string x and letter a there is a transition from L/x to $L/x a$ with label a .

- Example: Suppose $L = \{01\}^*10$

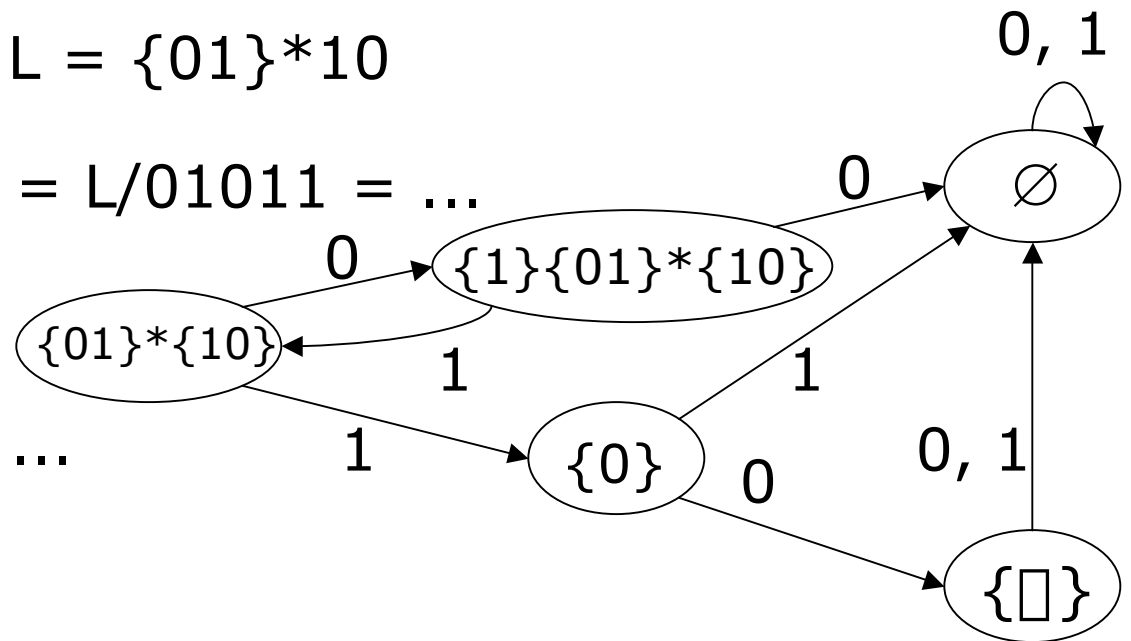
$$L/1 = \{0\} = L/011 = L/01011 = \dots$$

$$L/10 = \{\epsilon\}$$

$$1\{01\}^*10 = L/0 = \dots$$

$$\emptyset = L/00 = \dots$$

$$\{01\}^*10 = L/\epsilon = L/01 = L/0101 = \dots$$





Distinguishability

- Two strings x and x' are called **indistinguishable** under L , written

$$x \equiv_L y$$

provided


$$L/x = L/x'.$$

- Two strings are **distinguishable** iff they are not indistinguishable.



Properties of \equiv_L

- \equiv_L is an **equivalence relation**
- \equiv_L is **right-invariant**, meaning:
$$x \equiv_L y \text{ implies } (\exists z) xz \equiv_L yz$$



Note on Right-Invariance of \equiv_L

- The following are equivalent:
 1. $x \equiv_L y$ implies $(\exists z) xz \equiv_L yz$
 2. $x \equiv_L y$ implies $(\exists \square) x\square \equiv_L y\square$
- That 1 \square 2 is obvious.

That 2 \square 1 requires string induction on z .



Theorem (Myhill-Nerode)

- L is accepted by some DFA iff \equiv_L has a finite number of equivalence classes.
- Proof: This theorem is a summarization of the preceding discussion.



A Language that is not DFA-acceptable

- $L = \{0^n 1^n \mid n \geq 0\}$
- Abstract states $L/0, L/00, L/000, \dots$ are distinct, and there is an infinite number of them.
- The reason that they are distinct is that $L/0^n$ contains 1^m iff $n = m$.



Summary

- ❑ The following are equivalent for a language L:
 - ❑ There is a DFA accepting L.
 - ❑ There is an NFA accepting L.
 - ❑ L is regular.
 - ❑ The equivalence relation \equiv_L has a finite number of equivalence classes.
- ❑ Regarding the fourth point, the classes are effectively the states of the minimal DFA.



NFA's as Language Generators

- ❑ There is another way to interpret an NFA, other than as a language acceptor.
- ❑ This is as a language generator, which is similar to a grammar.
- ❑ The **language generated by** an NFA is the set of all strings that can be created by starting with some starting state and traversing to some accepting state, and concatenating the labels encountered.
- ❑ Clearly the language generated and language accepted by an NFA are exactly the same. Generation just provides a different way of thinking about the NFA.
- ❑ More on this topic when we get to grammars.



The Pumping Lemma

- The pumping lemma provides another way to show that a language is **not** finite-state.
- A language being finite-state means that there is a natural number n such that some n -state machine accepts the language.
- A string of length k accepted by the machine takes the machine through a sequence of $k+1$ states, some of which may be the same.
- If $k \geq n$, then some of those states *must* be the same.
- This implies that there is a *shorter* string that is also accepted by the machine, as well as certain longer strings that are too.



The Pumping Lemma

- Let L be the language accepted by an m -state machine, and x be an accepted sequence of length $\geq m$.
- Then x can be decomposed as a concatenation uvw , where

$$v \neq \epsilon, |v| \leq m$$


and

$$(\forall k \geq 0) uv^k w \in L.$$



Proof of The Pumping Lemma

- Let L be the language accepted by an m -state machine, and x be a sequence of length $\geq m$ accepted by the machine. Let q_0, q_1, q_2, \dots be the states through which the machine moves in accepting that sequence.
- Since there are only m states in the machine, the same state must be repeated. Let r and s be indices of repeated states, i.e. $q_r = q_s$, where $r < s$.
- Let u be the sequence of labels from q_0 , to q_r , v be the sequence of labels from q_r , to q_s , and w be the sequence of labels from q_s , to q_m . The desired properties of uvw are easily seen to be satisfied.



Another proof that $\{0^n 1^n \mid n \geq 0\}$ is not DFA-acceptable

- Suppose the language is acceptable by an m -state machine.
- Let $x = 0^m 1^m$. According to the pumping lemma, x can be written uvw , where $v \neq \epsilon$, and $(\forall k \geq 0) uv^k w \in L$.
- Whatever v is, it is one of the following forms:
 - $0^r, r > 0$
 - $1^s, s > 0$
 - $0^r 1^s, r > 0$ and $s > 0$
- But this is absurd, because for example, $0^{m-r} (0^r)^k 1^m \notin L$ unless $k = 1$.



More Examples of Non-Regular Languages

- $\{x x \mid x \in \{0, 1\}^*\}$
- $\{x x^R \mid x \in \{0, 1\}^*\}$
- $\{0^n 1^m \mid m, n \in \mathbb{N}, m < n\}$
- $\{x \in \{0, 1\}^* \mid \#_0(x) = \#_1(x)\}$
- $\{1^n \mid n \text{ a perfect square}\}$
- $\{1^n \mid n \text{ a prime number}\}$
- The set of palindromes over an alphabet with more than one symbol.
- The set of well-formed strings of parens =
 $\{(), ()(), (()), ()(()), (()()), (())(()), ()()(), \dots\}$



Pumping Lemma Corollary

- The language accepted by a DFA of n states is infinite iff it accepts a string of length $\geq n$.
- Proof:
 - (⇒) If the DFA accepts an infinite language, there must be strings of arbitrary length it accepts.
 - (⇐) If a DFA accepts a string of length $\geq n$, we can generate an infinite number of strings that it also accepts by pumping one such string.



Decision Problems

- A **decision problem** is the problem of finding an algorithm that will answer questions about formal systems, such as automata, regular expressions, grammars, etc.
- If an algorithm exists, the problem is said to be **solvable**, otherwise **unsolvable**. (The words **decidable** and **undecidable** are also used.)
- Later we will see many examples of unsolvable decision problems. But for systems related to **regular languages**, many are solvable.



Examples of Decision Problems

- ❑ Given two DFA's (expressed as a string representation of their graphs) are their languages equal?
- ❑ Given a DFA, is its language empty?
- ❑ Given a DFA, is its language finite?
- ❑ Given two DFA's, is the language of one included in the language of the other?



Solution to Decision Problems

- ❑ To describe the solution of a decision problem, we must give an algorithm.
- ❑ Consider the first decision problem on the previous page:
 - ❑ Given two DFA's (expressed as a string representation of their graphs) are their languages equal?
- ❑ An algorithm for this problem might be as follows:
 - ❑ Consider the union machine of the DFA's: the machine that includes all states and transitions of each. Determine whether what would have been the initial states of each are equivalent using the equivalence checking algorithm. If they are equivalent, then the machines accept the same language. Otherwise they don't.



Solution to the Finiteness Decision Problem

- ❑ An algorithm that will determine whether or not any DFA accepts an infinite set:
- ❑ By the same kind of reasoning as in the pumping lemma, we can show that the language an n -state DFA is infinite iff it accepts a string of length between n and $2n$.
- ❑ Since the number of strings in this range is finite, we can check them all to see if there is an acceptance.
- ❑ Can you devise another algorithm?



A Regular Expression Decision Problem

- It is decidable whether or not two regular expressions denote the same language.
- Proof: Construct DFA's for the regular expressions, then use the corresponding result for DFA's.