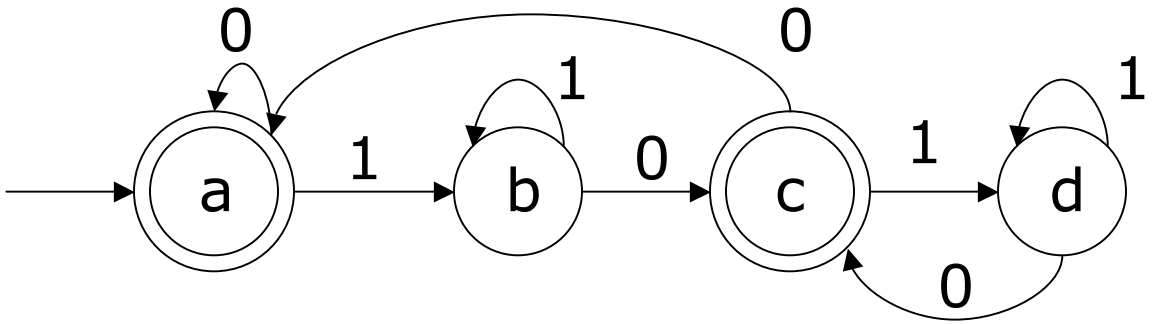
State Equivalence

Robert M. Keller
Harvey Mudd College
March 2010

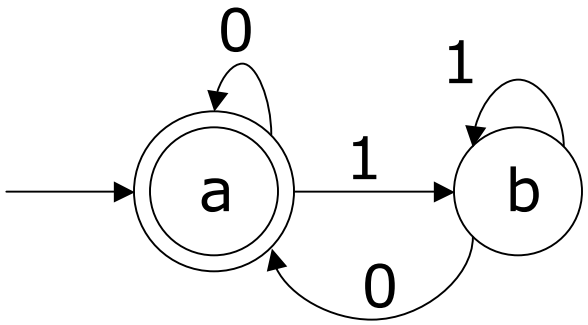
Equivalence of States

- 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



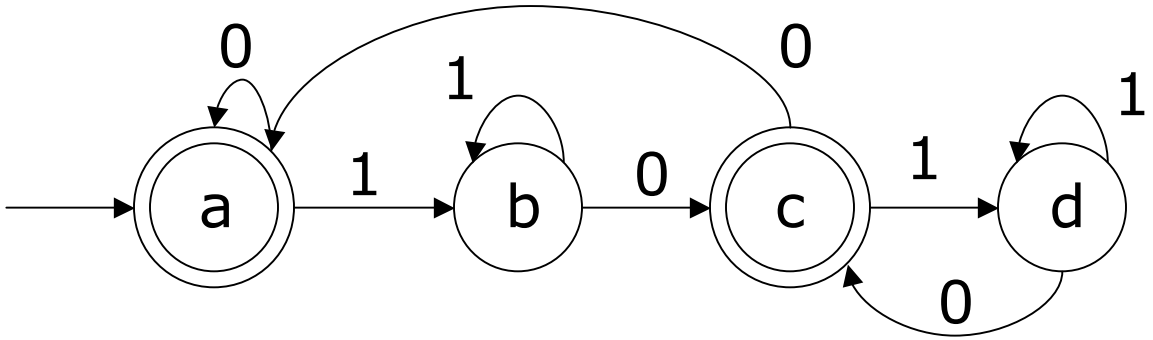
In this example, state a is equivalent to state c and state b is equivalent to state d. The simplified automaton is:

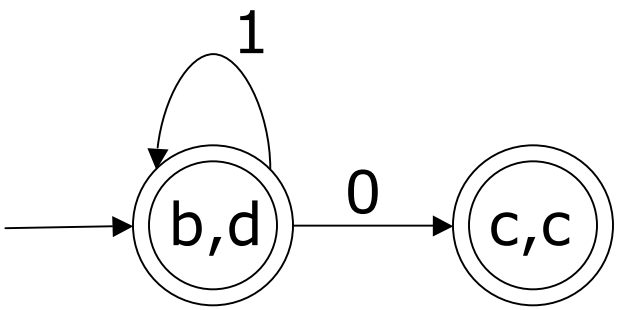


Equivalence Testing

- 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:





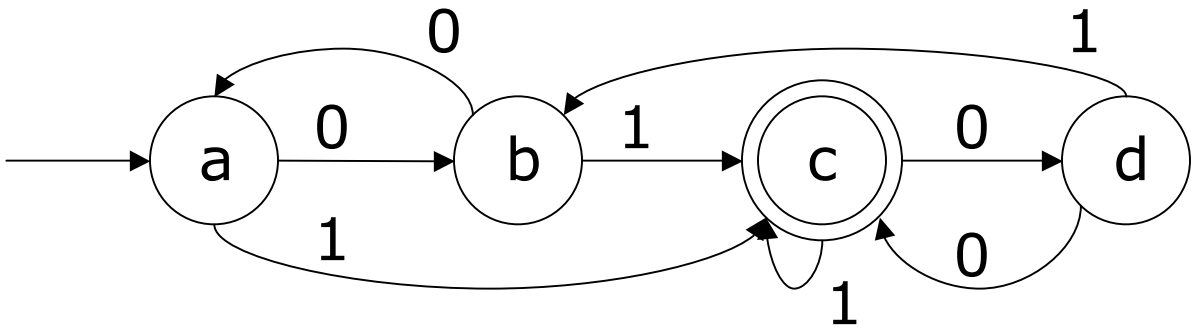
Note: Once we get to a state where both components are the same, we don't need to complete that branch of the construction because the graph will be that of the original machine.

Conclusion: b and d are equivalent.

State Partitioning Algorithm

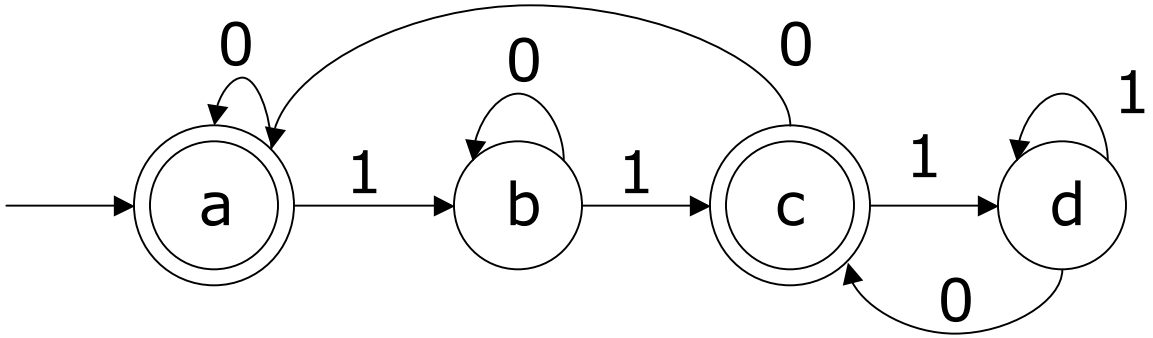
- This algorithm finds the equivalence classes of states in a DFA.
- Start with a partition P of the states into accepting and non-accepting.
- Repeat until there is no change in P:
  - For each element of P, determine whether for each input symbol the next states are in the same element of P. If not,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

State Partitioning Example 1



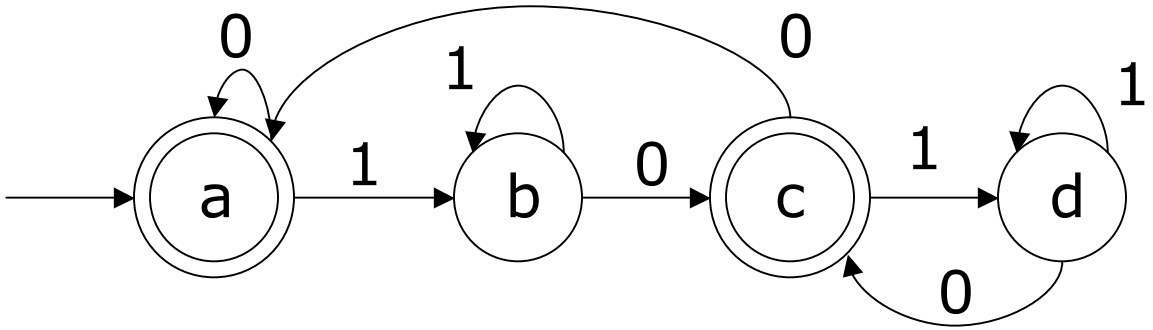
- P = {{a, b, d}, {c}}
- Next step: P = {{a, b}, {c}, {d}},
  since b:1 = c but d:1 = b and b and c are in different elements.
- Now there can be no change.

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, ...) step must be at least i+1 for the algorithm to have another step.

Examples of How States Can Partition

- Shortest: $\{\{s_1, s_2, \ldots s_n\}\}$ No further splitting possible.

- Nominal:

  $\{\{s_1, s_2, \ldots s_{n-1}\}, \{s_n\}\}$ 1st step

  $\{\{s_1, s_2\}, \{s_3, s_4\}, \ldots, \{s_{n-1}\}, \{s_n\}\}$ 2nd step

  . . .

  $\{\{s_1, s_2\}, \{s_3\}, \{s_4\}, \ldots, \{s_{n-1}\}, \{s_n\}\}$ $(n/2)^{nd}$ step

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

- Longest:

  $\{\{s_1, s_2, \ldots s_{n-1}\}, \{s_n\}\}$ 1st step

  $\{\{s_1, s_2, \ldots s_{n-2}\}, \{s_{n-1}\}, \{s_n\}\}$ 2nd step

  $\{\{s_1, s_2, \ldots s_{n-2}\}, \{s_{n-2}\}, \{s_{n-1}\}, \{s_n\}\}$ 3rd step

  . . .

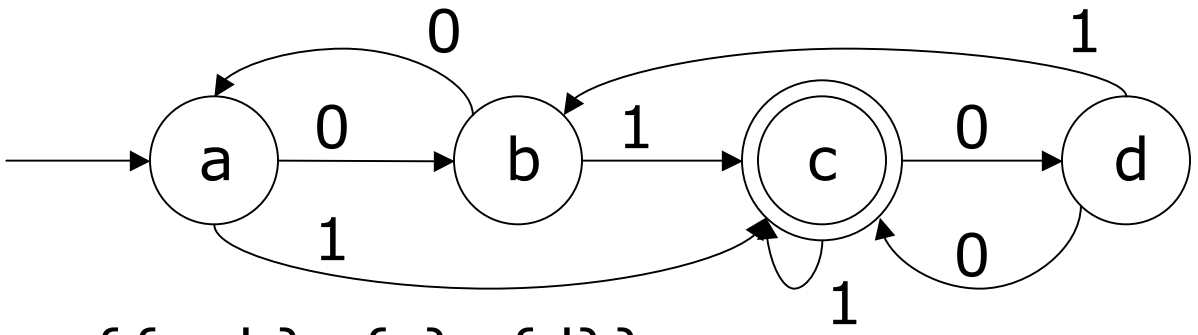
  $\{\{s_1\}, \{s_2\}, \ldots, \{s_{n-2}\}, \{s_{n-1}\}, \{s_n\}\}$ $(n-1)^{th}$ step

Finding the Minimum State Acceptor

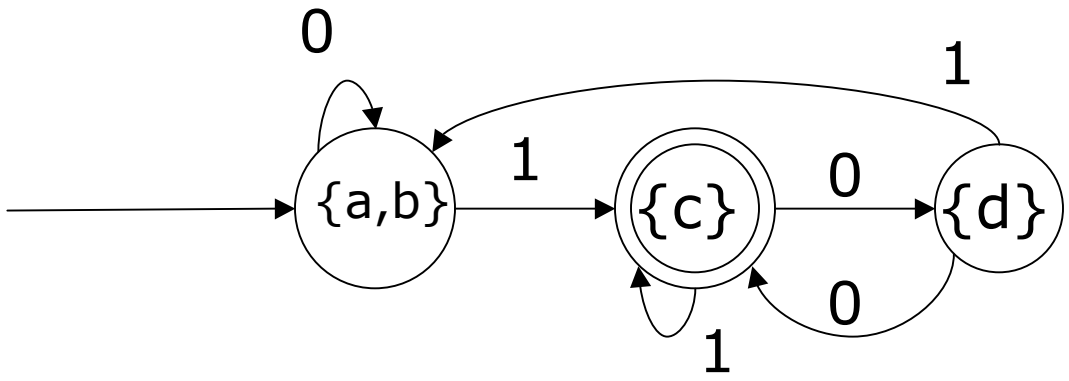
- Once equivalent states have been determined, an equivalent acceptor with the minimum number of states can be constructed.

- 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 = ...

  L/10 = {ε}

  L/0 = {1}{01}*{10}

  L/00 = ∅ = L/11 = ...

  L/ε = {01}*{10} = L/01 = L/0101 = ...

The Abstract Machine for L

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

- Example: Suppose L = {01}*10

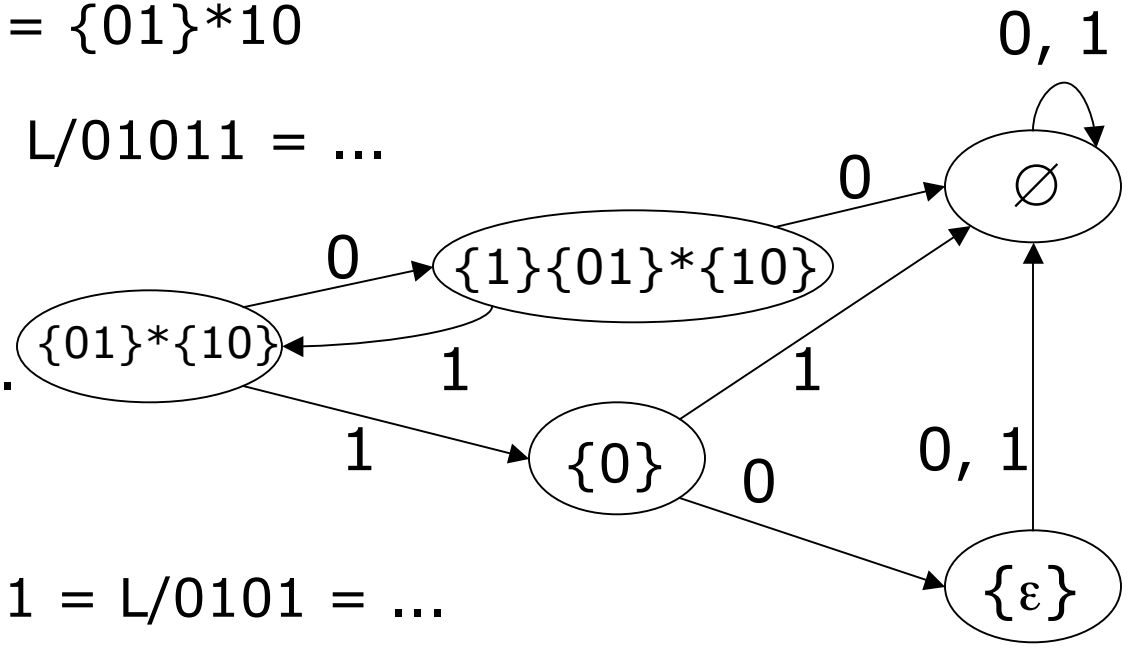
  L/1 = {0} = L/011 = L/01011 = ...

  L/10 = {ε}

  1{01}*10 = L/0 = ...

  ∅ = L/00 = ...

  {01}*10 = L/ε = L/01 = L/0101 = ...



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:

  $x \equiv_L y$ implies $(\forall z)\ xz \equiv_L yz$

Note on Right-Invariance of $\equiv_L$

- The following are equivalent:

  1. $x \equiv_L y$ implies $(\forall z)\ xz \equiv_L yz$
  2. $x \equiv_L y$ implies $(\forall \sigma)\ x\sigma \equiv_L y\sigma$

- That 1 $\Rightarrow$ 2 is obvious.

  That 2 $\Rightarrow$ 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

- $L = \{0^n 1^n \mid n \in N\}$
- Abstract states L/0, L/00, L/000, . . . 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

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

  and

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

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, ... 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 $\{0^n1^n \mid n \in N\}$ is not DFA-acceptable

- Suppose the language is acceptable by an m-state machine.
- Let $x = 0^m1^m$. According to the pumping lemma, x can be written uvw, where $v \neq \varepsilon$, and $(\forall k \in N)\ uv^kw \in L$.
- 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,
  $0^{m-r}\,(0^r)^k1^m \notin L$ unless $k = 1$.

More Examples of Non-Regular Languages

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

Pumping Lemma Corollary

- The language accepted by a DFA of n states is infinite iff it accepts a string of length $\geq$ n.

- Proof:
- ($\Rightarrow$) If the DFA accepts an infinite language, there must be strings of arbitrary length it accepts.

- ($\Leftarrow$)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.